Ghent University
Faculty of Sciences
Department of Applied Mathematics and Computer Science

Generation Algorithms for Mathematical and Chemical Problems

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Jan Goedgebeur

May 2013

Promotor: prof. dr. Gunnar Brinkmann

To Annelies

Dankwoord

Deze thesis zou er zeker niet gekomen zijn zonder de hulp en steun van heel wat mensen. Eerst en vooral wil ik mijn promotor, Gunnar Brinkmann, bedanken voor de uitstekende begeleiding. Zijn deur stond altijd voor mij open en hij gaf altijd met veel plezier en met een engelengeduld antwoorden op mijn vele vragen. Het is onmogelijk om op te sommen wat ik allemaal van hem heb bijgeleerd. Ik heb ook enorm genoten van onze vele fietstochten samen en hoop dat we dit in de toekomst zullen kunnen blijven doen.

Ik wil ook graag de leden van mijn jury bedanken voor het lezen van mijn thesis en voor hun goede suggesties om mijn thesis nog verder te verbeteren.

Mijn vriendin Annelies, mijn ouders en mijn toekomstige schoonouders kan ik nooit genoeg bedanken. Zij hebben me altijd enorm gesteund en gemotiveerd en boden me telkens een luisterend oor.

Ik ben ook mijn collega's en vrienden van de onderzoeksgroep "Combinatorische Algoritmes en Algoritmische Grafentheorie" (CAAGT) heel dankbaar. Tijdens onze wekelijkse CAAGT-lunches hadden we vele interessante discussies. Telkens als ik een Chinees restaurant zie, zal ik aan jullie denken.

Natuurlijk ben ik ook mijn collega's en vrienden van onze vakgroep heel dankbaar voor de leuke sfeer. Tijdens de koffiepauzes heb ik van jullie veel straffe anekdotes gehoord en ook de spelletjes- en filmavonden zal ik nooit vergeten.

In het bijzonder wil ik mijn bureaugenoten Annick, Jean-Marie en Steven bedanken. Het was een waar genoegen om met jullie een bureau te delen. Daarnaast wil ik graag nog eens uitdrukkelijk Michaël bedanken voor onze vele aangename gesprekken over de middag, Stéphanie voor het nalezen van stukken van mijn thesis en Nico voor het goede gezelschap op de conferenties die we samen bijgewoond hebben.

Ik wil zeker ook mijn vrienden van de "reguliere" bedanken met wie ik op regelmatige basis onder het genot van een goede trappist vele wetenschappelijke

en minder wetenschappelijke discussies gevoerd heb.

During my PhD I have also had a wonderful research stay at the Australian National University in Canberra. I was very warmly welcomed there and I immediately felt at home. Brendan, Fateme, Leanne, Miriam, Mirka, Mohammadreza and Pascal, thanks a lot for the great time you gave me in Australia and I hope I will be seeing you again sooner or later.

During the last year of my PhD I also had the honour of collaborating with Professor Staszek Radziszowski on Ramsey numbers. I really learned a lot from him. During this period we exchanged hundreds of emails and I was very happy to meet Staszek in person at two conferences. I also really enjoyed my research stay with him in Rochester where he took great care of me. So Staszek, really thanks a lot for everything!

Tenslotte wil ik ook het Fonds Wetenschappelijk Onderzoek Vlaanderen bedanken voor de vele kansen die ze me geboden hebben door me deze beurs te geven. Ik heb tijdens mijn doctoraat ook veel gebruikgemaakt van de STEVIN supercomputer van de Universiteit Gent. Daarom ben ik het STEVIN-team heel dankbaar voor hun uitstekende ondersteuning.

Jan Goedgebeur
mei 2013

Contents

Summary

In this thesis we develop efficient structure generation algorithms which can be used to solve problems in chemistry and mathematics. To this end we design and implement algorithms to construct mathematical structures (more specifically: *graphs*).

In the **first chapter** we give an introduction about structure generation and introduce the necessary graph theoretical concepts which will be used in this thesis. A graph is a structure which consists of a set of vertices and a set of edges which represent connections between these vertices. A graph can serve as a model for a road network where the vertices represent crossroads and the edges streets. This model can for example be used to determine the shortest route between two crossroads. A graph can also model a molecule. Here the vertices represent atoms and the edges bindings between atoms. A graph can also be used to determine an assignation of teachers to courses. In this case the vertices represent teachers and courses and the edges denote which courses a teacher can give.

The algorithms for the generation of structures are designed to generate structures from a specific graph class very efficiently, for example the class of graphs where all vertices have exactly 3 neighbours. In this thesis we investigate and design *exhaustive* generation algorithms. These are algorithms that make sure that *all* graphs in the class of graphs which we want to generate are enumerated. Our algorithms are also isomorphism-free, i.e. they guarantee that no *isomorphic* copies are output (that are graphs which have the same structure).

Structure generation has applications in amongst others mathematics and chemistry. In mathematics, complete lists of structures with specific properties are used to test mathematical conjectures. In theoretical chemistry such lists are used to determine or predict the structure of molecules. In this thesis we develop algorithms for the generation of complete lists of graphs which have important

applications in mathematics as well as generation algorithms which are important in chemistry.

In the **second chapter** we discuss the generation of 3-regular graphs. These are graphs where every vertex has exactly 3 neighbours. The first complete list of 3-regular graphs was already enumerated in 1889 by J. de Vries: he constructed all 3-regular graphs with up to 10 vertices by hand [42, 43]. Since then, several other scientists have done research on the enumeration of complete lists of 3-regular graphs. Hence this can be considered a benchmark problem in structure enumeration.

Our generation algorithm for 3-regular graphs consists of two parts: first we generate prime graphs (this is a special subclass of the 3-regular graphs) and then we start from these prime graphs to construct all 3-regular graphs. Our implementation of this algorithm is more than 4 times faster than the previously fastest generators for 3-regular graphs. We also extend our algorithm such that 3-regular graphs without triangles or squares can be enumerated efficiently.

In the **third chapter** we adapt our generation algorithm for 3-regular graphs to generate *snarks* efficiently. Snarks form an important subclass of 3-regular graphs. They do not contain triangles and have special colourability properties. It is known that 3 or 4 colours are required to colour the edges of a 3-regular graph such that the 3 edges of every vertex each have a different colour. Snarks do not have such an edge colouring with 3 colours. This class of graphs is especially interesting since for a lot of mathematical conjectures it can be proven that these conjectures are true if and only if they are true for snarks. Often it has even been proven that if there would be a counterexample, the smallest counterexample would be a snark.

Our implementation of this specialised generation algorithm for snarks is more than 14 times faster than the fastest existing generator for snarks. This allowed us to enumerate all snarks up to 34 vertices and all snarks without squares up to 36 vertices. Previously only complete lists of snarks up to 32 vertices were known. Using these new lists we tested 22 (open) mathematical conjectures. Our lists yielded several counterexamples for 8 of these conjectures. This shows that (complete) lists of snarks are not only theoretically a good source for counterexamples to conjectures, but also in practice.

In the **fourth chapter** we discuss an important application of structure generation in chemistry. More specifically, we discuss the generation of (models of) *fullerenes*. These are 3-regular graphs that can be drawn on a sphere and where every face is a pentagon or a hexagon. Here the vertices of the graph

represent carbon atoms. The first (chemical) fullerene was discovered in 1985 by H.W. Kroto and his colleagues [80]. They discovered a fullerene with the shape of a soccer ball: the famous C_{60} buckyball. Kroto and his colleagues later also received the Nobel Prize in Chemistry for this discovery. The carbon nanotubes (these are fullerenes with a tubular shape) are an important subclass of fullerenes. They are very promising for potential applications [45]. For example, researchers are trying to build nanotransistors using nanotubes.

Already a lot of research has been done on the generation of fullerenes, but until now only one successful algorithm for the generation of fullerenes was known. This algorithm was designed by G. Brinkmann and A.W.M. Dress [19]. They also implemented an efficient program – called *fullgen* – based on their algorithm. In this chapter we discuss a new and entirely independent generation algorithm for fullerenes and show how it can be implemented efficiently. Our implementation of this algorithm is more than 3.5 times faster than *fullgen*. Contradictory results of our generator and *fullgen* led to the detection of a programming error in *fullgen*. Due to this error, not all fullerenes were generated starting from 136 vertices. In the meantime this error has been fixed and now the results of *fullgen* and our program are in complete agreement. Using our program we were able to generate all fullerenes up to 400 vertices. This allowed us to prove that the smallest counterexample to the *spiral conjecture* has 380 vertices [30] (which has been an open problem since 1991).

We also design a specialised algorithm for the generation of IPR fullerenes. These are fullerenes which do not contain any adjacent pentagons. IPR fullerenes are especially interesting due to a general tendency to be chemically more stable and thus they are more likely to occur in nature. Also in this case our implementation of this specialised algorithm for the generation of IPR fullerenes is faster than other generators for IPR fullerenes.

In the **last chapter** we discuss another application of structure generation in mathematics, namely the generation of Ramsey graphs in order to determine triangle Ramsey numbers $R(K_3, G)$. A triangle Ramsey graph is a graph which does not contain any triangles and its complement does not contain a given graph G as subgraph. The computation of Ramsey numbers is a difficult computational problem which has already been studied by various researchers and can therefore also be considered a benchmark problem.

In the first part of this chapter we describe an optimised version of an existing algorithm for the generation of *maximal* triangle-free graphs [9]. These are triangle-free graphs where no edges can be added without creating a triangle.

Afterwards we show how this algorithm can be extended to enumerate triangle Ramsey graphs efficiently. This algorithm allowed us to determine all triangle Ramsey numbers up to 30 for graphs of order 10. By combining our computational results with new theoretical results, we were able to determine the triangle Ramsey number of nearly all of the 12 005 168 graphs of order 10, except for 10 of the hardest cases. Because of the rapid growth of Ramsey numbers, the list of triangle Ramsey numbers for graphs of order 10 will very likely be the last list of Ramsey numbers that can be completed for a very long time.

In the second part of this chapter we develop completely different specialised algorithms to improve the upper bounds of classical triangle Ramsey numbers. These are triangle Ramsey numbers $R(K_3, G)$ where the graph G is a complete graph. Using these algorithms we managed to determine improved upper bounds for several classical triangle Ramsey numbers. More specifically, we proved that $R(K_3, K_{10}) \leq 42$, $R(K_3, K_{11}) \leq 50$, $R(K_3, K_{13}) \leq 68$, $R(K_3, K_{14}) \leq 77$, $R(K_3, K_{15}) \leq 87$ and $R(K_3, K_{16}) \leq 98$. All of these new upper bounds improve the old upper bounds by one. We also determine all *critical* Ramsey graphs for K_8 and prove that the known Ramsey graph for K_9 is unique. A Ramsey graph for G is *critical* if it has $R(K_3, G) - 1$ vertices.

Chapter 1

Introduction

A graph is an object which consists of a set of vertices and a set of edges which represent connections between these vertices. A graph can for example serve as a model for a molecule, a road network or a communication network. As a model of a road network it can for example be used to determine the shortest route between two cities. Complete lists of graphs (i.e. lists with all graphs from a given class of graphs) are used in many applications, amongst others in mathematics and in chemistry. Therefore we develop algorithms in this thesis to generate all graphs from a given class efficiently. Such algorithms are called structure generation algorithms and the process of generating all structures (such as graphs) from a given class is called structure generation.

The enumeration of complete lists of structures with given properties has a long tradition. Already in 400 B.C. Theaetetus determined the complete list of regular polyhedra and in 1889 J. de Vries [42, 43] gave a first list of all cubic graphs up to 10 vertices (i.e. graphs where every vertex has exactly three neighbours). Starting from the sixties, computers were also used to generate complete lists of structures.

In mathematics, complete lists of structures with specific properties are used to test conjectures. For example lists of *snarks* (see Chapter 3) are very useful to test conjectures as for a lot of conjectures it can be proven that these conjectures are true if and only if they are true for snarks. Structure generation can also be used to solve problems of "finite nature" such as the computation of specific Ramsey numbers (see Chapter 5).

In chemistry, lists of graphs are used to determine or predict the structure of molecules (i.e. *structure elucidation* and *structure prediction*, respectively). In

structure elucidation, the goal is to identify molecules. By analysing the mass spectrum of a substance, chemists can determine the chemical formula of the molecule of which the substance consists together with forbidden or forced substructures. One can then use a program to generate all structures which represent molecules with this chemical formula and forbidden/forced substructures and compute the spectrum of the generated structures. Then by comparing the computed spectra with the measured spectrum of the substance one can often propose a small list of candidate structures. This helps to determine the structure of the molecule. The DENDRAL project [82] which was initiated in the sixties at Stanford University was the first project for the automatic recognition of the structure of molecules.

In structure prediction, a generation algorithm is used to generate all graphs which represent molecules of a given class which have useful properties. One can then compute the chemical energy of each of the generated structures. The ones with the best energy are chemically the most stable ones and chemists can investigate their potential applications and the possibility to synthesise these molecules.

Programs for generating models of fullerenes (see Chapter 4) have also been used by chemists to investigate the hypothesis that IPR fullerenes are chemically more stable than non-IPR fullerenes [1].

Of course structure generation has also applications outside the field of mathematics or chemistry. Structure generation algorithms have for example already been used to determine the optimal *pitch sequence* of tires. The surface of a tire can contain several types of blocks, e.g. short, medium and long ones. The sequence of these blocks is called the pitch sequence of the tire. One pitch sequence can be more resistant to wear or have a lower noise level than another. So in order to determine the best pitch sequence, one can generate all possible pitch sequences (here the generated structures are not graphs) and compute the resistance to wear or noise level of each generated pitch sequence.

In this thesis we develop algorithms for the generation of complete lists of graphs which have important applications in mathematics as well as generation algorithms which are important in chemistry.

In principle one can generate all graphs from a specific graph class by using an existing algorithm to generate all graphs with a filter at the end which only outputs graphs which are part of the specific graph class. However for most graph classes this approach is very inefficient as often only a tiny percentage of all graphs are part of this graph class. For example only 0.000000057 percent of

the graphs with 12 vertices are cubic and the percentage is decreasing rapidly. So for interesting graph classes it is justified to develop specialised algorithms to generate graphs from these classes efficiently.

Note: several parts of this thesis were taken from articles which were written in collaboration with not only the promotor of this thesis, but also with other co-authors. Parts which were mainly developed and written by other co-authors are clearly marked.

1.1 Definitions and preliminaries

In this section we give the definitions and notations which are used in this thesis. Most of them are widely used in the field of graph theory. For a good introduction to graph theory, we refer the reader to [44] or [120].

Unless stated otherwise, all graphs in this thesis are simple and undirected. Let G be such a graph. The vertex set of G is denoted by $V(G)$ and the edge set of G by $E(G)$. Sometimes we also denote a graph as $G = (V, E)$. Here V represents the set of vertices of G and E the set of edges. We also refer to $|V|$ as the *order* of G.

Two vertices $v, w \in V(G)$ are called *adjacent* if $\{v, w\} \in E(G)$. We say that an edge $e \in E(G)$ is *incident* to $v \in V(G)$ if $v \in e$. Two edges are called *adjacent* if they share a vertex.

The *degree* of a vertex $v \in V(G)$ is the number of edges which are incident to v and is denoted by $deg_G(v)$ (or simply $deg(v)$ if G is fixed). The minimum and maximum degree of vertices in G is denoted by $\delta(G)$ and $\Delta(G)$, respectively. If $\delta(G) = \Delta(G) = d$, G is called d-regular. The set of neighbours (i.e. adjacent vertices) of a vertex $v \in V(G)$ is written as $N_v(G)$ (or $N(v)$ if G is fixed).

A graph G' is a *subgraph* of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. If G' is a subgraph of G and $V(G') = V(G)$, then G' is a *spanning* subgraph of G. If G' is a subgraph of G and $\forall\, v, w \in V(G')$ the following holds: $\{v, w\} \in E(G) \Rightarrow \{v, w\} \in E(G')$, then G' is called an *induced* subgraph of G. We also refer to G' as the subgraph of G induced by the set of vertices $X = V(G')$ (denoted by $G[X]$).

If G' is a subgraph of G, then G is also called a *supergraph* of G'.

The *complement* of a graph $G = (V, E)$ (denoted by G^c), is the graph $G^c = (V, \binom{V}{2} \setminus E)$.

A subgraph P of a graph G is a *path* if $V(P) = \{v_0, v_1, ..., v_n\}$ and $E(P) = \{\{v_i, v_{i+1}\} \,|\, 0 \leq i \leq n\}$ where all vertices of $V(P)$ are distinct. The *length* of the

path is its number of edges.

The distance $d(v, w)$ between 2 vertices $v, w \in V(G)$ is the length of the shortest path from v to w in G. If there is no path from v to w, $d(v, w)$ is defined as ∞. The *diameter* of G is the greatest distance between any two vertices in G.

A subgraph C of a graph G is a *cycle* if $V(C) = \{v_0, v_1, ..., v_n\}$ and $E(C) = \{\{v_i, v_{i+1}\} \mid 0 \leq i < n\} \cup \{v_0, v_n\}$ where all vertices of $V(C)$ are distinct. The *length* of the cycle is its number of edges (or vertices).

A *complete graph* of order n (denoted K_n) is a graph $G = (V, E)$ with $E = \binom{V}{2}$.

A graph G is called *bipartite* if there are two sets of vertices V_1 and V_2 such that $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V(G)$ and $\forall\, e \in E(G) : |e \cap V_1| = |e \cap V_2| = 1$.

A *complete bipartite graph* $G = (V_1 \cup V_2, E)$ is a bipartite graph with partitions V_1 and V_2 such that for any two vertices $v_1 \in V_1$ and $v_2 \in V_2$, $\{v_1, v_2\}$ is an edge in G. The complete bipartite graph with partitions of size $|V_1| = m$ and $|V_2| = n$, is denoted $K_{m,n}$.

The *girth* of a graph G is the length of its shortest cycle and is denoted by $g(G)$. For acyclic graphs the girth is defined as ∞. A *tree* is a connected acyclic graph.

A graph $G = (V, E)$ is *connected* if for every $v, w \in V$ there is a path from v to w. A *disconnected* graph is a graph which is not connected. Given a connected graph $G = (V, E)$. A set $S \subseteq V$ is a *vertex-cut* if $G[V \setminus S]$ is disconnected. If the vertex-cut only contains one vertex, we call that vertex a *cutvertex*. A graph G is k-connected if G is connected and no vertex-cut S exists with $|S| < k$.

Given a connected graph $G = (V, E)$. A set $S \subseteq E$ is an *edge-cut* if the graph $G' = (V, E \setminus S)$ is disconnected. If the edge cut only contains one edge, we call that edge a *bridge*. A graph G is k-edge-connected if G is connected and no edge-cut S exists with $|S| < k$. We call an (edge) cut S with $|S| = k$ a k-(edge)-cut. Thus if a graph is k-edge-connected but not $(k + 1)$-edge-connected, it has a k-edge-cut.

A graph G is *cyclically k-edge-connected* if the deletion of fewer than k edges from G does not create two components both of which contain at least one cycle. The largest integer k such that G is cyclically k-edge-connected is called the cyclic edge-connectity of G and is denoted by $\lambda_c(G)$. If no set of edges can be deleted such that two components are created which both contain at least one cycle, the cyclic edge-connectivity is defined as ∞ (so $\lambda_c(K_4) = \infty$).

The *chromatic index* $\chi'(G)$ of a graph G is the minimum number of colours required for an edge colouring of that graph such that no two adjacent edges have the same colour.

A k-factor of a graph G is a spanning k-regular subgraph of G. A 1-factor is also called a perfect matching.

A graph is called *hamiltonian* if it contains a hamiltonian cycle, this is a cycle which contains all vertices of the graph.

An *independent set* of G is a set of vertices $S \subseteq V(G)$ such that $\forall\, v, w \in S : \{v, w\} \notin E(G)$. The size of the largest independent set of G is called the *independence number* of G and is denoted by $\alpha(G)$.

Two graphs G and G' are *isomorphic* (denoted by $G \cong G'$) if and only if there is a bijective function $\phi : V(G) \to V(G') : \{v, w\} \in E(G) \iff \{\phi(v), \phi(w)\} \in E(G')$. The function ϕ is called an *isomorphism* from G to G'. An isomorphism from G to itself is called an *automorphism*. The *automorphism group* $\mathrm{Aut}(G)$ of G is the group of all automorphisms of G.

If the automorphism group only contains one element (namely the identity automorphism which maps every vertex to itself), we call that group *trivial*.

The orbit of a vertex $v \in V(G)$ is the set of vertices to which v can be mapped by an automorphism of G.

In this thesis we also use several definitions which are specific for *plane* graphs. These definitions are introduced in Chapter 4.

For an overview of the notations which are used in this thesis, see Appendix A.

1.2 Exhaustive generation

All algorithms described in this thesis are designed to generate graphs from a specific class of graphs (e.g. cubic graphs, fullerenes,...) efficiently. All of these algorithms generate these classes of graphs *exhaustively*, i.e. they generate *all* graphs of a given graph class (for a given number of vertices). The graphs are obtained by recursively applying construction operations to a set of initial graphs.

A *construction* or *expansion operation* is an operation which constructs a larger graph from a given graph. A *reduction operation* is an operation inverse to a construction operation. We call the graphs which are obtained by applying a construction operation to a smaller graph *children* and this smaller graph where these children are obtained from the *parent*. We also call the graph obtained by applying an expansion to a graph an *expanded* graph.

To prove that all graphs of a given class can be generated by our algorithm, we usually prove that each graph from the class of graphs (except for a small number of irreducible graphs) can be reduced to a smaller graph of the same class by our reduction operations.

1.3 Isomorphism-free generation

When generating combinatorial structures, it is important that no isomorphic copies are output by the generation algorithm. In the literature various techniques exist to avoid the generation of isomorphic copies. The most commonly used ones are McKay's canonical construction path method [89] and the method of generating canonical representatives (which was independently pioneered by Faradžev [50] and Read [105]). A good survey of isomorphism rejection techniques can be found in [14].

Both the method of generating canonical representatives and the canonical construction path method use a *canonical labelling*.

If $\mathcal{L}$ is the set of all labelled graphs, then a canonical representative function is a function $c : \mathcal{L} \to \mathcal{L}$ with the properties that for every $G \in \mathcal{L}$, $c(G)$ is isomorphic to G and that $c(G) = c(G')$ if and only if G and G' are isomorphic. The graph $c(G)$ is also called the *canonical representative* of G. An isomorphism ϕ from G to $c(G)$ is called a *canonical labelling* of G.

We can implement our own canonical representative function or we can use a program such as *nauty* [88] to compute a canonically labelled isomorph of a given graph. A labelling can for example be represented by a string consisting of the adjacency matrix of the graph written into one row. The adjacency matrix of a graph with n vertices is an $n \times n$ matrix where the entry of row i and column j is 1 if and only if there is an edge between vertex i and vertex j, else the entry is 0. One could then define the labelling of the canonical representative of G to be e.g. the one with the lexicographically minimal representation.

A straightforward isomorphism rejection technique is isomorphism rejection by lists. In this method, a list of non-isomorphic graphs which were generated so far is stored (preferably in the memory for efficiency reasons). Each time a new graph is generated, it is compared to the graphs from the list. If it is not isomorphic to any of the graphs in the list, the graph is added to the list, else it is rejected. Usually the canonical representative is stored rather than the generated graph, since then one does not have to recompute the canonical representative of the stored graphs. When the canonical representative of the new graph is computed, one only has to compare it to the stored canonical representatives.

This approach is of course only feasible if the number of non-isomorphic graphs is not too big. We use this technique for the generation of irreducible IPR fullerenes (see Section 4.3) and minimal Ramsey graphs (see Section 5.3).

The method of generating canonical representatives only accepts graphs if their labelling is canonical. The difficulty of this method is to define a canonical

labelling in such a way that it allows an early bounding criterion with respect to the canonicity criterion. Therefore the canonical labelling and the construction operations must be compatible to make it possible to detect at an early stage that a given partial graph cannot lead to a canonically labelled graph. Algorithms for generating canonical representatives which use such an early bounding criterion are sometimes also referred to as Read/Faradžev-type orderly algorithms. This method is amongst others used in Brinkmann's generator for cubic graphs [11] (called *minibaum*) and in Meringer's generator for regular graphs [95] (called *genreg*).

Before we go into the details of the canonical construction path method, we first discuss how isomorphic copies can occur. This can happen in two different ways: isomorphic graphs which are obtained from the same parent and isomorphic graphs which are obtained from different parents.

We illustrate this for the example of generating *cubic* or *3-regular* graphs. These are graphs where each vertex has degree 3. The basic edge insertion operation to make a cubic G' with n vertices from a cubic graph G with $n-2$ vertices is shown in Figure 1.1. It is the insertion of a new edge between new vertices inserted in two different edges of G (see Chapter 2 for more details). Two edge insertions of a graph G are called *equivalent* if there is an automorphism of G mapping the pairs of edges to which the operation is applied onto each other.

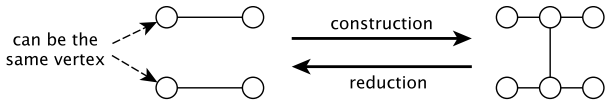


Figure 1.1: The basic edge insertion operation for cubic graphs.

Isomorphic children can be obtained from the same parent: applying the edge insertion operation to pairs of edges of the parent graph which are equivalent under the automorphism group of the parent graph yields isomorphic children. For example applying the edge insertion operation to the edge pair $\{\{0,1\},\{0,2\}\}$ or $\{\{1,2\},\{1,3\}\}$ (and many other edge pairs) of the graph in Figure 1.2 both yield the same (isomorphic) graph.

Isomorphic children can also be generated from the same parent by applying the edge operation to non-equivalent pairs of edges of the parent graph G. This can be seen in Figure 1.3: edge pairs $\{\{0,1\},\{2,3\}\}$ and $\{\{4,5\},\{6,7\}\}$ are not equivalent under the automorphism group of G, but applying the edge insertion operation to each of them yields the same (isomorphic) graph. This is sometimes

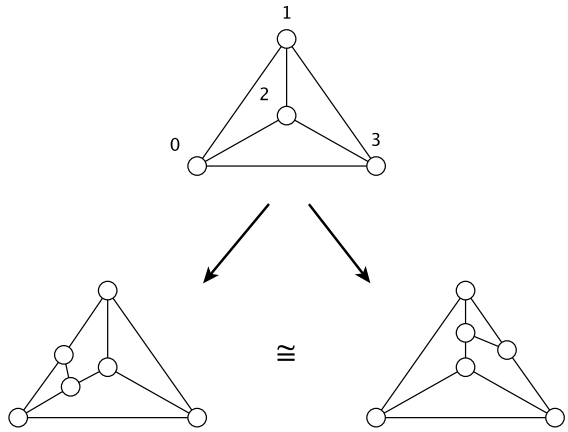


Figure 1.2: A parent graph with isomorphic children.

called *pseudosimilarity* of construction operations.

Isomorphic graphs can also be obtained from different parents. For example when the basic edge insertion operation is applied to the dashed pairs of edges of the non-isomorphic graphs G and G' of Figure 1.4, they yield two isomorphic graphs: H and H', respectively.

The canonical construction path method takes care of all of these sources of isomorphism. When using this method, we first have to define, for every graph G of the class of graphs which we want to generate, a *canonical reduction* which is unique up to isomorphism. We call the graph which is obtained by applying the canonical reduction to G the *canonical parent* of G and an expansion that is the inverse of a canonical reduction a *canonical expansion*. The two rules of the canonical construction path method are:

1. Only accept a graph if it was constructed by a canonical expansion.

2. For every graph G to which construction operations are applied, only perform one expansion from each equivalence class of expansions of G.

Note that accepting a graph if it was constructed from a canonical parent would not be sufficient, as this does not eliminate pseudosimilarity. A *canonical* graph is a graph which was constructed by a canonical expansion.

The canonical construction path method is applied in most of the generation algorithms which are described in this thesis. More specifically, it is used for

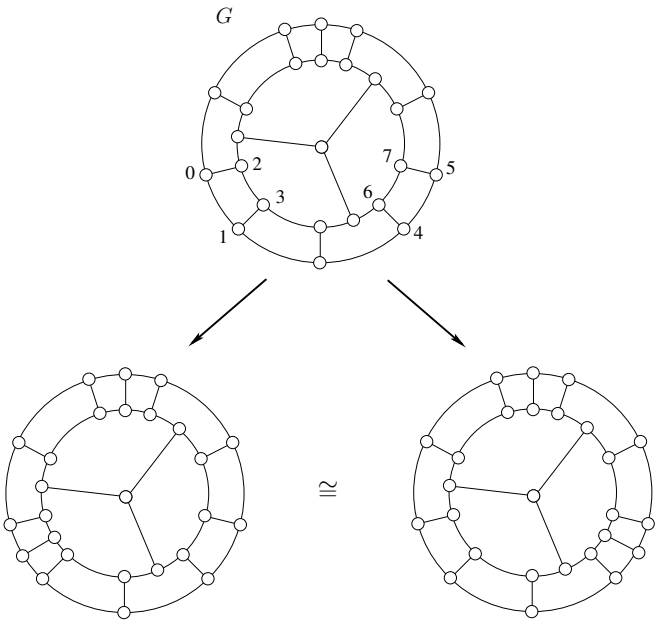


Figure 1.3: A parent graph with isomorphic children obtained by non-equivalent expansions.

the generation of cubic graphs (see Chapter 2), fullerenes (see Chapter 4) and maximal triangle-free graphs (see Chapter 5).

The coarse structure of an algorithm to recursively generate all non-isomorphic graphs of a given class using the canonical construction path method is given as pseudocode in Algorithm 1.1.

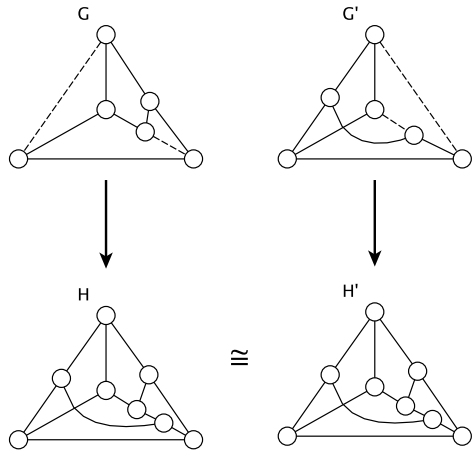


Figure 1.4: A graph which can be obtained from multiple parents.

Algorithm 1.1 Construct(graph G)

```
if G has the desired number of vertices then
  output G
else
  find expansions
  compute classes of equivalent expansions
  for each equivalence class do
    choose one expansion X
    perform expansion X
    if expansion is canonical then
      Construct(expanded graph)
    end if
    perform reduction X^{-1}
  end for
end if
```

Chapter 2

Generation of cubic graphs

Hier fängt die Geschichte an.

Walter Moers, Die Stadt der Träumenden Bücher

In this chapter we describe a new algorithm for the efficient generation of all non-isomorphic connected cubic graphs. Our implementation of this algorithm is more than 4 times faster than previous generators for cubic graphs. The generation can also be efficiently restricted to cubic graphs with girth at least 4 or 5.

In Chapter 3 we describe how this algorithm can be extended to generate snarks efficiently.

Remark: our algorithm for the generation of cubic graphs is also described in [28]. This was joint work also with Brendan D. McKay and various parts of this chapter were adopted from that article.

2.1 Introduction

Cubic or *3-regular* graphs are (simple) graphs where each vertex has degree 3.

Some examples of cubic graphs are given in Figure 2.1. Figure 2.1a depicts K_4, Figure 2.1b the Petersen graph and Figure 2.1c the buckminsterfullerene C_{60}.

When we speak about graphs in the rest of this chapter, we always mean cubic graphs (unless specified otherwise). For a cubic graph G, the number of edges $|E(G)|$ is equal to $\frac{3 \cdot |V(G)|}{2}$.

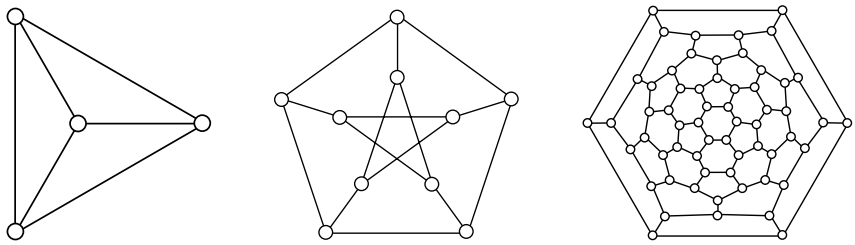


Figure 2.1: Some examples of cubic graphs: K_4, the Petersen graph and the C_{60} fullerene respectively.

The class of cubic graphs is especially interesting for mathematical applications because for various important open problems in graph theory, the smallest or simplest possible potential counterexamples are cubic graphs (see Chapter 3 for more information). In chemistry, cubic graphs serve as models for e.g. the Nobel Prize winning fullerenes [80] (see Chapter 4) or, more generally, for some cyclopolyenes [5].

The generation of cubic graphs can be considered a benchmark problem in structure enumeration. The first complete lists of cubic connected graphs were given by de Vries at the end of the 19th century, who gave a list of all cubic (connected) graphs up to 10 vertices [42, 43]. The first computer approach was by Balaban, a theoretical chemist. He generated all cubic graphs up to 12 vertices in 1966/67 [5]. De Vries' lists were independently confirmed by hand by Bussemaker and Seidel in 1968 [38] and Imrich in 1971 [70]. From 1974 on, various algorithms for the generation of cubic graphs were published. Each algorithm was implemented in a computer program that could generate larger lists of cubic graphs, see [97],[50],[37],[92],[11]. In 1983 Robinson and Wormald [106] published a paper on the non-constructive enumeration of cubic graphs.

When present research began, the fastest publicly available program for the generation of cubic graphs was *minibaum* [11]. When developed in 1992, *minibaum* could be used to generate complete lists of all cubic graphs up to 24 vertices and several more restricted subclasses of cubic graphs with more vertices, like cubic bipartite graphs or cubic graphs with higher girth. Later, when more and faster computers were available, *minibaum* was used to generate all cubic graphs up to 30 vertices in order to test them for Yutsis decompositions [2].

In 1999, Meringer [95] published a very efficient algorithm for the generation

of regular graphs of given degree. But for the generation of all cubic graphs the program based on his algorithm is slower than *minibaum*. In 2000, Sanjmyatav [110] and her supervisor McKay developed a set of very fast specialised programs for various classes of cubic graphs. Unfortunately these programs were never released or published. In this chapter we describe an algorithm based on ideas already used by de Vries and Sanjmyatav/McKay but also with several new ideas. Our implementation of this algorithm is faster than any previous program.

The algorithm is described in Sections 2.2-2.5. In Section 2.6 we describe how we adapted the algorithm to generate cubic graphs with a non-trivial lower bound on the girth efficiently and in Section 2.7 we describe how the algorithm can be used to generate cubic graphs with certain connectivity requirements. We conclude this chapter with a comparison of our algorithm with previous generators for cubic graphs.

2.2 The generation algorithm

Our basic construction operation to make a cubic graph G' with n vertices from a cubic graph G with $n-2$ vertices is the insertion of a new edge between new vertices inserted in two different edges of G. This operation was already used by de Vries and can be seen in Figure 2.2. The inverse operation involves removing an edge $\{x, y\}$ and its endpoints x and y, then adding an edge between the two vertices other than y that were previously adjacent to x, and an edge between the two vertices other than x that were previously adjacent to y. We call this an *edge reduction operation* in case the resulting graph is a connected cubic graph. In that case we call the edge $e = \{x, y\}$ *reducible*, otherwise *irreducible*. So e is irreducible if and only if it is a bridge, has an endpoint in a triangle that does not contain e, or has two endpoints in the same 4-gon that does not contain e. The latter two situations are depicted in Figure 2.3.

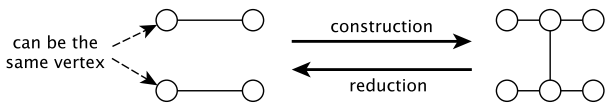
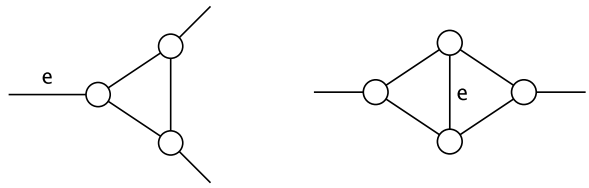


Figure 2.2: The basic edge insertion operation.

A cubic connected graph without reducible edges is called a *prime graph*. By definition, each connected cubic graph can be constructed from a prime graph by



(a) **(b)**

Figure 2.3: Two examples of an irreducible edge e: an edge which has an endpoint in a triangle that does not contain e (i.e. Figure 2.3a) and an edge with two endpoints in the same 4-gon that does not contain e (i.e. Figure 2.3b).

recursive application of the edge insertion operation. The class of all connected cubic graphs is constructed in two steps:

$$\boldsymbol{K_4}$$
$$\Downarrow \text{ (operations to generate prime graphs)}$$
$$\textbf{Prime graphs}$$
$$\Downarrow \text{ (edge insertion operation)}$$
$$\textbf{All cubic graphs}$$

The construction operations to generate prime graphs are described in Section 2.3. The second step is described in more detail in Sections 2.4 and 2.5. In both steps we use the canonical construction path method (see Section 1.3) to make sure no isomorphic graphs are output by the algorithm. In those sections we also describe how this isomorphism rejection technique is applied for the generation of cubic graphs.

2.3 Generation of prime graphs

2.3.1 Introduction

We refer to a subgraph of a graph isomorphic to $K_4 - e$ (with e an edge of K_4) as a K_4^-. The two vertices in a K_4^- that have degree 2 in this subgraph are called *extremal vertices*. This is shown in Figure 2.4a, where the extremal vertices are labelled v_1 and v_2. K_4 with an edge subdivided by inserting a vertex of degree 2 is referred to as K_4^+. The vertex in a K_4^+ which has degree 2 in this subgraph is

called the *central vertex*. This is shown in Figure 2.4b where the central vertex is labelled v_c.

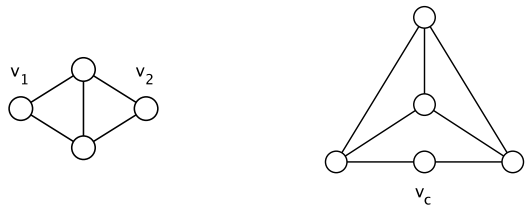


Figure 2.4: A K_4^- and a K_4^+

Lemma 2.1. *A connected cubic graph is a prime graph if and only if each edge is a bridge or has an endpoint in a* K_4^-.

Proof. The direction $\Leftarrow$ is immediate, because neither bridges nor edges with endpoints in a K_4^- can be reduced.

So suppose that $e = \{x, y\}$ is an edge in a prime graph G which is not a bridge and has no endpoints in a K_4^-. Note that if an edge has two endpoints in the same cycle, the edge reduction applied to this edge can reduce the length of the cycle by at most 2. If the endpoints of the reduced edge are not part of the same cycle, the edge reduction can reduce the length of each cycle in which the reduced edge has endpoints by at most 1. Also note that if an edge in a prime graph has two endpoints in the same square, it is part of a K_4^-. Thus if e has no endpoints in a triangle, then e is reducible. On the other hand, if x is contained in a triangle, but not in a K_4^-, each edge of the triangle is reducible. □

We use the three construction operations from Figure 2.5 to generate all prime graphs. We also refer to operation (a) as the *lollipop insertion operation*, operation (b) as the *edge diamond insertion operation* and operation (c) as the *non-adjacent edge diamond insertion operation.*

We call a K_4^+ which is part of a prime graph and which is reducible by a lollipop reduction a *reducible lollipop*. A K_4^- which is part of a prime graph and which is reducible by an edge diamond reduction a *reducible edge diamond* and similarly a K_4^- which is reducible by a non-adjacent edge diamond reduction is called a *reducible non-adjacent edge diamond*. Reductions are only valid if the

reduced graph is a connected cubic graph without parallel edges or loops. Note that each reducible non-adjacent edge diamond is also a reducible edge diamond.

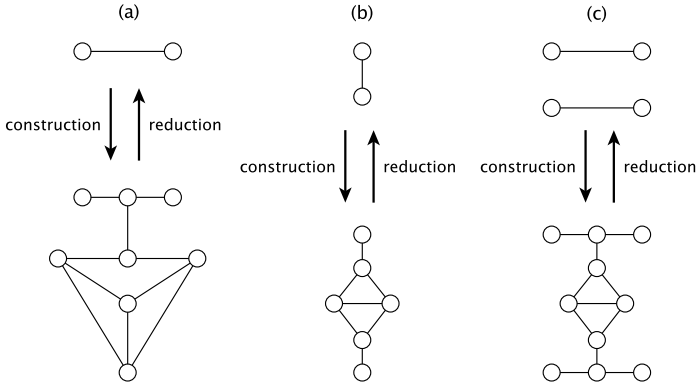


Figure 2.5: The construction operations for prime graphs.

To prove that all prime graphs can be generated by our construction operations, we will show in Section 2.3.2 that each prime graph can be reduced to a smaller prime graph by one of the reduction operations (i.e. Lemma 2.2). Note that the class of prime graphs is not closed under these construction operations (for an example, see Figure 2.6). So after applying an operation it must be tested whether the new graph is a prime graph or not. We use some simple look-aheads that avoid the construction of non-prime graphs. These look-aheads and other details about the construction are explained in Section 2.3.3. In Section 2.3.2 we explain how we make sure that the algorithm does not output isomorphic prime graphs and prove that our algorithm generates exactly one representative of every isomorphism class of prime graphs.

2.3.2 Isomorphism rejection

We use the canonical construction path method (see Section 1.3) to make sure no isomorphic prime graphs are generated. We will now describe how to apply the canonical construction path method for the generation of prime graphs.

Note that the lollipop insertion operation and the edge diamond insertion operation are applied to edges, while the non-adjacent edge diamond insertion operation is applied to pairs of non-adjacent edges. Two lollipop (or edge diamond) expansions of a prime graph G are *equivalent* if there is an automorphism

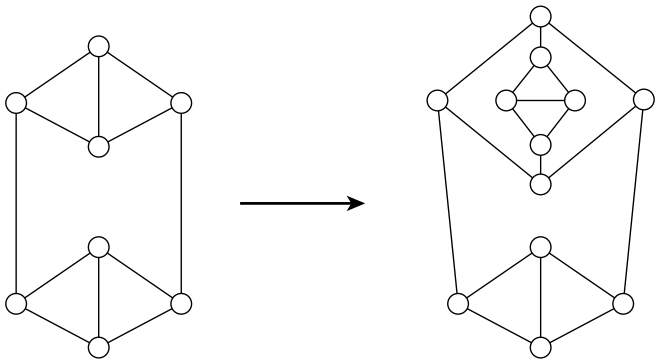


Figure 2.6: Application of the edge insertion operation which yields a non-prime graph.

of G mapping the edges onto each other (i.e. they are in the same orbit of edges of the automorphism group of G). Two non-adjacent edge diamond expansions are equivalent if there is an automorphism of G mapping the pairs of edges onto each other.

Recall from Section 1.3 that we first have to define a canonical reduction which is unique up to isomorphism and that the two rules of the canonical construction path method (applied to the generation of prime graphs) are:

1. Only accept a prime graph if it was constructed by a canonical expansion.
2. For every prime graph G to which construction operations are applied, only perform one expansion from each equivalence class of expansions of G.

A canonical expansion is the inverse of a canonical reduction. The three construction operations for prime graphs were introduced in Section 2.3.1. We will now introduce the concepts of *canonical* reducible lollipops, edge diamonds and non-adjacent edge diamonds in order to define a canonical reduction. The concepts of central vertex, extremal vertex and reducible lollipop, edge diamond and non-adjacent edge diamond were already introduced in Section 2.3.1 and $\mathrm{Aut}(G)$ stands for the automorphism group of a graph G.

Definition 2.1 (Canonical reducible lollipop)**.** *Given a graph G and a canonical representative function f. We call a reducible lollipop of G canonical if it is in the same orbit of* $\mathrm{Aut}(G)$ *as the reducible lollipop which contains the central vertex with the largest label in $f(G)$ (i.e. the canonically labelled representative of G).*

We use the program *nauty* [88] to compute a canonically labelled isomorph of a given graph. Note that it is of no importance that we chose the reducible lollipop with the central vertex with the *largest* label in the canonically labelled graph. We also could have chosen the one with the smallest label or have used another criterion. The only requirement is that it is uniquely defined up to isomorphism.

Definition 2.2 (Canonical reducible non-adjacent edge diamond)**.** *Given a graph G and a canonical representative function f. We call a reducible non-adjacent edge diamond of G canonical if it is in the same orbit of* $\mathrm{Aut}(G)$ *as the reducible non-adjacent edge diamond which contains the extremal vertex with the largest label in $f(G)$.*

Definition 2.3 (Canonical reducible edge diamond)**.** *Given a graph G and a canonical representative function f. We call a reducible edge diamond of G canonical if it is in the same orbit of* $\mathrm{Aut}(G)$ *as the reducible edge diamond which contains the extremal vertex with the largest label in $f(G)$.*

Definition 2.4 (Canonical reduction for prime graphs)**.** *Given a prime graph G with more than 4 vertices.*

- *If G contains a reducible lollipop, it is reduced by applying the lollipop reduction operation to a canonical reducible lollipop.*
- *If G contains no reducible lollipops but at least one reducible non-adjacent edge diamond, it is reduced by applying the non-adjacent edge diamond reduction operation to a canonical reducible non-adjacent edge diamond.*
- *If G contains no reducible lollipops and no reducible non-adjacent edge diamonds, it is reduced by applying the edge diamond reduction operation to a canonical reducible edge diamond.*

So the lollipop operation has the highest priority, the non-adjacent edge diamond operation the second highest priority and the edge diamond operation the lowest priority.

Lemma 2.2. *Every prime graph G with more than 4 vertices is reducible by the canonical reduction and the reduced graph is also prime.*

Proof. If the canonical reduction is a lollipop (or a non-adjacent edge diamond) reduction, the new edge (respectively edges) is a bridge or contains vertices of a K_4^-. Thus the reduced graph is also prime.

So suppose G contains no reducible lollipops and no reducible non-adjacent edge diamonds. If G contains a reducible edge diamond, the canonical reduction is an edge diamond reduction. Suppose that the reduced graph G' obtained by applying the edge diamond reduction to a canonical reducible edge diamond is not prime. This means that the edge e by which the reducible edge diamond has been replaced by the canonical reduction is reducible. So e is not a bridge and has no endpoints in a K_4^-. But this implies that G also had a non-adjacent edge diamond reduction, so the edge diamond reduction was not canonical.

If G also does not contain any reducible edge diamonds, applying the edge diamond reduction to any K_4^- of G yields a cubic graph with parallel edges or loops. Let e be the edge by which a K_4^- has been replaced by the edge diamond reduction. If e is parallel with an edge e', e' was also present in G. But e and e' are no bridges and do not have an endpoint in a K_4^-, so this would imply that G was not prime. Now suppose e is a loop. Let v be the endpoint of e and let w be the other neighbour of v in the reduced graph G'. If w is an extremal vertex of a K_4^- this would mean that this K_4^- was a reducible edge diamond in G. So suppose w is not an extremal vertex of a K_4^-. Since G is prime, this means that G contained a reducible lollipop with central vertex v, contrary to our assumption.

So every prime graph with at least 4 vertices is reducible by the canonical reduction. □

The construction is the inverse of the reduction, so a prime graph is only accepted if it was constructed from a graph isomorphic to its canonical parent by the inverse of the canonical reduction (i.e. if it was constructed by a canonical expansion).

We also have to take the second rule of the canonical construction path into account. This rule says:

2. For every graph G to which construction operations are applied, only perform one expansion from each equivalence class of expansions of G.

The lollipop insertion operation and the edge diamond insertion operation are applied to edges, while the non-adjacent edge diamond insertion operation is applied to pairs of non-adjacent edges.

Recall that two lollipop (or edge diamond) expansions of a prime graph G are equivalent if there is an automorphism of G mapping the edges onto each other. Two non-adjacent edge diamond expansions are equivalent if there is an automorphism of G mapping the pairs of edges onto each other.

We use *nauty* [88] to compute the automorphism group of a graph G. *Nauty* does not compute orbits of edges or pairs of edges of G under the action of $\mathrm{Aut}(G)$ by default, but by using the *userautomproc* option in *nauty*, we can save the generators of $\mathrm{Aut}(G)$. Using these generators we can compute the orbits of edges or pairs of edges (e.g. by using a union-find algorithm).

The class of prime graphs is not closed under the three construction operations. So after applying a construction operation we still have to test whether the constructed graph is still prime. If it is not prime, the graph is not accepted by the algorithm.

The coarse pseudocode for generating all non-isomorphic prime graphs is given in Algorithm 2.1 (cf. Algorithm 1.1 from Section 1.3).

Algorithm 2.1 Construct_prime_graphs(prime graph G)

```
output G
if G has less vertices than the desired number of vertices then
  find expansions
  compute classes of equivalent expansions
  for each equivalence class do
    choose one expansion X
    perform expansion X
    if expansion is canonical and expanded graph is prime then
      Construct_prime_graphs(expanded graph)
    end if
    perform reduction X^{-1}
  end for
end if
```

Theorem 2.3. *When the generation algorithm for prime graphs which was just described (cf. Algorithm 2.1) is recursively applied to K_4, then exactly one representative of each isomorphism class of cubic connected prime graphs is accepted.*

Proof. The proof can be split into 2 parts:

1. At least one prime graph is accepted by the algorithm for each isomorphism class of prime graphs.

2. At most one prime graph is accepted by the algorithm for each isomorphism class of prime graphs.

Both parts are proven by induction on the number of vertices n of a prime graph.

Proof of part 1. Assume that all prime graphs with less than n vertices ($n > 4$) are generated at least once by the algorithm.

Suppose that no graph from the isomorphism class of a prime graph G with n vertices was generated and accepted by the algorithm. It follows from Lemma 2.2 that the canonical reduction reduces G to a smaller prime graph. The following cases can occur:

1. G contains at least one reducible lollipop.

   So the canonical reduction of G is a lollipop reduction. We reduce G by reducing a canonical reducible lollipop and call the reduced graph $p(G)$. The reducible lollipop l of G which was reduced is mapped to an edge e in $p(G)$. It follows from Lemma 2.2 that $p(G)$ is prime.

   By induction a graph H from the same isomorphism class as $p(G)$ was generated by the algorithm. Let γ be an isomorphism from $p(G)$ to H.

   The algorithm applies the lollipop insertion operation to one edge of each equivalence class of edges of H. So if the lollipop operation was not applied to $\gamma(e)$, it was applied to some other edge e' from the same orbit of edges of H under the action of $\mathrm{Aut}(H)$. Let l' be the lollipop which was obtained by applying the expansion to e'. This yields an expanded graph G' which is isomorphic to G, so there is an isomorphism γ^* from G to G'. l' is in the same orbit as $\gamma^*(l)$ of $\mathrm{Aut}(G')$, so l' is a canonical reducible lollipop. Thus G' is accepted by the algorithm $\lightning$.

2. G contains no reducible lollipops, but contains at least one reducible non-adjacent edge diamond.

   The proof is completely analogous to the proof of 1.

3. G contains no reducible lollipops and no reducible non-adjacent edge diamonds, but at least one reducible edge diamond.

   The proof is completely analogous to the proof of 1.

Proof of part 2. Assume that all prime graphs with less than n vertices ($n > 4$) are generated at most once by the algorithm.

Suppose that the algorithm generated two isomorphic prime graphs G and G' with n vertices which are both accepted by the algorithm. The following cases can occur:

1. G contains at least one reducible lollipop.

   In this case G and G' were constructed by a lollipop insertion operation. So we apply the canonical reduction to G and G' and call the reduced graphs $p(G)$ and $p(G')$ respectively. It follows from Lemma 2.2 that $p(G)$ and $p(G')$ are prime. Suppose the lollipop l was reduced in G and the lollipop l' in G' and let γ be an isomorphism from G to G'. Since l and l' are both canonical reducible lollipops, $\gamma(l)$ is in the same orbit as l' under the action of $\mathrm{Aut}(G')$, so $p(G) \cong p(G')$. By induction $p(G) = p(G')$.

   Suppose that G was constructed from $p(G)$ by applying the lollipop insertion operation to the edge e of $p(G)$ and that G' was obtained by applying the lollipop insertion operation to edge e'. But restricting the isomorphism γ (that maps l to l') to the vertices that already belonged to $p(G)$ gives an automorphism of $p(G)$ mapping e to e', showing that they were in the same orbit of $\mathrm{Aut}(p(G))$, contrary to our procedure.

2. G contains no reducible lollipops, but contains at least one reducible non-adjacent edge diamond.

   The proof is completely analogous to the proof of 1.

3. G contains no reducible lollipops and no reducible non-adjacent edge diamonds, but at least one reducible edge diamond.

   The proof is completely analogous to the proof of 1.

□

2.3.3 Determining possible expansions

Recall from Section 2.3.2 that a graph is only accepted if it was constructed by a canonical expansion. The class of prime graphs is not closed under the construction operations for prime graphs, but every prime graph can be constructed from a smaller prime graph by a canonical expansion (see Lemma 2.2). So if a canonical expansion yields a non-prime graph, we do not accept the expanded graph.

We call an edge where both endpoints are part of the same K_4^- a *diamond edge*. Note that if a lollipop or edge diamond insertion operation is applied to a diamond edge, the resulting graph will not be prime (unless the parent graph was K_4). If one of the edges of the pair of edges to which a non-adjacent edge diamond insertion operation is applied is a diamond edge, the resulting graph will also not be prime.

In the remainder of this section we describe for each of the three construction operations which expansions can yield an expanded graph which is accepted by the algorithm. Expansions which can clearly not be canonical are not performed by the algorithm.

Lollipop insertion operation

The lollipop insertion operation is applied to edges. Since this operation has the highest priority, it can be applied to each edge of a given prime graph as long as the expanded graph is also prime.

Applying the lollipop insertion operation to an edge e of a prime graph with more than 4 vertices yields a prime graph if and only if e is a bridge or both endpoints of e are part of a different K_4^-.

Non-adjacent edge diamond insertion operation

The non-adjacent edge operation has a lower priority than the lollipop operation. Thus if the graph obtained after a non-adjacent edge diamond insertion contains reducible lollipops, it is not accepted. So the non-adjacent edge diamond insertion operation has to destroy all existing reducible lollipops without generating any new ones. Figure 2.7 gives an example of how a non-adjacent edge diamond insertion can yield a new reducible lollipop. Next to that the expanded graph still has to be prime.

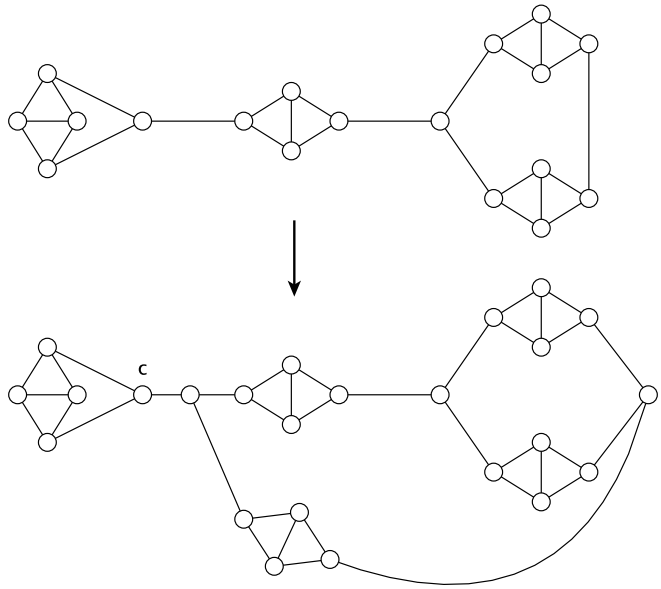


Figure 2.7: A non-adjacent edge diamond insertion which yields a new reducible lollipop with central vertex c.

The non-adjacent edge diamond insertion operation is applied to pairs of non-adjacent edges. Suppose that we have a prime graph G, then the following cases can occur:

- G contains at least 3 reducible lollipops.

  The non-adjacent edge diamond insertion operation can destroy at most 2 reducible lollipops. So the list of non-adjacent edges which are eligible for expansion is empty.

- G contains contains 2 reducible lollipops.

  The lollipops can only be destroyed if both endpoints of the first edge in the pair of edges to which the expansion is applied are in the K_4^+ of the first reducible lollipop and the endpoints of the second edge of the edge pair are in the K_4^+ of the second reducible lollipop. However applying the non-adjacent edge diamond operation to such a pair of edges yields a non-prime graph. So also in this case the list of non-adjacent edges which are eligible for expansion is empty.

- G contains 1 reducible lollipop.

  The reducible lollipop can only be destroyed if both endpoints of an edge in the pair of edges to which the non-adjacent edge diamond expansion is applied are in the K_4^+ of the reducible lollipop. However applying an expansion to such a pair of edges yields a non-prime graph. So also here the list of non-adjacent edges which are eligible for expansion is empty.

- G contains no lollipops.

  Here we only have to make sure we do not generate any new reducible lollipops and that the expanded graph is still prime. The pairs of edges which can be chosen for expansion consist of all pairs of non-adjacent edges where none of the edges is a diamond edge.

When an expansion is applied we still have to check if no new reducible lollipops are generated, because the non-adjacent diamond edge insertion operation can turn a K_4^- into a reducible lollipop (cf. Figure 2.7). We also have to test if the expanded graphs are prime.

For the efficiency of the algorithm it is not a problem that various graphs might be rejected by the generation algorithm for prime graphs. Since, as is shown in Section 2.3.4, the generation of prime graphs is certainly not the bottleneck in the generation algorithm for all cubic graphs.

Edge diamond insertion operation

Since the edge diamond operation has the lowest priority among the three operations for prime graphs, we have to make sure that this insertion operation destroys all existing reducible lollipops and reducible non-adjacent edge diamonds without generating any new ones. Next to that, we also have to make sure that the expanded graphs are still prime.

The edge diamond insertion operation is applied to edges. Suppose that we have a prime graph G, the following cases can occur:

- G contains 2 reducible lollipops and no reducible non-adjacent edge diamonds.

  It is only possible to destroy these 2 reducible lollipops if the 2 central vertices of those lollipops are neighbours. This is only the case in the graph of Figure 2.8. Here the only edge which is eligible for expansion is the edge connecting the 2 central vertices.

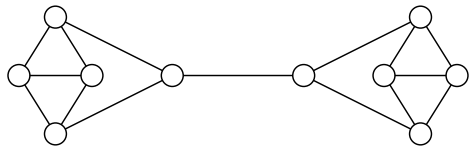


Figure 2.8: A prime graph which contains 2 lollipops.

- G contains 1 reducible lollipop and no reducible non-adjacent edge diamonds.

  Here there are exactly 3 edges which can be chosen for expansion: the 3 edges which are incident to the central vertex of the reducible lollipop. If the expansion is applied to other edges of the lollipop, the expanded graph will not be prime.

- G contains reducible 1 non-adjacent edge diamond and no reducible lollipops.

  The only way to destroy the reducible non-adjacent edge diamond without generating a new one (and where the expanded graph is still prime) is to apply the expansion to the edges where one vertex is an extremal vertex of the K_4^- of the reducible non-adjacent edge diamond and where the other vertex of the edge is the neighbour of that extremal vertex which is not part of the K_4^-. So there are exactly 2 edges which can be chosen for expansion.

- G contains no reducible lollipops and no reducible non-adjacent edge diamonds.

  In this case all edges which are no diamond edges are eligible for expansion.

- In any other case the list of edges which are eligible for expansion is empty: since the edge diamond insertion operation only modifies one edge of the original graph, it cannot destroy more than 1 reducible lollipop or reducible non-adjacent edge diamond (except for the graph from Figure 2.8).

The edge diamond insertion operation only modifies one edge in the original graph and the new edges of the expanded graph are irreducible since they share a vertex with a K_4^-. Since we do not apply this operation to diamond edges (unless the parent graph is K_4), the expanded graph will be prime if the original graph was. So for the edge diamond operation we do not have to check if the expanded graphs are prime.

2.3.4 Conclusion

We could apply additional optimisations to further speed up the generation of prime graphs, but as Table 2.1 shows, the number of prime graphs is negligible compared to the number of cubic graphs. So these optimisations would make no difference in the total generation time for the algorithm to construct all cubic graphs. Thus the generation algorithm for prime graphs is certainly not a bottleneck.

Though it is unimportant in practice, we can prove the asymptotic rarity of prime graphs: A cubic graph with $2n$ vertices has $3n$ edges. Each construction operation for prime graphs adds a new K_4^- and increases the number of vertices by at most 6. So by induction using Lemma 2.2, each prime graph of order $2n$ for $n \geq 3$ has at least $\frac{2n}{6} = \frac{n}{3}$ copies of K_4^-.

By simultaneously removing the central edge in all copies of K_4^-, we obtain a cubic multigraph (i.e. parallel edges are allowed) with at most than $2n - \frac{2n}{3} = \frac{4n}{3}$ vertices. This shows that the number of prime graphs of order $2n$ is at most equal to the number of connected cubic multigraphs of order at most $\frac{4n}{3}$. It was shown by Read [104] that the numbers of labelled cubic graphs or labelled cubic multigraphs on $2k$ vertices is, in each case, asymptotically

$$\frac{\Theta(1)(6k)!}{288^k(3k)!}$$

We can divide by $(2k)!$ to obtain the asymptotic counts of unlabelled graphs, since in both classes most graphs have trivial automorphism groups [93]. Let $F(2k)$ be $\frac{\Theta(1)(6k)!}{288^k(3k)!(2k)!}$. By Stirling's formula $F(2k) \approx k^k e^{\mathcal{O}(k)}$. This gives us:

$$\begin{aligned} \frac{F(2/3 \cdot 2n)}{F(2n)} &\approx \frac{(\frac{2n}{3})^{\frac{2n}{3}}}{n^n} \\ &\approx n^{(\frac{2}{3}-1)n} \end{aligned}$$

This shows that the fraction of cubic graphs which are prime is $n^{-\Omega(n)}$.

$\|V(G)\|$	# prime graphs	# cubic graphs
4	1	1
6	0	2
8	1	5
10	1	19
12	1	85
14	3	509
16	2	4 060
18	5	41 301
20	4	510 489
22	9	7 319 447
24	11	117 940 535
26	16	2 094 480 864
28	32	40 497 138 011
30	37	845 480 228 069
32	73	18 941 522 184 590

Table 2.1: Number of prime graphs vs. number of cubic graphs.

2.4 Generation of graphs with reducible triangles

2.4.1 Introduction

As can be seen from Table 2.1, the number of prime graphs is very small compared to the number of cubic graphs. So the efficiency of the algorithm is entirely determined by the efficiency of the algorithm to generate non-prime graphs. These graphs are constructed by the edge insertion operation from Figure 2.9.

The class of connected cubic graphs is closed under the edge insertion operation, so the only time consuming routines are those that make sure that only pairwise non-isomorphic graphs are generated. As with the generation for prime graphs, we also use the canonical construction path method for this.

When using the canonical construction path method, we first have to define a canonical reduction which is unique up to isomorphism for every non-prime graph. The inverse of a canonical reduction is a *canonical expansion*. Recall from Section 1.3 that the two rules of this method are:

1. Only accept a graph if it was constructed by a canonical expansion.

2. For every graph G to which construction operations are applied, only perform one expansion from each equivalence class of expansions of G.

The isomorphism rejection for non-prime graphs is described in more detail in Sections 2.4.2 and 2.5.1.

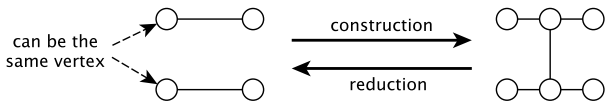


Figure 2.9: The basic edge insertion operation.

The theory of random cubic graphs (see [122] for example) says that the number of triangles is asymptotically a Poisson distribution with mean $\frac{4}{3}$. So asymptotically the probability that a cubic graph contains no triangles is $e^{-\frac{4}{3}} = 0.264...$ The expected number of copies of K_4^- is $o(1)$, so this implies that asymptotically about 74% of cubic graphs have triangles whose edges are reducible (that are triangles which are not contained in a K_4^-). This percentage is also approximately true for the graphs of the small sizes we are dealing with, which justifies paying special attention to these *reducible triangles* as they allow some optimisations that do not work in general. Table 2.2 shows statistics about the number of reducible triangles in cubic graphs with 26 vertices.

# reducible triangles	# graphs
0	497 010 000
1	774 885 044
2	540 977 972
3	218 274 256
4	54 459 966
5	8 183 373
6	666 137
7	23 837
8	279

Table 2.2: Counts of all 2 094 480 864 connected cubic graphs with 26 vertices according to the number of reducible triangles they have.

2.4.2 Construction

Note that the operation of edge insertion applied to two vertices having a common endpoint can be seen as that of replacing the common endpoint by a triangle. The result depends only on what the common endpoint was and not which two edges were used. We call this *triangle insertion*. Similarly, the reverse operation of *triangle reduction* can be seen as that of replacing a reducible triangle by a vertex. This operation is also shown in Figure 2.10.

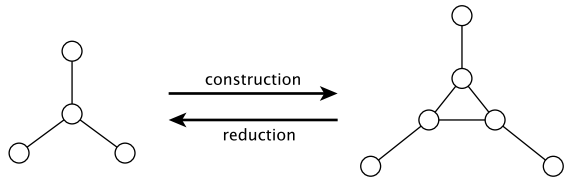


Figure 2.10: The basic triangle operation.

We give triangle reductions priority over other edge reductions. So the canonical reduction of a graph with reducible triangles is a triangle reduction. This allows us to *bundle* triangle operations. The idea is to reduce each reducible triangle at the same time, but there is a twist in that the two triangles in the subgraph $ext(K_4^-)$ shown in Figure 2.11 cannot be reduced at once, since parallel edges are not allowed. However reduction of either of the two triangles in an $ext(K_4^-)$ produces the same smaller graph, so we define our bundled triangle reduction as simultaneously reducing one triangle from each $ext(K_4^-)$ and every other reducible triangle.

Thus a graph with reducible triangles has only one reduction (up to isomorphism). So this is the canonical reduction and it provides (up to isomorphism) a unique ancestor for each connected cubic graph that has reducible triangles.

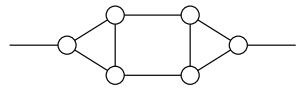


Figure 2.11: A subgraph $ext(K_4^-)$ with two reducible triangles that cannot be reduced at once.

We next identify the inverse operation of a bundled triangle reduction. For a graph G call a set $S \subseteq V(G)$ *extensible* if at least one vertex of every reducible

triangle of G is contained in S. Then a *bundled triangle insertion* is to insert a triangle at each of the vertices in S (note that it is also possible that $|S| = 1$). After a bundled triangle insertion, all the $ext(K_4^-)$ subgraphs and other reducible triangles have been created in this last operation, so every bundled triangle insertion is canonical.

Applying a bundled triangle insertion to distinct extensible sets of a graph G might yield isomorphic graphs. To avoid this, we define an equivalence relation on the extensible sets and will apply bundled triangle insertion to only one extensible set in every equivalence class (i.e. the second rule of the canonical construction path method).

The equivalence relation "$\equiv$" on extensible sets is generated by the following two equivalences:

(a) If there is an automorphism γ of G with $\gamma(S) = S'$, then $S \equiv S'$.

(b) If $|S| = |S'|$ and $(S \setminus S') \cup (S' \setminus S)$ is the set of extremal vertices of a K_4^- in G so that each of S, S' contains exactly one vertex in this K_4^-, then $S \equiv S'$.

Lemma 2.4. *Consider a graph G and two extensible sets S, S' of G. Let $T(G, S)$, respectively $T(G, S')$, denote the graphs obtained by applying bundled triangle insertion to S, respectively S'. Then $T(G, S)$ and $T(G, S')$ are isomorphic if and only if $S \equiv S'$.*

Proof.
Part 1: $\Longleftarrow$

If $S \equiv S'$, there is a set of generating relations $S \equiv S_1$, $S_1 \equiv S_2$, $S_2 \equiv S_3$, ..., $S_n \equiv S'$. If $S_i \equiv S_{i+1}$ is an equivalence of type (a), $T(G, S_i)$ and $T(G, S_{i+1})$ are isomorphic. If $S_i \equiv S_{i+1}$ is an equivalence of type (b), $T(G, S_i)$ and $T(G, S_{i+1})$ are also isomorphic. So we have $T(G, S) \cong T(G, S_1)$, $T(G, S_1) \cong T(G, S_2)$, ..., $T(G, S_n) \cong T(G, S')$. Since $\cong$ is transitive we have $T(G, S) \cong T(G, S')$.

Part 2: $\Longrightarrow$

Suppose that γ is an isomorphism from $T(G, S)$ to $T(G, S')$. Note that γ must map the set of subgraphs $ext(K_4^-)$ of $T(G, S)$ onto the set of subgraphs $ext(K_4^-)$ of $T(G, S')$, and also map the other reducible triangles of $T(G, S)$ onto the other reducible triangles of $T(G, S')$. Consider the mapping ϕ from $V(T(G, S))$ to $V(G)$ defined by contracting the central two edges of each $ext(K_4^-)$ (drawn horizontally in Figure 2.11) and contracting all the edges of each reducible triangle that is not in an $ext(K_4^-)$ and relabelling the vertices of the contracted graph as in G. Define

the mapping ϕ' from $V(T(G,S'))$ to $V(G)$ similarly. The permutation γ_0 of $V(G)$ defined by $\gamma_0(\phi(v)) = \phi'(\gamma(v))$ for all $v \in V(T(G,S))$ is an automorphism of G. Moreover, γ_0 maps S onto S' with the possible exception that when S contains one extremal vertex w, and no other vertex, of a K_4^- subgraph H, S' might contain either $\gamma_0(w)$ or the other extremal vertex of the K_4^- subgraph $\gamma_0(H)$. In either case, $S \equiv S'$ by the definition of equivalence. □

Since for every graph with reducible triangles the graph resulting from a bundled triangle reduction is uniquely determined (i.e. every graph obtained by a bundled triangle insertion operation is canonical), we get the following Lemma:

Lemma 2.5. *If exactly one representative of every isomorphism class of cubic connected graphs up to $n-2$ vertices is given, then applying bundled triangle insertion to one member of each equivalence class of extensible sets that leads to a cubic connected graph on n vertices generates exactly one representative for every isomorphism class of cubic connected graphs on n vertices that contain reducible triangles.*

Thus no isomorphism rejection is needed for graphs with reducible triangles (other than the equivalence relation on extensible sets): graphs constructed by the bundled triangle operation are always canonical and pairwise non-isomorphic. This is one of the main advantages of the triangle insertion bundling, as checking whether the last operation was canonical can be very expensive in general. This does not only avoid the time for canonicity checking, but also the time needed for constructing graphs that are rejected after construction because they are not canonical. Also note that graphs with r reducible triangles and n vertices are generated from graphs with $n-2r$ vertices. So as can be seen from Table 2.2 at the beginning of this section, graphs with reducible triangles are often generated from much smaller graphs.

2.4.3 Optimisations

One of the routines that can be time consuming in structure generation programs is the computation of automorphism groups. We used *nauty* for this task. Although *nauty* is very efficient, the large number of calls can be expensive.

The argument used in the proof of Lemma 2.4 shows that a non-trivial automorphism of $T(G,S)$ must come from a non-trivial automorphism of G. So if $\mathrm{Aut}(G)$ is trivial, this implies that $\mathrm{Aut}(T(G,S))$ is also trivial. Thus if $\mathrm{Aut}(G)$ is trivial, we do not have to call *nauty* to compute $\mathrm{Aut}(T(G,S))$. This is another

big advantage of triangle insertion bundling. E.g. for 26 vertices about 78% of the graphs have a trivial group and the ratio is increasing.

Even if $\mathrm{Aut}(G)$ is non-trivial, the orbits of $\mathrm{Aut}(G)$ can be used to define vertex colours (see [88]) for $T(G,S)$ to speed up the computation of $\mathrm{Aut}(T(G,S))$. More specifically it is possible to give a partition or colouring of the set of vertices when calling *nauty*. Vertices which are in a different partition (i.e. have a different colour), are assumed to be in a different orbit of the automorphism group of the graph. An example of a simple and straightforward partitioning is to partition the vertices according to their degree. But since we are dealing with cubic graphs, this partitioning is not useful.

Since a non-trivial automorphism of $T(G,S)$ must come from a non-trivial automorphism of G, two vertices in G that are not in the same orbit of $\mathrm{Aut}(G)$ will also be in different orbits of $\mathrm{Aut}(T(G,S))$. If $v, w \in V(G)$ are in the same orbit of $\mathrm{Aut}(G)$ and v is in S and w is not, w and the vertices by which v was replaced will be in different orbits of $\mathrm{Aut}(T(G,S))$.

We illustrate these vertex colours in Figure 2.12, where the colours in the left graph denote vertices which are in the same orbit of the automorphism group of the graph and where the colours in the right graph denote vertices which may be in the same orbit. The vertices in the left graph with a square around them are vertices which are in an extensible set which will be blown up by the bundled triangle insertion operation. All this information can be used by *nauty* to speed up the computation of the automorphism group of the expanded graph. Note that we assign the same vertex colour to all 3 vertices of a reducible triangle in the expanded graph. In many cases we already know that these 3 vertices will not be in the same orbit of the automorphism group of the expanded graph, but we do not assign a different vertex colour to them, since *nauty* easily detects this.

Consider the mapping ϕ from $V(T(G,S))$ to $V(G)$ defined by contracting the central two edges of each $ext(K_4^-)$ (drawn horizontally in Figure 2.11) and contracting all the edges of each reducible triangle that is not in an $ext(K_4^-)$ and relabelling the vertices of the contracted graph as in G. More precisely, if G has a non-trivial automorphism group, then two vertices $v, w \in V(T(G,S))$ have the same vertex colour if and only if:

- v and w are part of an $ext(K_4^-)$.
- v and w are part of a reducible triangle but not of a $ext(K_4^-)$ and there is an automorphism of G mapping $\phi(v)$ to $\phi(w)$.
- v nor w are part of a reducible reducible triangle and there is an automor-

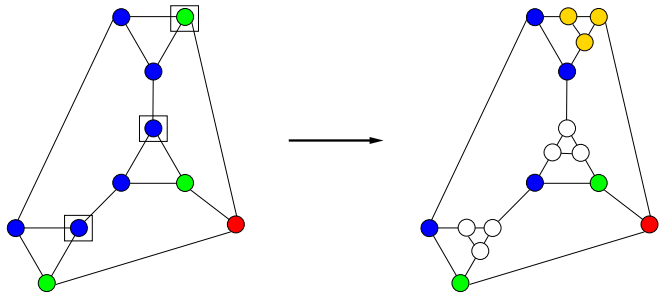


Figure 2.12: Reusing information about the orbits of the automorphism group of the parent graph to speed up the computation of the automorphism group of the expanded graph.

phism of G mapping $\phi(v)$ to $\phi(w)$.

If all vertices which are part of an extensible set S of a graph G are in a different orbit of $\mathrm{Aut}(G)$, we can sometimes apply another optimisation. Let $|\Gamma(v)|$ be the size of the orbit of a $v \in V(G)$ of $\mathrm{Aut}(G)$. If for an automorphism $\gamma \in \mathrm{Aut}(G)$ and a $v \in S$ it holds that $\gamma(v) \neq v$, then this automorphism cannot lead to an automorphism of $T(G, S)$. If $\exists\, v \in S : |\Gamma(v)| = |\,\mathrm{Aut}(G)|$, this means that $\forall \gamma \in Aut(G) : \gamma(v) \neq v$. So then the automorphism group of $T(G, S)$ will be trivial, thus we also do not have to call *nauty* to compute it. Though in practice this optimisation only yields a tiny speedup.

2.5 Generation of non-prime graphs without reducible triangles

2.5.1 Isomorphism rejection

Recall that we gave the triangle operation priority over the edge operation. So in principle the non-prime connected cubic graphs without reducible triangles on n vertices are generated by applying the edge insertion operation to each pair of edges in a graph on $n-2$ vertices that guarantees that no reducible triangles are present in the resulting graph. We call such a pair an *extensible* pair of edges. In Section 2.5.2 we go into more detail about determining which pairs of edges are extensible.

Applying the edge insertion operation to all extensible pairs of edges may lead to the construction of isomorphic copies. Therefore we will now describe how we can make sure that only pairwise non-isomorphic graphs are output. We also use the canonical construction path method for this.

The first task is to define, for each non-prime graph G without reducible triangles, a *canonical edge reduction* which is unique up to isomorphism. The result of performing the canonical edge reduction will be a graph G', uniquely determined by G up to isomorphism, from which G can be made by edge insertion. As is usual with the canonical construction path method (cf. rule 1), we will only accept G if it is made from a graph isomorphic to G' by the inverse of the canonical reduction (i.e. by a canonical expansion); otherwise we will reject it.

To define an efficient canonical edge reduction, we assign a 7-tuple $(x_0, \ldots, x_6)$ to every reducible edge e and choose a reducible edge with the largest 7-tuple. The values of $x_0, \ldots, x_4$ are combinatorial invariants of increasing discriminating power and cost. More specifically they are defined as follows:

1. $x_0 = 1$ if e is part of a 4-gon and 0 otherwise.
2. x_1 is the negative of the number of vertices at distance at most 2 from e.
3. x_2 is the number of 4-gons containing e.
4. x_3 is the negative of the number of vertices at distance at most 3 from e.
5. x_4 is the sum of x_1 and x_3 of all 4 edges incident with e.

We call the edge which was inserted by the last edge insertion operation the *inserted edge*. Each x_i is only computed if the previous values fail to determine a unique reducible edge and the inserted edge is still potentially one of those with the largest 7-tuple. So we first compute x_0 for all reducible edges. Only for the edges for which x_0 is maximal, the value of x_1 is computed. Then for the edges with maximal (x_0, x_1), x_2 is computed, etc. As soon as the inserted edge is not maximal or if it is the only maximal edge, we can stop. If the inserted edge is not maximal, we know the graph is not canonical. If it is the only maximal edge, we know it is canonical.

In case there is more than one reducible edge with the largest value of $(x_0, \ldots, x_4)$ (and the inserted edge is among them), we canonically label the graph using *nauty*, and define $x_5 > x_6$ such that $\{x_5, x_6\}$ is the lexicographically largest canonical labelling of an edge in the same orbit of $\mathrm{Aut}(G)$ as e.

The discriminating power of $x_0, \ldots, x_4$ is sufficient to avoid the more expensive computation of x_5, x_6 in 92% of the cases for $n = 26$. This fraction is increasing for larger values of n.

The values $x_0, \ldots, x_4$ are invariant under isomorphisms, so edges that are equivalent under the automorphism group have the same values. The values x_5, x_6 have an even stronger property: two edges are equivalent under the automorphism group if and only if x_5, x_6 are the same. So two edges have the same tuple $(x_0, \ldots, x_6)$ if and only if they are in the same orbit of the automorphism group of the graph. Together with the definition of canonical labelling, this implies the following:

Lemma 2.6. *Let G_1 and G_2 be connected cubic graphs with reducible edges, and let γ be an isomorphism from G_1 to G_2. Let e_1 and e_2 be, respectively, reducible edges of G_1 and G_2 having largest 7-tuples. Then $\gamma(e_1)$ is in the same orbit as e_2. Furthermore, the graph $\bar{G}_1$ obtained by reducing e_1 in G_1 is isomorphic to the graph $\bar{G}_2$ obtained by reducing e_2 in G_2.*

Next to that, there is an isomorphism from $\bar{G}_1$ to $\bar{G}_2$ mapping the edge pairs to which e_1 and e_2 were reduced onto each other: Let p_1 be the edge pair of $\bar{G}_1$ obtained by applying the edge reduction to e_1 in G_1 and let p_2 be the edge pair of $\bar{G}_2$ obtained by applying the edge reduction to e_2 in G_2. Restricting the isomorphism from G_1 to G_2 that maps e_1 to e_2 to the vertices that already belonged to $\bar{G}_1$ and $\bar{G}_2$, respectively, yields an isomorphism from $\bar{G}_1$ to $\bar{G}_2$ mapping p_1 to p_2.

The algorithm would work correctly if only x_5, x_6 were computed, but $x_0, \ldots, x_4$ are important for the efficiency of the algorithm. While for x_5, x_6 a canonical form must be computed, the earlier values are based on purely local criteria that are cheaper to compute. Furthermore these criteria allow some look-ahead. For example, when expanding a graph G by inserting a new edge, it is easy to decide already on the level of G whether x_0 will be 1 or 0 for this edge in the expanded graph G' and whether other edges with $x_0 = 1$ in G' will exist. This allows us to avoid the construction of a lot of children that would afterwards be rejected because the last edge inserted does not have maximal value of $(x_0, \ldots, x_6)$. We go into more detail about this in Section 2.5.2.

What we still have to make sure is that from a graph with $n - 2$ vertices we never construct two isomorphic graphs that are both accepted. Therefore, when applying the edge insertion operation to a graph G, we first determine its automorphism group $\mathrm{Aut}(G)$. After constructing the set of all extensible edge

pairs (see Section 2.5.2 for details), we compute the orbits of $\mathrm{Aut}(G)$ on the pairs and apply the edge insertion operation to exactly one pair in each orbit (implementing the second rule of the canonical construction path method). This suffices to prevent isomorphic graphs from being accepted, as the following lemma shows. The fact that all graphs are still constructed can be seen by observing that applying the operation to edge pairs in the same orbit gives isomorphic graphs.

Lemma 2.7. *Assume that exactly one representative of each isomorphism class of connected cubic graphs with $n-2$ vertices is given. Suppose we perform the following steps (cf. Algorithm 1.1 from Section 1.3):*

1. *Apply the edge insertion operation to one edge pair in each orbit of extensible edge pairs.*
2. *Accept each new non-prime graph with n vertices and without reducible triangles if and only if the last edge inserted has maximum value of $(x_0, \ldots, x_6)$.*

Then exactly one representative of each isomorphism class of connected cubic non-prime graphs with n vertices and without reducible triangles is accepted.

Proof. From Lemma 2.6 we know that isomorphic graphs must be made from the same parent. So assume that G is expanded by applying edge insertion to edge pairs $p_1 = \{e_1, e_1'\}$ and $p_2 = \{e_2, e_2'\}$ in G and that the results are two isomorphic graphs G_1, G_2 that are both accepted. The inserted edges $\bar{e}_1$ and $\bar{e}_2$ must both have maximum $(x_0, \ldots, x_6)$ (else G_1 or G_2 will not be accepted), so by Lemma 2.6 there is an isomorphism from G_1 to G_2 that maps $\bar{e}_1$ to $\bar{e}_2$. But then restricting the isomorphism to the vertices that already belonged to G gives an automorphism of G mapping p_1 to p_2, showing that they were in the same orbit of the automorphism group of G, contrary to our procedure. □

Together, Lemma 2.5 and Lemma 2.7 give the following theorem:

Theorem 2.8. *If the algorithm which was just described is recursively applied to all prime graphs up to n vertices, exactly one representative of every isomorphism class of cubic connected graphs on n vertices is constructed.*

2.5.2 Determining possible expansions

Since we give the (bundled) triangle operation priority over the edge insertion operation, pairs of edges which are eligible for edge insertion have to destroy all reducible triangles without generating new ones when the edge insertion operation

is applied to them. Next to that, the inserted edge must have maximal value of $(x_0, \ldots, x_6)$. This allows for some look-ahead. We will now describe which edge insertion expansions can lead to a canonical graph.

Suppose that we have a cubic graph G for which we want to determine the possible edge insertions. The following cases can occur:

- G contains at least 3 reducible triangles.

  The edge insertion operation can destroy at most 2 reducible triangles, so if the graph contains more than 2 reducible triangles, the list of edge pairs which are eligible for edge insertion is empty.

- G contains 2 reducible triangles.

  The only pairs of edges which destroy both triangles after the edge insertion operation has been applied to them, are the pairs of edges where one edge is part of the first reducible triangle and the other edge is part of the second triangle.

  So there are 9 edgepairs which could be chosen for expansion. However we also have to take canonicity requirements (see Section 2.5.1) into account. Since the expanded graph will contain reducible edges which are part of a 4-gon (namely at least the edges of the 2 reducible triangles which were destroyed by the edge insertion operation), the inserted edge also has to be part of a 4-gon. If this is not the case its value of x_0 will be zero, while there are reducible edges with $x_0 = 1$. So the 2 reducible triangles have to be connected by an edge.

  The inserted edge not only has to be part of a square in the expanded graph, the edges of the edge pair in the original graph also have to be part of the same square or pentagon. If this is not the case, the expanded graph will have another reducible edge which has the same values for (x_0, x_1) as the inserted edge, but with a larger value of x_2 (i.e. the number of 4-gons where the edge is part of). This is depicted in Figure 2.13: the inserted edge e and the reducible edge e' are both part of a square and both have 10 vertices at distance at most 2 (i.e. their value of (x_0, x_1) is $(1, -10)$. But e' is part of 2 squares, while e is only part of 1 square, so e cannot be a canonical edge. However if v and w are the same vertex (i.e. the edge pair to which the edge insertion is applied is part of a pentagon), the value of e for (x_0, x_1) is $(1, -9)$ while e' still has value $(1, -10)$.

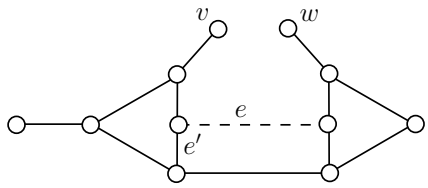


Figure 2.13: Subgraph of a cubic graph. The inserted edge e has value $(1, -10, 1)$ for (x_0, x_1, x_2) while e' has value $(1, -10, 2)$.

Thus each eligible pair of edges must contain one edge of each reducible triangle and the edges of the edge pair must be part of the same square or pentagon. This is quite restrictive, so often the list of edge pairs which are eligible for edge insertion will be empty if G contains 2 reducible triangles.

- G contains 1 reducible triangle.

  Each pair of non-adjacent edges which contains an edge from the reducible triangle will destroy the triangle. Similar to the case where G contains 2 reducible triangles, the inserted edge must be part of a square, otherwise there is a reducible edge with a larger value of x_0. Similarly, the inserted edge must also be part of at least 2 squares, otherwise there is a reducible edge which has the same value of (x_0, x_1), but which is part of more squares (i.e. which has a larger value of x_2) (cf. Figure 2.13).

  So an edge pair $\{e, e'\}$ is eligible for expansion if e is part of the reducible triangle and there are two edges between endpoints of e and e'.

- The graph contains no reducible triangles.

  In this case all pairs of non-adjacent edges are eligible for expansion.

We also have to make sure that no new reducible triangles are generated by the expansion. The edge insertion operation only generates new reducible triangles if the edges of the edge pair are adjacent or if one edge of the edge pair is part of a K_4^- and contains an extremal vertex, while the other non-adjacent edge of the edge pair is not part of the same K_4^-. So these pairs of edges are also not eligible for edge insertion.

2.5.3 Optimisations

Further optimisations speeded up the algorithm significantly. The first two optimisations avoid computing $(x_0, \ldots, x_6)$ for a lot of edges and the third optimisation reduces the time needed by *nauty*.

Look-ahead for maximality of $(x_0, \ldots, x_6)$

In order to be canonical, the inserted edge not only has to destroy all reducible triangles of the parent graph without generating any new ones. It also must have the largest value of $(x_0, \ldots, x_6)$ among all reducible edges of the expanded graph.

Although $x_0, \ldots, x_4$ are a lot cheaper to compute than x_5, x_6, the computation of $x_0, \ldots, x_4$ still consumes a large amount of CPU-time compared to the rest of the algorithm, especially if $x_0, \ldots, x_4$ have to be computed for a lot of reducible edges. Therefore it would be useful to have a look-ahead for the maximality of $(x_0, \ldots, x_6)$.

In Section 2.5.2 we already introduced some simple look-aheads using x_0.

We also use a look-ahead for x_1: suppose we want to apply the edge insertion operation to a pair of edges e_1, e_2 of a graph G. The minimal value of x_1 (i.e. the negative of the number of vertices at distance at most 2 from a given edge) is -14. The (vertex) neighbourhood $N(e)$ of an edge $e = \{v, w\}$ is defined as the union of the neighbourhoods of v and w. Then the value of x_1 of the inserted edge in the expanded graph is $-(14 - |N(e_1) \cap N(e_2)|)$.

Note that the value of (x_0, x_1) for an edge $\{v, w\}$ is not modified by applying the edge insertion operation to e_1, e_2 if $(N(e_1) \cup N(e_2)) \cap \{v, w\} = \emptyset$.

For every graph we store the reducible edges which have maximal value of (x_0, x_1). We call these *maximal* edges. Suppose that these edges have value (a, b) for (x_0, x_1) and that the inserted edge in the expanded graph will have value (a', b') for (x_0, x_1). If (a, b) is lexicographically larger than (a', b') and there is a maximal edge whose value of (x_0, x_1) will not be modified by applying an edge insertion to e_1, e_2, we do not have to perform this edge insertion since the expansion won't be canonical.

This does not only save the cost of computing (x_0, x_1) for a lot of reducible edges, but it also saves the time needed for performing those non-canonical expansions.

Remembering previous rejector

When an edge is inserted, we compute the value of x_0 for all reducible edges. If the inserted edge has maximal value of x_0, we compute the value of x_1 for all reducible edges which have maximal value of x_0 etc. Whenever the constructed graph is rejected because a reducible edge with a larger value of $(x_0, \ldots, x_4)$ was found, we save the edge which caused the rejection. We call this the *previous rejector*.

Note that the last edge which was inserted by a canonical edge insertion expansion in a graph has the largest value of $(x_0, \ldots, x_6)$ in that graph. We also call this the *max edge* of that graph.

Suppose that we applied an edge insertion operation to a graph and we want to test if the expansion is canonical. Instead of computing the values of (x_0, x_1) of the reducible edges in an arbitrary order and comparing them with the value of (x_0, x_1) of the inserted edge, we first compute the value of (x_0, x_1) of the max edge of the parent graph (at least if that edge is still present in the expanded graph). That edge will mostly still have a large value of (x_0, x_1) in the expanded graph.

If this edge has a larger value of (x_0, x_1) than the inserted edge, we can reject the graph. Otherwise we compute the value of (x_0, x_1) of the *previous rejector* (at least if it is still present and reducible in the current expanded graph). Since the edge insertion operation does not modify the values of (x_0, x_1) of many edges, the previous rejector will often have a larger value of (x_0, x_1) than the inserted edge. Only if this is not the case we compute (x_0, x_1) for the remaining reducible edges.

Often computing the values of (x_0, x_1) is sufficient to be able to reject the constructed graph. By first testing the max edge of the parent graph and the previous rejector, it is often possible to reject the constructed graph without having to compute $x_0, \ldots, x_6$ for many reducible edges. Table 2.3 shows statistics about the percentage of cases where (x_0, x_1) is sufficient to decide whether or not the constructed graph is canonical and the percentage of cases where first testing the max edge and the previous rejector is sufficient to reject the constructed graph.

Using vertex colours

Similar to the bundled triangle operation (see Section 2.4.3), vertex colours can also be used to speed up the computation of the automorphism group of the

number of vertices	percentage (x_0, x_1) sufficient	percentage max edge or prev. rejector sufficient
18	61.4	59.4
20	59.2	56.0
22	56.3	50.9
24	53.4	46.1
26	50.8	42.1

Table 2.3: The percentage of cases where (x_0, x_1) is sufficient to decide whether or not the constructed graph is canonical and the percentage of cases where first testing the max edge and the previous rejector is sufficient to reject the constructed graph.

graphs constructed by edge insertion. More specifically it is possible to give a partition or colouring of the set of vertices when calling *nauty*. Vertices which are in a different partition (i.e. have a different colour), are assumed to be in a different orbit of the automorphism group of the graph.

Recall that if the automorphism group of the parent graph G is trivial, the automorphism group of the graph $T(G, S)$ obtained by bundled triangle insertion to an extensible set of vertices S is trivial as well. This is not the case for the edge insertion operation. The automorphism group of a graph obtained by edge insertion is not necessarily a subgroup of the automorphism group of its parent graph. For example the expanded graph in Figure 2.14 has symmetries which were not induced by the parent graph.

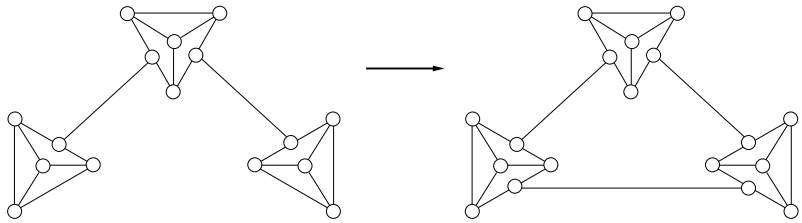


Figure 2.14: Edge insertion which yields symmetries which were not induced by the parent graph.

If we call *nauty* to compute the automorphism group of a graph obtained by edge insertion, it means that the inserted edge is among the reducible edges with

the largest value of $(x_0, \ldots, x_4)$. So the values for x_i $(0 \leq i \leq 4)$ were already computed for a lot of reducible edges. The values $x_0, \ldots, x_4$ are invariant under isomorphisms and can be interpreted as edge colours. Since *nauty* only accepts vertex colours, we translate them to vertex colours. We are using the edge colours x_1 and x_3 for this since they have the highest discriminating power.

Note that x_i was not necessarily computed for every edge. If an edge is irreducible or if the edge did not have maximal value of $(x_0, ..., x_{i-1})$, the value of x_i was not computed. For these edges we set the values of x_i to zero. For all edges which are in the same orbit of edges of the automorphism group of the graph, the value of x_i was either computed or not computed.

We define the colour of a vertex to be the sum of the value of x_3 of its three incident edges and partition them according to their vertex colour. For the partition where all vertices have vertex colour zero, we make subpartitions by using a different vertex colour: the sum of the value of x_1 of the edges which are incident to the vertex. This vertex colouring significantly speeds up *nauty*. We also tried to use the shift operation instead of sum for the vertex colours, but this did not yield an additional speedup. Furthermore we also tested other colours, but they yielded a less significant speedup than the vertex colouring which was described here.

2.6 Generation of graphs with girth at least 4 or 5

The girth of a graph is the length of its smallest cycle. So graphs with girth at least 4 do not contain any triangles and graphs with girth at least 5 contain no triangles or squares. The algorithm was developed and optimised for generation without a girth restriction, but it can be adapted to generate connected cubic graphs with a non-trivial lower bound on the girth quite efficiently. This is done by some simple look-aheads, which are described in the next two sections. Various interesting subclasses of cubic graphs such as snarks (see Chapter 3) have a lower bound on the girth.

2.6.1 Graphs with girth at least 4

Suppose we want to generate all cubic graphs with girth at least 4 and n vertices. Graphs with girth at least 4 do not contain any triangles, so we do not have to

apply the bundled triangle operation if the number of vertices of the graph after applying the operation will be n.

If G has at least 3 reducible triangles, then at least one of them remains when an edge insertion is done. Since we favour triangle reduction over other edge reductions, the expanded graph will not be accepted. Therefore all the descendants of G will be made by triangle insertion and will have triangles. So in a search for cubic graphs with girth at least 4, graphs with more than 2 reducible triangles do not have to be generated.

Recall from Section 2.5.2 that, because of the canonicity requirements, if a graph contains 2 reducible triangles, they can only be destroyed by the edge insertion operation if the edges of the edge pair are part of the same square or pentagon. This means that the only sets of 2 vertices to which the bundled triangle operation should be applied, are those which consist of 2 vertices of a triangle (possibly a triangle contained in a K_4^-).

The edge insertion operation can destroy at most 2 reducible triangles. If the graph contains no reducible triangles, it can destroy the triangles of at most 2 K_4^-'s such that the expanded graph has girth at least 4. If the graph contains 1 reducible triangle, it can destroy at most 1 reducible triangle and the triangles of 1 K_4^-.

This allows us to determine a simple lower bound for the minimal number of vertices which are required to destroy all triangles of a graph G. Let $num_\Delta(G)$ denote the number of reducible triangles in G and $num_{K_4^-}(G)$ the number of K_4^-'s. Then a lower bound $min_{g_4}(G)$ for the minimal number of vertices required to destroy all triangles of G is:

$$min_{g_4}(G) = \left\lceil \frac{num_\Delta(G) + num_{K_4^-}(G)}{2} \right\rceil \cdot 2$$

The "$\cdot 2$" is because the edge insertion operation adds 2 new vertices. So in search of cubic graphs with n vertices and girth at least 4, we do not have to expand graphs G for which $|V(G)| + min_{g_4}(G)$ is larger than n.

2.6.2 Graphs with girth at least 5

Graphs with girth at least 5 do not contain any triangles or squares. When an edge insertion operation destroys a triangle, it becomes a square. So we do not have to apply the bundled triangle operation if the number of vertices of the graph after bundled triangle insertion will be at least $n - 2$.

A lower bound $min_{g_5}(G)$ for the minimal number of vertices required to destroy all triangles and squares of G is given by:

$$min_{g_5}(G) = \left\lceil \frac{num_{\Delta}(G) + num_{K_4^-}(G)}{2} \right\rceil \cdot 4$$

(Here we use the same notation as in Section 2.6.1.) An edge insertion can destroy more than two squares (see Figure 2.15), but if a graph contains triangles, at least $\frac{min_{g_5}(G)}{2}$ edge insertions have to be performed in order to obtain a graph with girth at least 5. So in search of cubic graphs with n vertices and girth at least 5, we do not have to expand graphs G for which $|V(G)| + min_{g_5}(G)$ is larger than n.

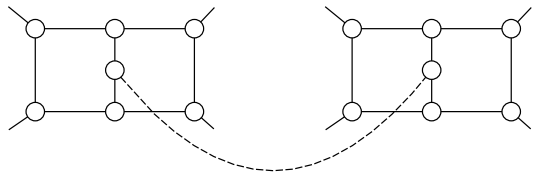


Figure 2.15: Edge insertion which destroys more than two squares.

2.6.3 Conclusion

By simple look-aheads which were described in the previous sections, the generation tree can be substantially pruned so that girth bounds of 4 or 5 can be imposed quite efficiently. Table 2.4 and 2.5 show how many non-isomorphic graphs are generated on each level when using a girth restriction, compared to the number of non-isomorphic graphs which would be generated when no look-aheads would be used (i.e. generating all cubic graphs and applying a filter for the girth). As can be seen from these tables, the look-aheads are most restrictive on the higher levels (i.e. level n and $n-2$). On the lower levels the percentage of graphs generated with girth less than 4 or 5 is much higher, but this is not a problem for the efficiency since there are a lot more graphs on level m than on level $m-2$ (approximately 20 times as many for $m = 30$ and the ratio is increasing for larger values of m). So avoiding the generation of cubic graphs with n or $n-2$ vertices which cannot lead to graphs with n vertices that have girth at least 4 or 5 saves a large amount of work.

$\|V(G)\|$	# cubic graphs	# cubic graphs generated	# cubic graphs with girth at least 4
4	1	1	0
6	2	2	1
8	5	5	2
10	19	14	6
12	85	62	22
14	509	342	110
16	4 060	2 612	792
18	41 301	25 878	7 805
20	510 489	315 940	97 546
22	7 319 447	4 505 972	1 435 720
24	117 940 535	72 430 903	23 780 814
26	2 094 480 864	1 183 209 311	432 757 568
28	40 497 138 011	8 542 471 494	8 542 471 494

Table 2.4: Number of non-isomorphic graphs generated on each level by the algorithm when generating all graphs with 28 vertices and girth at least 4.

$\|V(G)\|$	# cubic graphs	# cubic graphs generated	# cubic graphs with girth at least 5
4	1	1	0
6	2	2	0
8	5	5	0
10	19	14	1
12	85	62	2
14	509	342	9
16	4 060	2 612	49
18	41 301	25 877	455
20	510 489	315 902	5 783
22	7 319 447	4 452 362	90 938
24	117 940 535	71 745 564	1 620 479
26	2 094 480 864	241 172 459	31 478 584
28	40 497 138 011	656 783 890	656 783 890

Table 2.5: Number of non-isomorphic graphs generated on each level by the algorithm when generating all graphs with 28 vertices and girth at least 5.

Our generator is faster than existing generators for all connected cubic graphs. Also for girth bounds of 4 or 5 the program is faster than *minibaum* [11] and *genreg* [95]. More details are given in Section 2.8. It is also faster than the implementations by Sanjmyatav reported in [110], which use operations that remain within the classes defined by the girth bounds (though it remains to be determined whether those algorithms can be optimised).

This pruning technique is likely to become less efficient for larger lower bounds on the girth. For girth close to the maximum possible for a given number of vertices, the fastest method is that of McKay, Myrvold and Nadon [90].

2.7 Generation of graphs with connectivity requirements

The generation algorithm from the previous sections can also be modified to generate cubic graphs with certain connectivity requirements. This is done by a filter and a simple look-ahead.

Connectivity requirements are for example useful when graphs are used to model networks: it is desirable that all nodes (vertices) of the network can still communicate with each other when one link (edge) or node fails. Next to that, various interesting subclasses of cubic graphs such as snarks (see Chapter 3) have certain connectivity requirements.

We first describe an algorithm to determine the connectivity of a graph and then describe a simple look-ahead to determine if graphs with the desired connectivity can still be generated from a given graph.

Note that cubic graphs are never 4-edge-connected since every cubic graph has a 3-edge-cut: the three incident edges of a vertex. We call this a *trivial* 3-edge-cut. To have a stronger notion of connectivity, we use the concept of cyclic edge-connectivity (see Section 1.1). Cubic graphs with girth 3 and more than 4 vertices are never cyclically 4-edge-connected since the 3 outgoing edges of a triangle form a non-trivial 3-edge-cut.

In general, the vertex-connectivity is less than or equal to the edge-connectivity (for a proof see [44]). For cubic graphs it is known that they are equal (see e.g. [6]).

We use a straightforward algorithm to test if a cubic graph G is k(-edge)-connected (it uses a standard depth-first search algorithm to find cutvertices). This algorithm is relatively efficient for small k, but as k increases it also becomes less efficient. For $k > 3$ (e.g. when it is also used for non-cubic graphs which are 4-connected), it becomes very expensive.

We used the following algorithm to filter cyclically 4-edge-connected cubic graphs (where $d(v, w)$ stands for the distance between two vertices $v, w \in V(G)$ in G):

Algorithm 2.2 Procedure to test if a cubic graph is cyclically 4-edge-connected.

Input: A 3-edge-connected cubic graph $G = (V, E)$ with girth at least 4.
  for all $v, w \in V : d(v, w) > 2$ **do**
    if $G[V \setminus \{v, w\}]$ is not 2-connected **then**
      return G is not cyclically 4-edge-connected.
    end if
  end for
  return G is cyclically 4-edge-connected.

Theorem 2.9. *A 3-edge-connected cubic graph G with girth at least 4 is cyclically 4-edge-connected if and only if there are no two vertices of G which are at distance at least 3 such that the graph obtained by removing these vertices is not 2-connected.*

Proof. The proof is split into 2 parts:

1. The algorithm only marks cyclically 4-edge-connected graphs as cyclically 4-edge-connected.

2. Every cyclically 4-edge-connected graph is marked as cyclically 4-edge-connected by the algorithm.

Proof of part 1:
Given a 3-edge-connected cubic graph G with girth at least 4 which is not cyclically 4-edge-connected. Thus G has a 3-edge-cut which disconnects G into two components which both contain a cycle. Call these components G_1 and G_2. Suppose that this 3-edge-cut consists of the edges $\{a, d\}, \{b, e\}$ and $\{c, f\}$ where $a, b, c \in V(G_1)$ and $d, e, f \in V(G_2)$. This is shown in Figure 2.16.

The vertices a, b, c, d, e, f are distinct, otherwise G is not 3-edge-connected. Note that a, b, c each have exactly one neighbour which is in $V(G_2)$ and d, e, f each have exactly one neighbour which is in $V(G_1)$, else $\{a, d\}, \{b, e\}, \{c, f\}$ was not a 3-edge-cut.

The graphs $G[\{a, b, c\}]$ and $G[\{d, e, f\}]$ contain at most one edge, otherwise G would contain a 2-edge-cut, but this is not possible since G is 3-edge-connected.

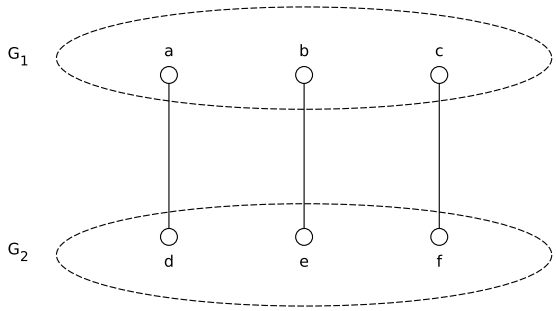


Figure 2.16: A non-trivial 3-edge-cut

So there is a vertex $x \in \{a, b, c\}$ such that x has no neighbour in $\{a, b, c\}$ and a vertex $y \in \{d, e, f\}$ such that y has no neighbour in $\{d, e, f\}$.

If x and y are no neighbours in G, x and y will removed by Algorithm 2.2 and a cutvertex will be found.

If x and y are neighbours, x is at distance more than 2 from a vertex $z \in \{d, e, f\} \setminus \{y\}$. So x and z will be removed by Algorithm 2.2 and a cutvertex will be found.

Thus G is marked as not being cyclically 4-edge-connected in each of the cases.

Proof of part 2:
Given a cyclically 4-edge-connected cubic graph G. Suppose that the algorithm finds a cutvertex after the removal of 2 vertices. So G has a vertex-cut S with $|S| = 3$. Consider a component $\bar{G}$ from $G[V \setminus S]$. Now we select for every $v \in S$ an edge as follows: if there is only one edge from v to $\bar{G}$, select that edge. Otherwise there are two edges from v to $\bar{G}$ and then the third incident edge of v is selected. This set of three edges T is a 3-edge-cut, since all paths from $V(\bar{G})$ to $V(G) \setminus (V(\bar{G}) \cup S)$ contain an edge from T.

However, since G is cyclically 4-edge-connected, one of the components obtained by removing the edges from T in G won't contain a cycle. We call this component G'. So G' is a tree with n vertices. Since G is cubic, G' has $3n - (n - 1) = n + 2$ outgoing edges in G (i.e. edges $\{a, b\}$ with $a \in V(G')$ and $b \in V(G) \setminus V(G')$). Since the outgoing edges form a 3-edge-cut, n must be 1. So G' is an isolated vertex. This means that the vertices from the 3-cut S are at distance at most 2, but this is not possible since Algorithm 2.2 only removes vertices which are at distance at least 3. So G is accepted by the algorithm. □

Algorithm 2.2 is applied as a filter to the graphs generated by the generation algorithm for all cubic graphs. Next to that, we also use a very simple look-ahead for the connectivity:

Note that the bundled triangle insertion operation does not change the edge-connectivity of a graph and that graphs constructed by this operation are not cyclically 4-edge-connected. The edge insertion operation can turn one or more k-edge-cuts into $(k+1)$-edge-cuts. In our implementation of the generation algorithm for all cubic graphs we maintain a list with the bridges of the graph G which is currently being constructed. A simple lower bound $min_{c_k}(G)$ for the minimal number of vertices required to obtain a (cyclically) k-edge-connected graph from G is:

$$min_{c_k}(G) = \begin{cases} 2 \cdot (k-1) & \text{if G contains a bridge} \\ 0 & \text{otherwise} \end{cases}$$

So in search of (cyclically) k-edge-connected cubic graphs with n vertices, we do not have to expand graphs G for which $|V(G)| + min_{c_k}(G)$ is larger than n.

In general, this simple look-ahead can only rarely be applied since most cubic graphs have no bridges, as can be seen from Table 2.6. Nevertheless, this look-ahead will be more effective when generating *snarks* [54] (that are cyclically 4-edge-connected cubic graphs that do not allow edge colourings with 3 colours), since most graphs that do not allow edge colourings with 3 colours have bridges. This is explained in more detail in Chapter 3.

Table 2.6 lists the counts for various classes of cubic graphs according to their connectivity. All results were independently verified by *minibaum* (which uses a different algorithm to determine the connectivities of a graph).

2.8 Testing and results

Our generator was used to generate all cubic graphs up to 32 vertices, with girth at least 3, 4 or 5. All graph counts were independently confirmed by running *minibaum*. They also agreed with previously published numbers (such as in [106]).

The graph counts, running times and a comparison with the fastest publicly available generator when present research began (i.e. *minibaum*) are given in Table 2.7. Our generator is called *snarkhunter*. The running times are for C code compiled by gcc and run on an Intel Xeon L5520 CPU at 2.27 GHz. They include writing the graphs to a null device. As *minibaum* depends more strongly on memory performance than *snarkhunter*, the speedup in the last column depends

strongly on the actual processor architecture. For other Intel Xeon processors it was considerably larger than given in Table 2.7.

So *snarkhunter* is more than 4 times faster than *minibaum* for generating all cubic graphs, more than 2 times faster than *minibaum* for generating cubic graphs with girth at least 4 and more than 5 times faster for generating cubic graphs with girth at least 5.

As it would take more than 3 weeks to generate all cubic graphs with 30 vertices on a single CPU, we split the generation into independent parts and execute them on multiple CPU's. This can be done by the method described in [89]: each CPU generates the whole generation tree up to some level ℓ and then just its own portion of the tree beyond that level. Since unrelated branches of the tree are entirely independent, this is easily implemented. For the generation of the cubic graphs with 30 and 32 vertices we used $\ell = 24$ and found that the overhead in parallelisation was a negligible portion of the total.

2.9 Closing remarks

As a measure of how much extra improvement might be possible, we note that the mere act of copying the graphs to an output buffer (with no alteration to the internal structure except for inserting a few null characters), and writing them to the output, contributes 5–9% of the running time.

One of the main motivations for the new generator was the generation of *snarks* [54] – that is cyclically 4-edge-connected cubic graphs that do not allow edge colourings with 3 colours. This subclass of cubic graphs is especially interesting as a source for possible counterexamples to graph theoretic conjectures. So far, the fastest way to get complete lists was to use *minibaum* and filter the output (see [35]). In our generator, the edge insertion operation allows a limited look-ahead detecting a lot of cases where the resulting graph will be 3-edge-colourable. This results in a speedup compared to previous methods that even exceeds the speedup for the classes of graphs discussed here. The algorithm for the efficient generation of snarks is discussed in Chapter 3.

The pruning techniques from Section 2.6 to generate cubic graphs with girth at least 4 or 5 are likely to become rather less efficient for larger lower bounds on the girth. Therefore future work could include the development of a specialised algorithm for the generation of cubic graphs with large lower bounds on the girth.

The latest version of *snarkhunter* (i.e. the program implementing the algorithm described here) can be downloaded from [27].

	All connected cubic graphs			
$\|V(G)\|$	connected	2-connected	3-connected	cyc. 4-connected
4	1	1	1	1
6	2	2	2	1
8	5	5	4	2
10	19	18	14	5
12	85	81	57	18
14	509	480	341	84
16	4 060	3 874	2 828	607
18	41 301	39 866	30 468	6 100
20	510 489	497 818	396 150	78 824
22	7 319 447	7 187 627	5 909 292	1 195 280
24	117 940 535	116 349 635	98 101 019	20 297 600
26	2 094 480 864	2 072 540 352	1 782 392 646	376 940 415

	Connected cubic graphs with girth at least 4			
$\|V(G)\|$	connected	2-connected	3-connected	cyc. 4-connected
4	0	0	0	0
6	1	1	1	1
8	2	2	2	2
10	6	6	6	5
12	22	22	21	18
14	110	109	104	84
16	792	788	750	607
18	7 805	7 772	7 486	6 100
20	97 546	97 292	94 666	78 824
22	1 435 720	1 433 333	1 404 963	1 195 280
24	23 780 814	23 754 936	23 396 452	20 297 600
26	432 757 568	432 431 403	427 301 078	376 940 415

	Connected cubic graphs with girth at least 5			
$\|V(G)\|$	connected	2-connected	3-connected	cyc. 4-connected
10	1	1	1	1
12	2	2	2	2
14	9	9	9	9
16	49	49	49	49
18	455	455	455	454
20	5 783	5 783	5 782	5 775
22	90 938	90 937	90 927	90 807
24	1 620 479	1 620 471	1 620 319	1 618 266
26	31 478 584	31 478 466	31 476 211	31 438 578
28	656 783 890	656 782 262	656 745 447	656 052 352

Table 2.6: Counts for various classes of cubic graphs according to their connectivity. Note that every cyclically 4-edge-connected cubic graph with more than 4 vertices has girth at least 4 since the 3 outgoing edges of a triangle form a non-trivial 3-edge-cut.

	All connected cubic graphs			
$\|V(G)\|$	# graphs	snarkhunter (s)	minibaum (s)	speedup
20	510 489	1.2	5.7	4.75
22	7 319 447	16	74	4.59
24	117 940 535	261	1 166	4.47
26	2 094 480 864	4 826	20 748	4.30
28	40 497 138 011	100 179	440 870	4.40
30	845 480 228 069	2 240 049		
32	18 941 522 184 590	53 177 371		

	Connected cubic graphs with girth at least 4			
$\|V(G)\|$	# graphs	snarkhunter (s)	minibaum (s)	speedup
20	97 546	0.8	2.5	3.13
22	1 435 720	11	34	2.99
24	23 780 814	191	542	2.85
26	432 757 568	3 626	10 107	2.79
28	8 542 471 494	76 218	216 837	2.84
30	181 492 137 812	1 756 557		
32	4 127 077 143 862	42 288 975		

	Connected cubic graphs with girth at least 5			
$\|V(G)\|$	# graphs	snarkhunter (s)	minibaum (s)	speedup
20	5 783	0.1	0.6	6.00
22	90 938	1.7	9.3	5.47
24	1 620 479	27	158	5.79
26	31 478 584	519	3 047	5.87
28	656 783 890	11 073	63 821	5.76
30	14 621 871 204	275 251		
32	345 975 648 562	6 473 440		

Table 2.7: Counts and generation times for classes of cubic graphs. The running times for *minibaum* for $|V(G)| > 28$ were omitted as these computations were performed on multiple heterogeneous clusters.

Chapter 3

Generation of snarks

In this chapter we show how the generation algorithm for cubic graphs from Chapter 2 can be extended to generate all non-isomorphic snarks and weak snarks of a given order quite efficiently. A weak snark is a simple, cyclically 4-edge-connected cubic graph with chromatic index 4. If the girth of a weak snark is at least 5, it is a snark. Snarks are of interest since for several interesting open conjectures it can be proven that if the conjecture is false, the smallest counterexample is a snark.

Our implementation of this new algorithm is more than 14 times faster than previous programs for generating snarks and more than 29 times faster for generating weak snarks. Using this generator we were able to generate all non-isomorphic snarks up to 36 vertices, which was impossible with previous methods. Previously complete lists of snarks up to 32 vertices were known.

We tested several open conjectures on these new lists of snarks. We verified that various conjectures (such as the (Strong) Cycle Double Cover conjecture) hold for all snarks up to 36 vertices. Next to these positive results we were also able to refute 8 published conjectures (amongst others of Bill Jackson and Cun-Quan Zhang) about cycle coverings and the general cycle structure of cubic graphs.

Remark: our algorithm for the generation of snarks is also described in [24]. Various parts of this chapter were adopted from that article. The investigation of invariants was joint work also with Jonas Hägglund and Klas Markström.

3.1 Introduction

A number of problems in graph theory can be solved in the general case if they can be solved for cubic graphs. Examples of such problems are the four colour problem (now a theorem), many of the problems concerning cycle double covers and surface embeddings of graphs, coverings by matchings and the general structure of the cycle space of a graph. For most of these problems one can additionally constrain this class of graphs to the subclass of cyclically 4-edge-connected cubic graphs with chromatic index 4. The chromatic index $\chi'(G)$ of a graph G is the minimum number of colours required for an edge colouring of that graph such that no two adjacent edges have the same colour. It follows from Vizing's famous theorem (which states that $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$) that a cubic graph has chromatic index 3 or 4.

Isaacs [71] called cubic graphs with chromatic index 3 *colourable* graphs and those with chromatic index 4 *uncolourable* graphs. All cubic graphs with bridges can easily be seen to be uncolourable and are therefore considered to be trivial. Uncolourable cubic graphs with cycle separating 2- or 3-cuts or 4-cycles can be constructed from smaller uncolourable graphs by certain standard operations, which also behave well with respect to most of the open problems at hand. So this generalises the notion of *trivial* uncolourable cubic graphs. Often minimal counterexamples to many problems reside, if they exist at all, among the remaining uncolourable cubic graphs, which are called *snarks* based on an article by Gardner [54] from 1976 who used the term with weaker connectivity requirements. He introduced snarks as possible counterexamples for the four colour theorem. Gardner called these graphs snarks after Lewis Caroll's poem *"The Hunting of the Snark"*. In that poem a snark is a mythical creature which is very hard to find. Later on, stronger criteria for non-triviality have also been proposed [96, 35], but in this thesis we focus on snarks.

For various interesting open conjectures it has been proven that if the conjecture is false, there is a snark that is a counterexample and very often it has also been proven that the smallest counterexample is a snark. This is amongst others the case with the Cycle Double Cover conjecture [116, 114] and Tutte's 5-flow conjecture [119].

So far the structure of the class of all snarks is not entirely understood, there is e.g. at present no known uniform random model for snarks, hence leaving us without a theoretical method for studying the typical behaviour of snarks. The only available alternatives have been to study the smallest snarks and certain families of snarks given by specialised constructions. The Petersen graph is the

smallest snark. In 1974 Isaacs [71] gave constructions for the first infinite families of snarks (amonst others for the flower snarks, see Figure 3.1). In 1996 Kochol [79] published a method for the construction of snarks with arbitrarily high girth.

Lists of snarks have already been given in [35, 41, 66, 39], but so far no specialised computer program for generating all snarks existed. The fastest program was the one used in [35] which was based on the program described in [11] (i.e. *minibaum*) and – just like the approach in [39] – is simply a generator for all cubic graphs with a lower bound on the girth, combined with a filter for colourability and cyclic connectivity at the end.

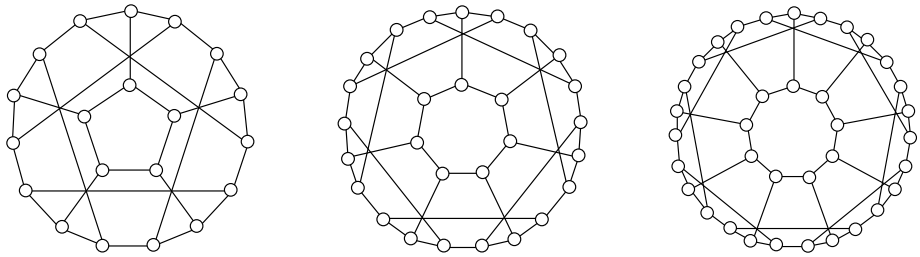


Figure 3.1: Isaacs flower snarks J_5, J_7 and J_9 [71] (i.e. Figures 3.1a, 3.1b and 3.1c, respectively). The flower snarks J_{2n+1} for $n > 4$ are defined similarly.

However, since the proportion of snarks among the cubic graphs rapidly decreases as the number of vertices increases (and is asymptotically zero [107]), this approach is not feasible for sizes even a few steps beyond those previously published. In this chapter we give a new algorithm which augments the efficient algorithm for generating cubic graphs from Chapter 2 with look-aheads which make it possible to avoid constructing many of the colourable cubic graphs, thus reducing the number of graphs passed to the final filter. This algorithm is described in Section 3.3 and 3.4.

Our implementation of this algorithm is more than 14 times faster than previous programs for generating snarks and more than 29 times faster for generating weak snarks. This allowed us to generate all non-isomorphic snarks up to 36 vertices and all weak snarks up to 34 vertices. We tested several conjectures on these new lists of snarks and were able to refute 8 published conjectures. This is described in Section 3.5.

3.2 Definitions

This section contains some definitions which are specific to this chapter. For definitions of more general concepts, see Section 1.1.

We call cubic graphs 3-edge-colourable or simply *colourable* if they have chromatic index 3. Those with chromatic index 4 are called *uncolourable*.

A *weak snark* is an uncolourable cyclically 4-edge-connected cubic graph (so it has girth at least 4). A *snark* is an uncolourable cyclically 4-edge-connected cubic graph with girth at least 5.

3.3 The generation algorithm

As mentioned in the introduction, the older programs for generating snarks were built by adding a filter for graphs with the desired properties to a program which generates all cubic graphs of a given order. The efficiency of an approach that generates a larger class of graphs and filters the output for graphs in a smaller class depends on one hand on the cost for the filter and on the other on the ratio between the number of graphs in the large and small class. On both criteria the generation of all cubic graphs with girth 4 and filtering for weak snarks scores badly: testing for 3-edge-colourability is NP-complete [67] and already for 28 vertices only 0.00015% of the cubic graphs with girth at least 4 are weak snarks (and only 0.00044% of the cubic graphs with girth at least 5 are snarks). In fact the ratio is even decreasing with the number of vertices. Nevertheless no better way to generate weak snarks was known until now. In this chapter we present a method that – although not generating only weak snarks – at least allows a limited look-ahead and increases the ratio of weak snarks among the graphs with the largest number of vertices by a factor of 85 to about 0.0135% for 28 vertices (and by a factor of 20 to about 0.0076% for snarks). But also for this approach the ratio of (weak) snarks is decreasing with the number of vertices.

The new algorithm is based on the algorithm described in Chapter 2. Prime graphs are never 3-connected, so they cannot be snarks. Thus the last operation for generating a snark will always be an *edge insertion operation* (see Section 2.2). Applying this operation to two vertices having a common endpoint can be seen as that of replacing the common endpoint by a triangle. Triangles are constructed by bundling these operations (see Section 2.4 for details), but since weak snarks have girth at least 4, the last operation will always be a single edge insertion operation applied to two non-adjacent edges.

As the number of cubic graphs grows fast with the number of vertices, just avoiding the insertion of the last edge gives already a considerable speedup.

We use the following well-known result:

Lemma 3.1. *A cubic graph is 3-edge-colourable if and only if it has a 2-factor where all cycles have even length, also called an even 2-factor.*

This gives us criteria to decide that graphs obtained by certain edge insertion operations will be 3-edge-colourable:

Lemma 3.2. *Given an even 2-factor F in a cubic graph G. All graphs G' obtained by applying the edge insertion operation to two edges e, e' which are part of the same cycle in F will be 3-edge-colourable.*

Replacing e respectively e' in F with the two edges that result from the subdivision gives an even 2-factor in G'.

Lemma 3.2 is applied by using that for any two colours $a \neq b$ of a 3-edge-colouring, the sets of edges coloured a or b form a 2-factor where all cycles have even length:

Theorem 3.3. *Given a cubic graph G and a 3-edge-colouring of G. If two edges e, e' belong to the same cycle of a 2-factor induced by two different colours, the graph G' obtained by applying the edge insertion operation to e, e' will be 3-edge-colourable.*

This implies the well-known fact that if under the above circumstances e, e' share a vertex, G' will always be 3-edge-colourable. So if G is 3-edge-colourable, all graphs obtained from G by the bundled triangle insertion operation will also be colourable.

Assume that snarks on n vertices are to be constructed and a graph G on $n - 2$ vertices has to be expanded. Then the algorithm first tries to construct a 3-edge-colouring of G. If G does not have a 3-edge-colouring, Theorem 3.3 cannot be applied. However, the ratio of graphs without a 3-edge-colouring (and without bridges) is very small. For example only 0.03% of the 2-connected graphs with 28 vertices and girth at least 4 have chromatic index 4 (and this percentage is decreasing as the number of vertices increases). If a 3-edge-colouring is found, then the complement of each colour class is an even 2-factor. So each 3-edge-colouring of G gives three even 2-factors. When computing the edge pairs for expansions, edge pairs with both edges in the same cycle of one of these 2-factors are not considered.

Computing different 3-edge-colourings – that is 3-edge-colourings producing different 2-factors – allows to detect more edge pairs that do not lead to snarks. We compute different 3-edge-colourings in two ways: interchanging the colours in a non-hamiltonian cycle in one of the 2-factors gives a colouring where two of the three induced 2-factors are different. So this is an efficient way to get a different 3-edge-colouring. We use one such modified colouring. Another way is to compute a different colouring from scratch by forcing some initial colours in order to guarantee a different colouring.

For 28 vertices the first colouring allows on average to discard 84% of the edge pairs. The colour changes in the non-hamiltonian cycles discards another 11%. A second colouring discards 3.4%. These percentages are approximately the same for generating snarks instead of only generating weak snarks. The cost for computing a third colouring turned out to be higher than the gain. So in total about 98.4% of the edge pairs are discarded.

Theorem 3.4 gives another criterion to avoid the generation of 3-edge-colourable graphs. This theorem is folklore in the community studying non-triviality of snarks, but we will give a short proof here.

Theorem 3.4. *Given a 3-edge-colourable graph G, all graphs G' obtained from G' by the edge insertion operation so that the edge with the two new end-vertices is part of a 4-cycle in G' are 3-edge-colourable.*

Proof. Given a 3-edge-colourable graph G and $e = \{a, b\}, e'' = \{c, d\}$ with a, b, c, d pairwise different and $e' = \{a, c\}$ with $e, e', e'' \in E(G)$. Furthermore let $c : E(G) \to \{1, 2, 3\}$ be a colouring with (w.l.o.g.) $c(e) = 1$ and $c(e') = 2$. If $c(e'') = 1$ then e, e'' belong to the same $1, 2$-cycle in the colouring and the result follows from Lemma 3.2. So assume $c(e'') = 3$ and that e, e'' do not belong to the same $1, 3$-cycle in the colouring (otherwise the result follows immediately). Then we can exchange the colours in the $1, 3$-cycle of e'' and obtain a colouring with e, e'' of the same colour. So we are in a situation for which we already proved the theorem. □

So if we want to generate all snarks with n vertices, we do not have to apply the edge insertion operation to 3-edge-colourable graphs with $n - 2$ vertices if the inserted edge will be part of a square. The fact that such an edge pair does not have to be expanded may also be discovered by the normal routine or by changing colours in a non-hamiltonian cycle in the colouring, but deleting such edge pairs in advance is much more efficient.

At first sight Theorem 3.4 looks only interesting for operations that do not produce triangles, because in the last step we never produce triangles. But in fact for triangles it allows an earlier look-ahead. We refer the reader to Chapter 2 for details about the isomorphism rejection, but one fact is important in this context: for graphs with girth at most 4, the last inserted edge is always an edge in the smallest cycle (see Section 2.5.1). So assume that we have a graph G with $n-2$ vertices with at least one triangle and that we want to construct weak snarks with n vertices. From this graph we can only get graphs with girth at most 4, so the last edge inserted will be in a 4-gon. It follows from Theorem 3.4 that if G was 3-edge-colourable, all graphs with n vertices obtained from G will also be 3-edge-colourable. Thus we do not have to construct 3-edge-colourable graphs with triangles of size $n-2$. So for triangles Theorem 3.4 gives a bounding criterion that can already be applied on level $n-4$. In fact the generation algorithm *bundles* triangle operations (see Section 2.4), but this is only a difference in detail.

The generator for cubic graphs from Chapter 2 also contains look-aheads to generate cubic graphs with girth at least 5 quite efficiently (see Section 2.6 for details). As the construction operations are the same, we can apply the look-aheads for colourability in combination with these and generate snarks directly instead of generating all weak snarks and filtering them for girth 5.

3.4 Optimisations

3.4.1 The order of applying filters

Snarks are very rare: e.g. for 28 vertices only 0.00044% of the cubic graphs with girth at least 5 are snarks (and only 0.00015% of the cubic graphs with girth at least 4 are weak snarks). However most of the cubic graphs with girth at least 4 are cyclically 4-edge-connected: e.g. more than 87% of the cubic graphs with 26 vertices and girth at least 4 are cyclically 4-edge-connected (see Table 2.6 in Section 2.7). So the requirement of having chromatic index 4 is much more restrictive than that of being cyclically 4-edge-connected.

Therefore, when a graph with the desired number of vertices is generated, it is first tested if it has chromatic index 4 and only if this is the case, it is tested if it is also cyclically 4-edge-connected. Testing for 3-edge-colourability is NP-complete, but nearly all of the tested graphs are 3-edge-colourable and usually a colouring is found very quickly.

3.4.2 Postponing isomorphism rejection

Recall from Section 2.5.1 that we assign a 7-tuple $(x_0, \ldots, x_6)$ to every reducible edge and choose a reducible edge with the largest 7-tuple as canonical reduction. The values of $x_0, \ldots, x_4$ are combinatorial invariants of increasing discriminating power and cost. The computation of x_5, x_6 is more expensive and involves calling *nauty* to compute a canonical labelling (and the automorphism group of the graph).

Even though the discriminating power of $x_0, \ldots, x_4$ is enough to avoid calling *nauty* in a lot of cases, *nauty* is still one of the most time consuming parts of the algorithm. (Note that *nauty* is also called to compute the automorphism group of the graph which is then used to compute the equivalence classes of expansions.)

Therefore we are using another optimisation which avoids even more *nauty* calls. Assume that we want to generate all non-isomorphic snarks with n vertices. Since there are only very few snarks, we postpone calling *nauty* for graphs with $n - 2$ vertices and only call *nauty* if snarks were generated from these graphs.

More specifically, when a graph with $n - 2$ vertices is generated and we still have to call *nauty* to decide if it is canonical (i.e. the last operation was an edge insertion operation and the graph has multiple reducible edges with maximal value of $x_0, \ldots, x_4$), we do not call *nauty* but act as if it was canonical. We also do not call *nauty* to determine the equivalence classes of possible expansions, but act as if the graph had a trivial automorphism group (i.e. every expansion is in a different equivalence class).

Of course now isomorphic or non-canonical graphs may be generated. Therefore the snarks which are generated from this parent are saved in a list. If at least one snark was generated from this parent, we call *nauty* to test if the parent graph was canonical (unless it was already known that the parent graph is canonical – i.e. if the inserted edge was the only edge with maximal value of $x_0, \ldots, x_4$ in the parent graph). If the parent is not canonical, the snarks which were generted from it are rejected. If the parent is canonical, the snarks which were generated might be isomorphic, since we applied the edge insertion operation to all eligible edge pairs instead of to only one edge pair for each orbit of eligible edge pairs.

If we had to call *nauty* to test if the parent graph is canonical, we already know the automorphism group of the parent graph, else we compute it. If the automorphism group is trivial, the graphs which were generated from this parent will not be isomorphic. If this is not the case, we investigate the list of children and filter the isomorphic copies. More specifically we call *nauty* to compute a canonical form for each of the children, compare the canonical forms and output

the non-isomorphic children. Computing such a canonical labelling is not a bottleneck for the algorithm since there are only very few snarks: most parents do not have any children which are snarks and if they do have such children, there are only very few of them. As can be seen from Table 3.1, parents which have snarks as children often only have one child.

# children	percentage
1	95.72
2	2.26
3	0.32
4	1.21
5	0.17
6	0.15
≥ 7	0.06

Table 3.1: Percentage of graphs which have a given number of (possibly isomorphic) children with 28 vertices which are snarks among the graphs with 26 vertices which are generated and which have at least one child and a non-trivial automorphism group.

For the graphs with n vertices which are generated, we also postpone calling *nauty*. If there are multiple edges with maximal value of $(x_0, ..., x_4)$ (and the inserted edge is one of them), we first test if the graph is 3-edge-colourable. Only if this is not the case, we call *nauty* to compute x_5, x_6 and test if it is canonical.

The coarse pseudocode of the algorithm for generating all non-isomorphic snarks using this optimisation is shown in Algorithm 3.1 (cf. Algorithm 1.1 from Section 1.3).

Algorithm 3.1 Construct_snarks(graph G)

```
if G has n vertices then
  if G is not 3-edge-colourable and cyclically 4-edge-connected then
    if G is generated by a canonical expansion then
      save in list of snarkchildren
    end if
  end if
else
  find expansions
  if G has n − 2 vertices then
    clear list of snarkchildren
    for each expansion X do
      perform expansion X
      if inserted edge has maximal value of (x_0, ..., x_4) then
        Construct_snarks(expanded graph)
      end if
      perform reduction X^{-1}
    end for
    if list of snarkchildren is not empty and G is canonical then
      output non-isomorphic snarkchildren
    end if
  else
    compute classes of equivalent expansions
    for each equivalence class do
      choose one expansion X
      perform expansion X
      if expanded graph has n − 2 vertices then
        if inserted edge has maximal value of (x_0, ..., x_4) then
          Construct_snarks(expanded graph)
        end if
      else
        if expansion is canonical then
          Construct_snarks(expanded graph)
        end if
      end if
      perform reduction X^{-1}
    end for
  end if
end if
```

3.5 Testing and results

3.5.1 Running times and the number of snarks

Our generator (which is called *snarkhunter*) was used to generate all snarks up to 36 vertices and all weak snarks up to 34 vertices. We compared all snarks and weak snarks up to 32 vertices with the snarks which were generated by the program *minibaum* [11]. The results were in complete agreement. For all snarks which were generated, we used an independent program to check the chromatic index, the cyclic edge-connectivity, the girth and whether they are all pairwise non-isomorphic.

The snark counts and statistics about their girth and cyclic edge-connectivity are listed in Table 3.2. Previously only the lists of all snarks up to 28 vertices were published [35], but *minibaum* had already been used to generate all snarks up to 32 vertices. The sizes of the automorphism group of all snarks up to 36 vertices are listed in Table 3.3.

The running times of *snarkhunter* up to 34 vertices and a comparison with the fastest generator for snarks when the present research began (i.e. *minibaum*) are given in Table 3.4. As can be seen in that table, *snarkhunter* is approximately 29 times faster than *minibaum* for generating weak snarks and 14 times faster for generating snarks and the speedup seems to be increasing with the number of vertices. The generation of snarks with more than 28 vertices was split into multiple independent parts and was executed on multiple CPU's. We used the same parallelisation method as for our generator for cubic graphs from Chapter 2 (see Section 2.8 for details). Also in case of the generation of snarks the overhead for parallelisation was only a negligible portion of the total running time. The generation of all snarks with 36 vertices was executed on multiple heterogeneous clusters (primarily on Intel Harpertown 2.66 GHz CPU's). The total running time for generating all snarks with 36 vertices was approximately 73 CPU years. An estimate based on running a partial generation for 38 vertices is that generating all snarks of this order would require 1100 CPU years (on the clusters used for generating all snarks with 36 vertices).

All snarks up to 36 vertices and all weak snarks up to 34 vertices can be downloaded from [23] and the latest version of *snarkhunter* can be downloaded from [27].

$\|V(G)\|$	# weak snarks	$\lambda_c(G) \geq 4$	$\lambda_c(G) \geq 5$	$\lambda_c(G) \geq 6$	$g(G) \geq 6$
10	1	1	1	0	0
12,14,16	0	0	0	0	0
18	2	2	0	0	0
20	6	6	1	0	0
22	31	20	2	0	0
24	155	38	2	0	0
26	1 297	280	10	0	0
28	12 517	2 900	75	1	1
30	139 854	28 399	509	0	1
32	1 764 950	293 059	2953	0	0
34	25 286 953	3 833 587	19 935	0	0
36	?	60 167 732	180 612	1	1

Table 3.2: The number of snarks. We write $\lambda_c(G)$ for the cyclic edge-connectivity of a graph G and $g(G)$ for the girth. None of the snarks listed here has $g(G) \geq 7$. The programs to compute the cyclic edge-connectivity and girth were independently developed in Belgium and Sweden and the results were in complete agreement.

3.5.2 Testing conjectures on snarks

We tested the snarks in cooperation with Jonas Hägglund and Klas Markström from Umeå University in Sweden. Together we tested a large number of conjectures and properties of the new lists of snarks. All of these tests were independently implemented and executed by the groups in Belgium and Sweden and all results were in complete agreement.

More specifically we tested 22 published conjectures on the new lists of snarks. For most of these conjectures it has been proven that a minimal counterexample must be a weak snark. For some of them it has been proven that a minimal counterexample must be a snark (i.e. have girth at least 5). This is for example the case with the *cycle double cover* conjecture. A *cycle double cover* (in short: CDC) of a graph G is a collection of cycles C such that every edge of G is contained in exactly 2 cycles in C. The cycle double cover conjecture (CDCC) is one of the most famous long-standing open problems in graph theory. It was independently formulated by Szekeres and Seymour in 1973 and 1979 respectively.

Conjecture 3.5 (CDCC, Szekeres [116], Seymour [114])**.** *Every bridgeless graph has a cycle double cover.*

Order	1	2	3	4	6	8	12	16	20	24	28	32	36	48	64	80	120
10																	1
18				1		1											
20	2	1		2					1								
22	4	11		1		1	2	1									
24	21	9		8													
26	174	75		23		7		1									
28	2536	290		62	1	6	2	2			1						
30	26 214	1924		226		25		9								1	
32	278 718	13 284		973		78		6									
34	3 684 637	143 783	7	4798	7	329	1	20		3		1		1			
36	58 191 667	1 950 129	2	24 855	3	1044	3	24				2	1	1	1		

Table 3.3: Counts of all snarks according to the order of their automorphism group.

$\|V(G)\|$	# weak snarks	snarkhunter (s)	minibaum (s)	speedup
22	31	1.5	41	27.33
24	155	24	643	26.79
26	1 297	430	11 629	27.04
28	12 517	8 868	259 495	29.26
30	139 854	209 897		
32	1 764 950	4 976 553		
34	25 286 953	119 586 562		

$\|V(G)\|$	# snarks	snarkhunter (s)	minibaum (s)	speedup
22	20	1	11	11.00
24	38	15	182	12.13
26	280	265	3 458	13.05
28	2 900	5 304	75 194	14.18
30	28 399	128 875		
32	293 059	2 875 911		
34	3 833 587	66 519 829		

Table 3.4: Counts and generation times for snarks and weak snarks. The running times are for C code compiled by gcc and run on an Intel Xeon L5520 CPU at 2.27 GHz. They include writing the snarks to a null device. The running times for snarks with 36 vertices are omitted as this computation was executed on multiple heterogeneous clusters.

The conjecture has been proven for some large classes of graphs such as planar graphs. It is also well-known that if the conjecture holds for cubic graphs, then it is true for all graphs. If a cubic graph is 3-edge-colourable, then the cover given by each pair of colours from a given proper colouring of the edges is a CDC. It is also well-known that a minimum counterexample must have girth at least 5 and be cyclically 4-edge-connected (see e.g. [73]), and hence it is sufficient to prove the CDCC for snarks. These conditions have been strengthened over the years and we now know that a minimum counterexample must have girth at least 12 (see [68]) and must therefore be much larger than the snarks under investigation in this thesis.

However, there are various stronger versions of the CDCC and auxiliary conjectures aimed at proving some version of the CDCC, which are not known to hold for small snarks. One of those strengthenings of the CDCC is the strong cycle double cover conjecture (SCDCC) which was formulated by Goddyn [56].

Conjecture 3.6 (SCDCC, Goddyn [56])**.** *Let G be a bridgeless graph. Then for every cycle C in G there is a CDC that contains C.*

Using our lists of snarks, we verified that the strong cycle double cover conjecture has no counterexamples with 36 vertices or less.

Next to this and other positive results, we were also able to refute 8 published conjectures about cycle coverings and the general cycle structure of cubic graphs. These include conjectures of Bill Jackson [72] and Cun-Quan Zhang [126]. This shows that snarks are not only theoretically a good source for counterexamples to conjectures, but also in practice.

In [126] Zhang conjectured the following:

Refuted Conjecture 3.7 (Zhang [126])**.** *Let G be a cubic cyclically 5-edge-connected permutation graph. If G is a snark, then G must be the Petersen graph.*

A *permutation graph* is a cubic graph which has a 2-factor that consists of 2 induced cycles (i.e. they have no chords).

We tested this conjecture on all cyclically 5-edge-connected snarks with at most 36 vertices. We found 12 cyclically 5-edge-connected permutation snarks with 34 vertices which provide counterexamples to Conjecture 3.7. One of these graphs can be found in Figure 3.2, where the 2-factor that consists of 2 induced cycles is drawn in bold. There are no counterexamples with 36 vertices. The entire set of counterexamples can be obtained from *House of Graphs* [15] by searching for the keywords "counterexample * Zhang".

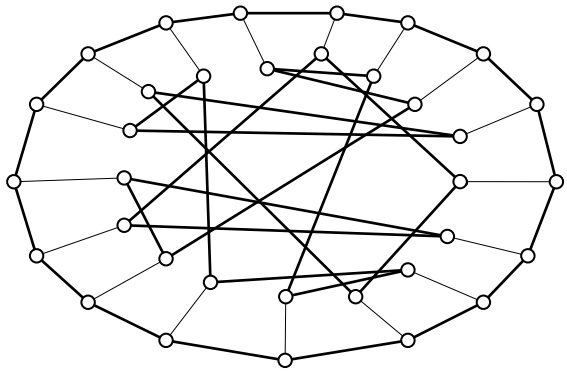


Figure 3.2: One of the twelve cyclically 5-edge-connected permutation snarks with 34 vertices. The bold cycles correspond to the 2-factor.

The following was posed as a problem in [72] and as a conjecture in [126]:

Refuted Conjecture 3.8 (Jackson [72], Zhang [126])**.** *Let G be a cyclically 5-edge-connected cubic graph and $\mathcal{D}$ be a set of pairwise disjoint cycles of G. Then $\mathcal{D}$ is a subset of a CDC, unless G is the Petersen graph.*

We tested this conjecture on all cyclically 5-edge-connected snarks with at most 36 vertices. The same 12 graphs from Conjecture 3.7 are also counterexamples for this conjecture and there are no additional counterexamples with 34 vertices. One of the counterexamples is shown in Figure 3.2, where the set of disjoint cycles that cannot be extended to a CDC is the bold 2-factor. The 2-regular subgraphs of these 12 graphs that cannot be part of any CDC are 2-factors which have exactly two induced components. There are also 44 cyclically 5-edge-connected snarks with 36 vertices with 2-regular subgraphs, not all of which are 2-factors, that cannot be part of any CDC.

The entire set of counterexamples can be obtained from *House of Graphs* [15] by searching for the keywords "counterexample * Jackson".

More details about the properties we investigated and the conjectures we tested can be found in [24].

3.6 Closing remarks

The results of the analysis of the snarks of order at most 36 show that some of the intuition gained from the study of the smallest snarks has been misleading.

$\|V(G)\|$	# snarks	growth
20	6	3.00
22	20	3.33
24	38	1.90
26	280	7.37
28	2 900	10.36
30	28 399	9.79
32	293 059	10.32
34	3 833 587	13.08
36	60 167 732	15.69

Table 3.5: Growth rate of the number of snarks.

Some of the unexpected behaviour of snarks can only be found for sufficiently large snarks. Table 3.5 shows the growth rate of the number of snarks. It shows a change at 34 vertices. Together with the fact that most counterexamples we found have at least 34 vertices, this could indicate that we might have just reached the size range where the snarks display a behaviour which is more typical for large snarks.

We believe that the results which we obtained also give a good demonstration of the importance of large scale computer based generation and analysis as tools for aiding our understanding of combinatorial problems.

Generating all weak snarks with 36 vertices with *snarkhunter* is probably computationally still feasible. Generating all snarks with 38 vertices might be just feasible, but it will require an excessive amount of computing time (probably 1100 CPU years). Next to that, even though only $5.22 \cdot 10^{-7}\%$ of the connected cubic graphs with 36 vertices are snarks, testing conjectures on all of the 60 167 732 snarks with 36 vertices is quite expensive and as can be seen from Table 3.5 the growth rate is increasing. So it would be very interesting if the class of snarks could be further restricted and still keep the property that the class contains smallest possible counterexamples for a lot of conjectures.

Jaeger [73] for example introduced the notion of strong snarks. A snark G is called *strong* if for every edge $e \in E(G)$ the result of the edge reduction (see Section 2.2) is not 3-edge-colourable. Celmins [40] proved that a minimum counterexample to the cycle double cover conjecture must be a strong snark.

We tested our list of snarks for this property and determined that there are 7 strong snarks with 34 vertices and 25 with 36 vertices. There are no strong

snarks of smaller order (which again indicates that 34 vertices seems to be a point of change).

However only for a very limited number of conjectures it is proven that minimal counterexamples must be strong snarks. None of the counterexamples which enabled us to refute the 8 conjectures were strong snarks.

Other possible strengthenings of snarks include for example more restrictions on the cyclic edge-connectivity. Future work could include the development of a specialised generation algorithm for such a more restricted subclass of snarks.

Chapter 4

Generation of fullerenes

In this chapter we describe an efficient new algorithm for the generation of fullerenes. Our implementation of this algorithm is more than 3.5 times faster than the previously fastest generator for fullerenes – *fullgen* – and the first program since *fullgen* to be useful for more than 100 vertices. Contradictory results of our program and *fullgen*, led to the detection of a non-algorithmic error in *fullgen* that caused problems for 136 vertices or more.

The new generator allowed us to generate all non-isomorphic fullerenes up to 400 vertices. We also check a conjecture of Barnette that cubic planar graphs with maximum face size 6 are hamiltonian up to 316 vertices and prove that the smallest counterexample to the spiral conjecture has 380 vertices.

We also develop a specialised algorithm for the generation of Isolated Pentagon Rule (IPR) fullerenes. Our implementation of this algorithm is also significantly faster than other generators for IPR fullerenes.

Remark: our algorithm for the generation of fullerenes is also described in [29]. This was joint work also with Brendan D. McKay and various parts of this chapter were adopted from that article.

4.1 Introduction

4.1.1 Definitions

Fullerenes are spherical carbon molecules that can be modelled as cubic plane graphs where all faces are pentagons or hexagons. We will refer to these mathematical models also as fullerenes. Euler's formula implies that a fullerene with n

vertices contains exactly 12 pentagons and $n/2 - 10$ hexagons.

A *planar graph* is a graph which can be drawn in the plane (or equivalently: on the sphere) without crossing edges. A *plane graph* is a planar graph together with a crossing-free drawing (i.e. an embedding). For computer programs the concept of crossing edges is inconvenient, therefore we use the combinatorial representation of an embedding. Up to (combinatorial) isomorphism, an embedding can be uniquely encoded by a combinatorial structure. In combinatorics the embedding of a graph is represented by giving the cyclic order (say: clockwise order) of the incident edges of each vertex. So a plane graph $G = (V, E, next())$ is a graph (V, E) where the function $next()$ specifies the clockwise order of the incident edges of each vertex. This representation for plane graphs also works more generally for orientable surfaces. The mirror image of a plane graph G is obtained by reversing the cyclic order at each vertex of G.

A cyclic order of a set of elements is a way of arranging these elements on a cycle with a chosen direction. So the cyclic order (a, b, c, d) represents the orders $[a, b, c, d]$, $[d, a, b, c]$, $[c, d, a, b]$ and $[b, c, d, a]$.

For plane graphs next to the usual notion of graph isomorphism (see Section 1.1), also the notion of isomorphism of embedded graphs exists.

Recall from Section 1.1 that an isomorphism from a graph G to a graph G' is a bijective function $\phi : V(G) \to V(G') : \{v, w\} \in E(G) \iff \{\phi(v), \phi(w)\} \in E(G')$. We will also call this an *abstract isomorphism*.

Given two plane graphs $G = (V, E, next())$ and $G' = (V', E', next'())$. An *orientation-preserving isomorphism* from G to G' is an abstract isomorphism ϕ from G to G' which preserves the embedding: if $(v_1, v_2, ..., v_k)$ is the cyclic order of the neighbours of $v \in V(G)$, then $(\phi(v_1), \phi(v_2), ..., \phi(v_k))$ is the cyclic order of the neighbours of $\phi(v) \in V(G')$. An *orientation-reversing isomorphism* from G to G' is an abstract isomorphism ϕ from G to G' which reverses the cyclic order at each vertex: if $(v_1, v_2, ..., v_k)$ is the cyclic order of the neighbours of $v \in V(G)$, then $(\phi(v_k), \phi(v_{k-1}), ..., \phi(v_1))$ is the cyclic order of the neighbours of $\phi(v) \in V(G')$.

We say that two plane graphs G and G' are isomorphic as embedded graphs if and only if there is an orientation-preserving or orientation-reversing isomorphism from G to G'. So here we treat the mirror image as equivalent. In the remainder of this chapter, by an isomorphism we will always mean an orientation-preserving or orientation-reversing isomorphism.

Clearly, if two plane graphs are isomorphic as embedded graphs, they are also isomorphic as (unembedded) graphs, but the other direction does not hold.

However, Whitney proved the following:

Theorem 4.1 (Whitney [121])**.** *Two 3-connected plane graphs G and G' are isomorphic if and only if they are isomorphic as embedded graphs.*

Since fullerenes are 3-connected, it follows from Whitney's theorem that the concept of graph isomorphism and isomorphism of embedded graphs coincide for fullerenes.

The *dual* of a fullerene is the plane graph obtained by exchanging the roles of vertices and faces: the vertex set of the dual graph is the set of faces of the original graph and two vertices in the dual graph are adjacent if and only if the two faces share an edge in the original graph. The rotational order around the vertices in the embedding of the dual fullerene follows the rotational order of the faces. The dual of a fullerene with n vertices is a *triangulation* (i.e. a plane graph where every face is a triangle) which contains 12 vertices with degree 5 and $n/2 - 10$ vertices with degree 6. The graph with black vertices and solid edges in Figure 4.1 is the fullerene C_{20} (i.e. the dodecahedron), the graph consisting of white vertices and dashed edges is its dual graph. We will also refer to the dual of a fullerene as a *dual fullerene* or a fullerene in *dual representation*.

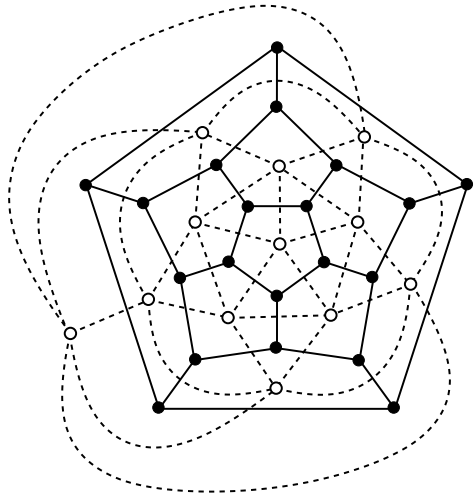


Figure 4.1: The fullerene C_{20} (drawn with black vertices and solid edges) and its dual graph (drawn with white vertices and dashed edges).

A *fullerene patch* is a connected subgraph of a fullerene where all faces except

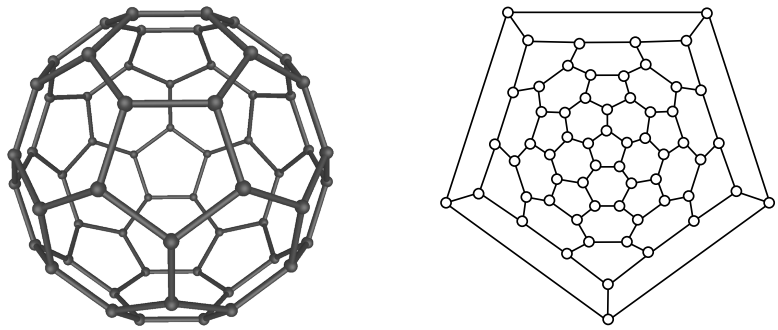
one exterior face are also faces in the fullerene. Furthermore all boundary vertices have degree 2 or 3 and all non-boundary vertices have degree 3. In the remainder of this chapter we will abbreviate *"fullerene patch"* as *"patch"*.

The *boundary* of a patch is formed by the vertices and edges which are on the unique unbounded face, i.e. the *outer face*.

Isolated Pentagon Rule (IPR) fullerenes are fullerenes where no two pentagons share an edge. IPR fullerenes are especially interesting due to a general tendency to be chemically more stable and thus they are more likely to occur in nature [113, 1].

4.1.2 Literature review

The first fullerene molecule was discovered in 1985 by Kroto et al. [80], namely the famous C_{60} buckminsterfullerene or "buckyball". It has the shape of a truncated icosahedron (which looks like a soccer ball). The (combinatorial) C_{60} fullerene is shown in Figure 4.2. In 1996 Kroto, Curl and Smalley were awarded the Nobel Prize in Chemistry for this discovery. After that discovery several attempts have been made to generate complete lists of fullerene isomers. *Isomers* have the same kind and number of atoms, but a different structure, i.e. they have the same molecular formula (e.g. C_{60}), but a different structural formula.



(a) **(b)**

Figure 4.2: The C_{60} buckminsterfullerene drawn on the sphere (a) and in the plane (b), respectively.

The first approach to generate complete lists of fullerenes was the spiral algorithm given by Manolopoulos et al. in 1991 [86]. This algorithm is briefly sketched in Section 4.4.2. The spiral algorithm was relatively inefficient and also

incomplete in the sense that not every fullerene isomer could be generated with it. Manolopoulos and Fowler [85] gave an example of a fullerene that cannot be constructed by this algorithm. Using the algorithm which is described in this chapter we were able to prove that the counterexample given by Manolopoulos and Fowler [85] is in fact the smallest possible (see Section 4.4.2).

The spiral algorithm was later modified to make it complete, but the resulting algorithm was not efficient [84]. In 1995 Yoshida and Osawa [125] proposed a different algorithm using folding nets, but its completeness has not been proven.

Other methods are described by Liu et al. [83] and Sah [109], but they also did not lead to sufficiently efficient algorithms.

The most successful approach until now dates from 1997 and is given by Brinkmann and Dress [19]. The algorithm described there is proven to be complete and has been implemented in a program called *fullgen*. The basic strategy can be described as stitching together patches which are bounded by zigzag (Petrie) paths. Unfortunately a simple (non-algorithmic) bug in the source code produced an error that occurred for the first time at 136 vertices – far too many vertices to be detectable by any of the other programs until now. Due to this error not enough fullerenes were generated by *fullgen* starting from 136 vertices. Hence the lists in the article of Brinkmann and Dress [19] contain some incorrect numbers which we will correct in Section 4.4.

The method of patch replacement can be described as replacing a finite connected region inside some fullerene with a larger patch with identical boundary. For energetic reasons, patch replacement as a chemical mechanism to grow fullerenes would need very small patches. Brinkmann et al. [20] investigated replacements of small patches and introduced two infinite families of operations. These operations can generate all fullerenes up to at least 200 vertices, but – as already shown in their paper – fail in general. In 2008 Hasheminezhad, Fleischner and McKay [65] described a recursive structure using patch replacements for the class of all fullerenes.

The carbon nanotubes [45] form a special subclass of fullerenes. These fullerenes consist of two *caps* which contain 6 pentagons each and a tube body which only consists of hexagons and which can become arbitrarily large. An example of such a nanotube is shown in Figure 4.3. Carbon nanotubes were discovered by Iijima in 1991 [69]. This subclass of fullerenes is very promising for potential applications because of their remarkable structural and electronic properties [117]. In 2002 Brinkmann et al. [36] developed a fast generator for nanotube caps.

In Section 4.2 we describe an algorithm for the efficient generation of all

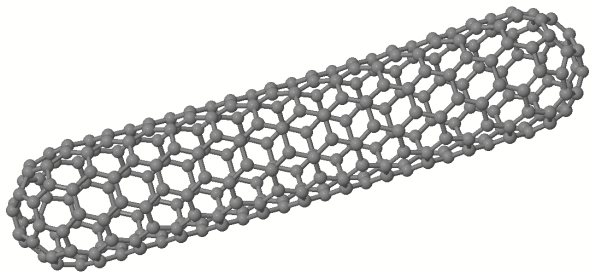


Figure 4.3: A carbon nanotube with 280 vertices.

non-isomorphic fullerenes using the construction operations from Hasheminezhad et al. [65]. In Section 4.3 we discuss the generation of IPR fullerenes. More specifically, in Section 4.3.1 we show how to extend the generation algorithm for all fullerenes to generate only IPR fullerenes by using some simple look-aheads. In Section 4.3.2 we describe a specialised generation algorithm for IPR fullerenes for which the construction stays entirely within the class of IPR fullerenes. The results obtained by these algorithms and a comparison with *fullgen* are given in Section 4.4.

4.2 Generation of fullerenes

4.2.1 The construction algorithm

We call the patch replacement operations which replace a connected fragment of a fullerene by a larger fragment *expansions* and the inverse operations *reductions*. If G' is obtained from G by an expansion, we call G' the child of G and G the parent of G'.

It follows from the results of Brinkmann et al. [32] that no finite set of patch replacement operations is sufficient to construct all fullerenes from smaller ones. So each recursive structure based on patch replacement operations must necessarily allow an infinite number of different expansions.

Hasheminezhad et al. [65] used two infinite families of expansions: L_i and $B_{i,j}$ and a single expansion F. These expansions are sketched in Figure 4.4. The lengths of the paths between the pentagons may vary and for operation L_i the mirror image must also be considered. All faces drawn completely in the figure

or labelled f_k or g_k have to be distinct. The faces labelled f_k or g_k can be either pentagons or hexagons, but when we refer to *the* pentagons of the operation, we always mean the two faces drawn as pentagons. For more details on the expansions, see the article of Hasheminezhad et al. [65].

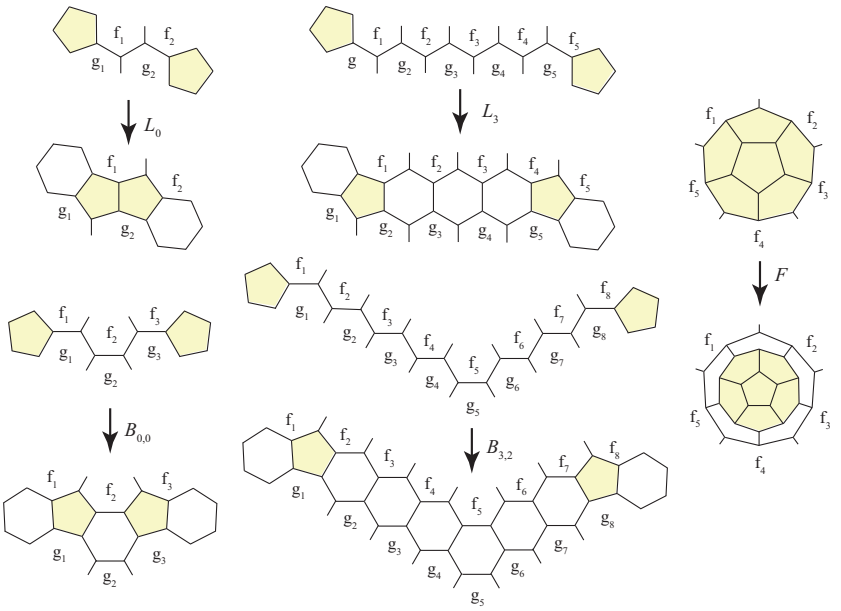


Figure 4.4: The L, B and F expansions for fullerenes.

In Figure 4.5 the L and B expansions of Figure 4.4 are shown in dual representation. We will refer to vertices which have degree $k \in \{5, 6\}$ in the dual representation of a fullerene as k-vertices. The solid white vertices in the figure are 5-vertices, the solid black vertices are 6-vertices and the dashed vertices can be either. The two 5-vertices which are involved in the reduction and the vertices which must be 6-vertices in the reduction are also called the *active* vertices of the reduction.

Three special fullerenes C_{20} (the dodecahedron), $C_{28}(T_d)$ and $C_{30}(D_{5h})$ are shown in Figure 4.6. The type-(5,0) nanotube fullerenes are those which can be made from $C_{30}(D_{5h})$ by applying expansion F zero or more times. We will refer to all fullerenes not in one of these classes as *reducible*. The following theorem which was proven by Hasheminezhad et al. [65] shows that all reducible fullerenes can be reduced using a type L or B reduction.

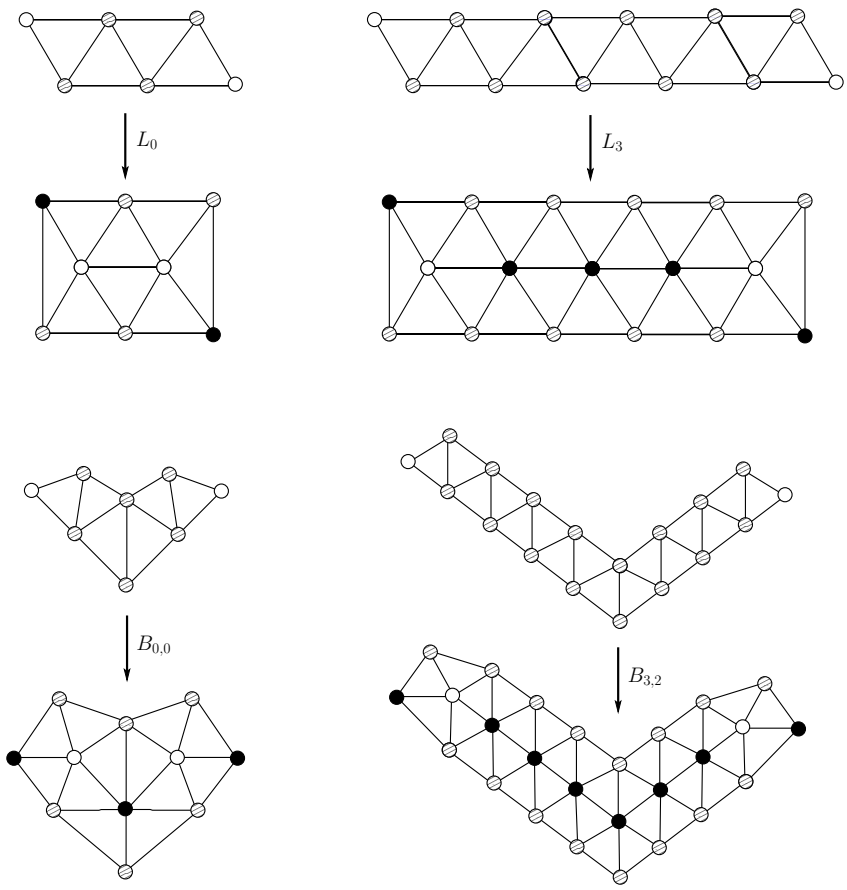


Figure 4.5: The L and B expansions in dual representation.

Theorem 4.2. *Every fullerene isomer, except $C_{28}(T_d)$ and type-(5,0) nanotube fullerenes can be constructed by recursively applying expansions of type L and B to C_{20}.*

Our algorithm uses this theorem by applying L and B expansions starting at C_{20} and $C_{28}(T_d)$, together with a separate (easy) computation of the type-(5,0) nanotube fullerenes.

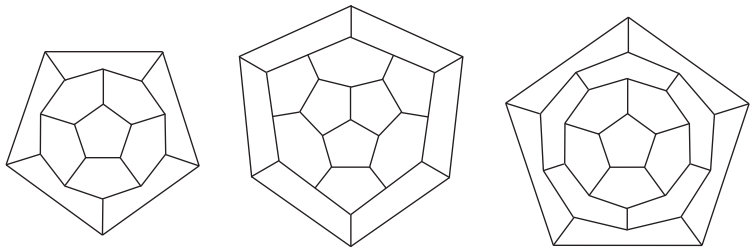


Figure 4.6: The irreducible fullerenes.

4.2.2 Isomorphism rejection

If the expansions are applied in all possible ways, a lot of isomorphic copies will be generated. Similar to the other generation algorithms described in this thesis, we only wish to output one fullerene of each equivalence class of fullerenes. We also use the canonical construction path method for this (see Section 1.3).

The two rules of the canonical construction path method applied to dual fullerenes are:

1. Only accept a dual fullerene if the last step in its construction was a canonical expansion.

2. For each dual fullerene G to which expansions are applied, only apply one expansion from each equivalence class of expansions.

So in order to use this method, we first have to define a *canonical reduction* for every reducible dual fullerene G. This reduction must be unique up to automorphisms of G. We call the dual fullerene which is obtained by applying the canonical reduction to G the *canonical parent* of G and an expansion that is the inverse of a canonical reduction in the expanded graph a *canonical expansion*.

We also define an equivalence relation on the set of all expansions or reductions of a given dual fullerene G. An expansion is completely characterized by the patch that is replaced with a larger patch. Two expansions are called *equivalent* if there is an automorphism of G mapping the two corresponding initial patches onto each other. For reductions, the definition is similar. In the

non-dual representation, the patch of a reduction also uniquely determines the reduction, but for the dual representation in addition to the patch, a rotational direction is necessary to uniquely encode a reduction of type L. This direction can be a flag describing whether the new position of the 6-vertex is in clockwise or counterclockwise position of the central path connecting the 5-vertices. In the remainder we will use the dual representation. Two type L reductions are equivalent if the patches are mapped onto each other by an orientation-preserving automorphism and the flags are the same or they are mapped onto each other by an orientation-reversing automorphism and the flags are different.

The expansions and reductions must of course be represented in an efficient way. Reductions are represented by a triple (e, x, d), where e is a directed edge that is the first edge on the central path between the two 5-vertices, x is the parameter set for the reduction (such as "(2,3)" for $B_{2,3}$) and d is a direction. For B reductions, d indicates whether the turn in the path is to the left or the right. For L reductions, d distinguishes between this reduction and its mirror image. Since e can be at either end of the path, there are two equivalent triples for the same reduction, as illustrated in Figure 4.7. We call these triples the *representing triples* of the reduction. Expansions are also represented by triples in a similar fashion.

When we translate the notion of equivalent reductions or expansions to representing triples, the equivalence relation is generated by two relations. The first is that two triples are equivalent if they represent the same reduction. The second is that (e, x, d) and (e', x', d') are equivalent if $x = x'$ and in case $d = d'$ the edge e can be mapped to e' by an orientation-preserving automorphism and in case $d \neq d'$ the edge e can be mapped to e' by an orientation-reversing automorphism.

For an efficient implementation of the canonicity criteria, it is important that in many cases simple and easily computable criteria can decide on the canonical reduction or at least reduce the list of possible reductions. To this end we assign a 6-tuple $(x_0, \ldots, x_5)$ to every triple (e, p, f) representing a possible reduction. We then choose the canonical reduction to be a reduction which has a representing triple with the smallest 6-tuple.

The values of $x_0, ..., x_4$ are combinatorial invariants of increasing discriminating power and cost. The value of x_0 is the *length* of the reduction represented by (e, p, f). The length of the reduction is the distance between the two 5-vertices of the reduction before actually applying the reduction. So in case of a $B_{x,y}$ reduction (2 parameters) it is $x + y + 2$ and in case of an L_x reduction (1 parameter) it is $x + 1$. Thus we give priority to short reductions. These are easier to detect and

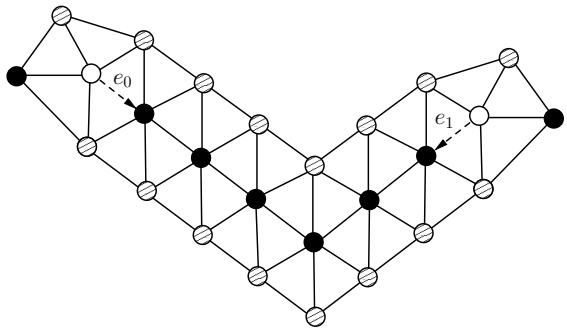


Figure 4.7: An example of two triples $(e_0, (3, 2), 1)$ and $(e_1, (2, 3), 0)$ representing the same B reduction.

allow some look-ahead. The entry x_1 is the negative of the length of the longest straight path in the reduction. For an L reduction, the value of x_1 is $-x_0$, which does not distinguish between two reductions with the same value of x_0. For a $B_{x,y}$ reduction it is $-\max\{x, y\} - 1$, which sometimes distinguishes between B reductions with the same value of x_0 and always distinguishes between an L and a B reduction with the same x_0.

The entries x_2, x_3 and x_4 are strings which contain the degrees of the vertices in well-defined neighbourhoods of the edge in the triple. These neighbourhoods are of increasing (constant) size.

In each case the value x_i is only computed for those representing triples that have the smallest value of $(x_0, ..., x_{i-1})$. As our main interest is whether an expansion we applied is canonical, we can also stop as soon as we have found a smaller 6-tuple, which may just mean a reduction with smaller value of x_0. In case of a unique triple with minimal value of $(x_0, ..., x_{i-1})$ or two such triples representing the same reduction, we have found the canonical reduction and can stop the computation of the remaining values.

If after the computation of $(x_0, ..., x_4)$ there is still more than one possibly canonical triple, we define x_5 as a string encoding the whole structure of the graph relative to the edge and the direction in the representing triple. We could use *nauty* [88] to compute a canonical labelling (like we did for the other generation algorithms which are described in this thesis), but for fullerenes there are algorithms which can compute a canonical labelling in linear time. Therefore we use such a specialised algorithm to compute the string x_5. The details of this string can be found in the article of Brinkmann and McKay [33], but it can in

short be described as the code of a BFS-numbering starting at the edge of the representing triple and evaluating the neighbours of a vertex in the rotational order (clockwise/counterclockwise) given by the direction of the representing triple. Two triples coding patches in two graphs (that may be identical or not) containing the same directions are assigned the same value x_5 if and only if there is an orientation-preserving isomorphism of the graphs mapping the edges in the triples onto each other. In case of different directions, the same value of x_5 is assigned if and only if there is such an orientation-reversing automorphism. This final value x_5 makes sure that two patches (in the same or different graphs) with the same value of $(x_0, ..., x_5)$ can be mapped onto each other by an isomorphism $\phi()$ of the graphs. When performing the corresponding reductions, the patches are replaced by smaller patches and replacing the images $\phi(v)$ of vertices inside the patch appropriately, one gets an isomorphism of the reduced graphs that maps the reduced patches onto each other.

When x_5 is computed and the graph G that is tested for canonicity is accepted, as a byproduct we also have the automorphism group of G. For a given triple, each of $x_0, ..., x_4$ can be computed in constant time and x_5 can be computed in linear time, so the canonicity test can be done in linear time.

Even though it is a nice feature that deciding canonicity of a given set of possible reductions can be done in linear time, for practical performance it is more important that computing the combinatorial invariants $(x_0, ..., x_4)$ for a given reduction is of a small constant cost. For dual fullerenes with 152 vertices (fullerenes with $2.(152-2) = 300$ vertices), the discriminating power of $(x_0, ..., x_4)$ is enough to decide whether or not the last expansion was canonical in more than 99.9% of the cases.

In some cases these cheap invariants also allow look-aheads for deciding whether or not an expansion can be canonical before actually performing it. When making the lists of possible expansions, we can often already tell that a certain expansion cannot be canonical since it will not destroy all shorter reductions or since there will still be a reduction of the same length but with a smaller value of x_2. This avoids the application of a lot of non-canonical expansions. Counting only expansions passing this look-ahead, for dual fullerenes with 152 vertices still in 95.6% of the expansions a final decision can be found by only computing $(x_0, ..., x_4)$.

If there is only one representing triple with minimal value of $(x_0, ..., x_i)$ $(i \leq 4)$, the automorphism group of G is trivial, so no extra computations are necessary. For dual fullerenes with 152 vertices, there is only one representing triple with minimal value of $(x_0, ..., x_i)$ $(i \leq 4)$ in 80.9% of the cases. The ratio is decreasing

with the number of vertices: for 102 vertices of the dual fullerene it is 93.3% and for 127 vertices it is 86.9%.

Theorem 4.3. *Assume that exactly one representative of each isomorphism class of dual fullerenes up to $n-2$ vertices is given. Suppose we perform the following steps (cf. Algorithm 1.1 from Section 1.3):*

1. *Perform one expansion of each equivalence class of L and B expansions which lead to a dual fullerene with n vertices.*
2. *Accept each new dual fullerene if and only if a triple representing the inverse of the last expansion has the minimal value of $(x_0, ..., x_5)$ among all possible reductions.*

Then exactly one representative of each isomorphism class of reducible dual fullerenes with n vertices is accepted.

Proof. Let G be a reducible dual fullerene with n vertices. Since G is reducible, there is at least one reduction, thus also a canonical reduction ρ, that applies to G. The graph resulting from ρ is isomorphic to a graph in the input set, which has an expansion which is equivalent to the inverse of ρ. But this expansion produces a graph isomorphic to G and the parameters of its inverse reduction are the same as those of ρ, so the result of the expansion is accepted.

This implies that at least one representative of each isomorphism class in question is generated. It remains to be shown that at most one is generated.

Suppose that the algorithm accepts two isomorphic fullerenes G and G' with n vertices. As they are isomorphic, the canonical reductions have the same parameter set $(x_0, ..., x_5)$. As they were both accepted, they were constructed by a canonical expansion, so – as mentioned before – the two parents G_0 and G'_0 are isomorphic and there is an isomorphism that maps the corresponding expansions onto each other. By our assumption this means that G_0 and G'_0 are identical and that the two expansions are equivalent, but our procedure only performs one expansion of each equivalence class of expansions (i.e. step 1). So at most one representative of each isomorphism class of reducible dual fullerenes with n vertices is accepted. □

By recursively applying expansion F to C_{20}, all type-(5,0) fullerenes are constructed. As this constructs all type-(5,0) fullerenes exactly once and these fullerenes cannot be constructed by L or B expansions, this completes the algorithm.

4.2.3 Optimisations

It turned out that most fullerenes contain short reductions. As we give priority to short reductions, the overwhelming majority of long expansions are not canonical. For example 80.5% of the dual fullerenes with 152 vertices have an L_0 reduction and for 19.3% of the dual fullerenes with 152 vertices the shortest reduction has length 2. For efficiency reasons it is interesting to determine an upper bound on the length of a canonical expansion.

Lemma 4.4. *Reducible dual fullerenes which contain adjacent 5-vertices have an L_0, L_1 or $B_{0,0}$ reduction.*

Proof. For a proof, see the article of Hasheminezhad et al. [65]. □

So each reducible dual non-IPR fullerene has a reduction with length at most 2. In dual IPR fullerenes the shortest reduction is a reduction with the same length as the minimum distance of two 5-vertices in the dual fullerene.

Consider a 5-vertex with k rings of 6-vertices. The first ring contains 5 6-vertices, the second one 10 and the k-th ring contains $5k$ 6-vertices. So a 5-vertex with k rings of 6-vertices contains $f(k) = 1+\frac{5}{2}(k+1)k$ vertices. Figure 4.8 depicts a 5-vertex with 2 rings of 6-vertices. In dual fullerenes where the shortest distance between two 5-vertices is at least d, the sets of vertices at distance at most $\lfloor\frac{d-1}{2}\rfloor$ of different vertices are disjoint. This gives us a lower bound of $12\,f(\lfloor\frac{d-1}{2}\rfloor)$ for the number of vertices in the dual fullerene.

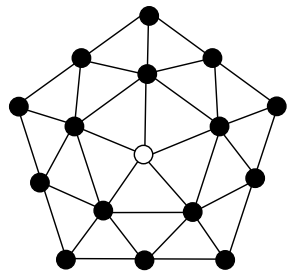


Figure 4.8: A 5-vertex with 2 rings of 6-vertices.

So expansions of length d cannot be canonical if the expanded graph contains fewer than $12\,f(\lfloor\frac{d-1}{2}\rfloor)$ vertices. This result does not only help to avoid the application of non-canonical expansions, but also avoids the need to search for long expansions.

We can often determine even sharper upper bounds for the maximum length of a canonical expansion:

Lemma 4.5. *If a dual fullerene G has a reduction of length $d \leq 2$, all children G' of G have a reduction of length at most $d + 2$.*

Proof. If G' is not IPR, this follows from Lemma 4.4, so assume that G' is IPR. The length of the shortest reduction is then the shortest distance between two 5-vertices. Let us look at the shortest path W between two 5-vertices allowing a reduction of length d in G.

As $d \leq 2$ and as all vertices in the patch P used for expansion must be distinct, W can contain at most 2 maximal subpaths entering P and ending there, starting in P and leaving it or crossing P.

The distance between a 5-vertex in P from vertices on the boundary grows at most by 1. The same is true for each pair of vertices on the boundary. So the path W can grow in two places by at most 1, proving the result. Figure 4.9 shows an example of an L expansion which can turn a reduction of length 2 into a reduction of length 4. □

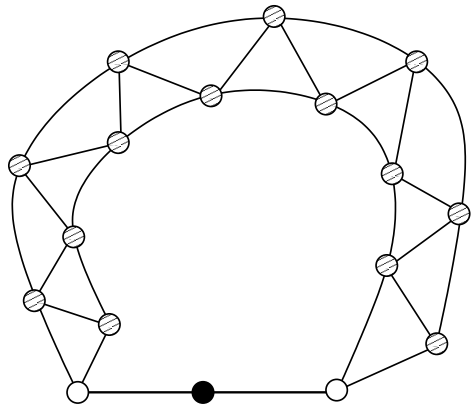


Figure 4.9: L expansion which can turn a reduction of length 2 into a reduction of length 4. The solid white vertices are 5-vertices, the solid black vertices are 6-vertices and the dashed ones can be either.

This lemma could be proven for larger d if one required the child to be canonical, but as $12\, f(\frac{5-1}{2}) = 192$, all dual fullerenes with less than 192 vertices (or

fullerenes with 380 vertices) have a reduction of length at most 4. Therefore, even for $d = 2$, Lemma 4.5 is only useful for fullerenes with at least 380 vertices.

We state the following theorem in terms of fullerenes instead of dual fullerenes since the proof relies on results which were proven for fullerenes (so this avoids translating these results into results for dual fullerenes).

Lemma 4.6. *If a fullerene G has an L_0 reduction, all canonical children G' of G have a reduction of length at most 2.*

Proof. If G' is not IPR, this follows from Lemma 4.4, so assume that G' is IPR. By Lemma 4.5, G' has a reduction of length at most 3, so a canonical child was constructed by an expansion of length at most 3. If G' was constructed by an L_0, L_1 or $B_{0,0}$ expansion, the statement follows immediately.

Figure 4.10 and Figure 4.11 show the only ways that an L_2 (respectively $B_{1,0}$) expansion can destroy an L_0 reduction which involves two pentagons p_1 and p_2 such that the expanded fullerene G' contains no reduction of length shorter than 3. The faces f_i and g_i ($1 \leq i \leq 4$) which are on the boundary of the L_2 or $B_{1,0}$ expansion have to be hexagons otherwise the dual of G' would contain 5-vertices which are at distance at most 2. Since p_1 and p_2 are involved in the L_0 reduction, they must share an edge. So there is an edge $a \in \{e_1, e_2, e_3\}$ which is equal to an edge $b \in \{e_4, e_5, e_6\}$ and as the pentagons share an edge, they must also share two faces each containing an endpoint of this common edge.

In Figure 4.12 the case of Figure 4.10 where $a = e_2$ and $b = e_5$ is shown. This patch contains a non-trivial cyclic 5-edge-cut. Also for all other possible choices of a and b this implies that a fullerene containing a patch from Figure 4.10 or Figure 4.11 must have a cyclic 4-edge-cut or a cyclic 5-edge-cut. However it follows from the results of Bornhöft et al. [7] that fullerenes are cyclically 5-edge-connected, so cyclic 4-edge-cuts do not exist. Kardoš and Škrekovski [76] showed that the type-(5,0) nanotubes are the only fullerenes which have a non-trivial 5-edge-cut.

So there is no expansion which can be applied to G such that the shortest reduction of the expanded fullerene has length 3. Thus all canonical children of G have a reduction of length at most 2. □

For the next lemmas the following observation is useful:

Observation 4.7. *If the set of vertices contained in the initial patch of an expansion of length l contains at least three 5-vertices (so in addition to the two 5-vertices of the expansion there is at least one more 5-vertex in the boundary), then in the extended patch there are two 5-vertices at distance at most $l/2 + 1$.*

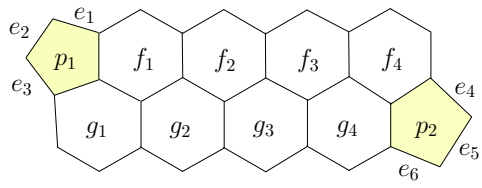


Figure 4.10: The initial patch of an L_2 expansion involving two neighbouring pentagons p_1 and p_2. One of the edges from $\{e_1, e_2, e_3\}$ is equal to an edge in $\{e_4, e_5, e_6\}$.

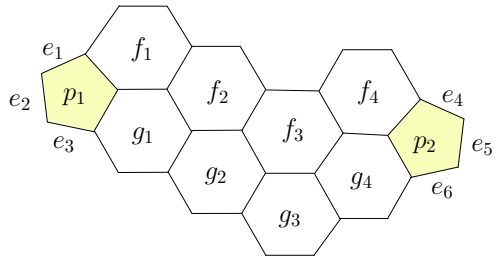


Figure 4.11: The initial patch of a $B_{1,0}$ expansion involving two neighbouring pentagons p_1 and p_2. One of the edges from $\{e_1, e_2, e_3\}$ is equal to an edge in $\{e_4, e_5, e_6\}$.

Lemma 4.8. *If a dual fullerene G has at least two reductions of length 2 which do not have the same set of 5-vertices of the reduction, all canonical children G' have a reduction of length at most 3.*

Proof. If G' is not a dual IPR fullerene, the result follows immediately, so assume the opposite. In order to prove the lemma, we have to find a bound for the minimum distance of two 5-vertices. By Lemma 4.5 each child has a reduction of length 4. So each canonical child was constructed by an expansion of length at most 4. If there were three 5-vertices in the initial patch of the expansion, the result follows from Observation 4.7. So assume this is not the case and one 5-vertex of a reduction of length 2 is not contained in the initial patch. However then the distance to the other 5-vertex in the reduction can grow by at most 1, proving the lemma. □

Lemma 4.9. *If a dual fullerene G has at least three reductions of length 2 with pairwise disjoint sets of 5-vertices of the reduction, all canonical children G' of*

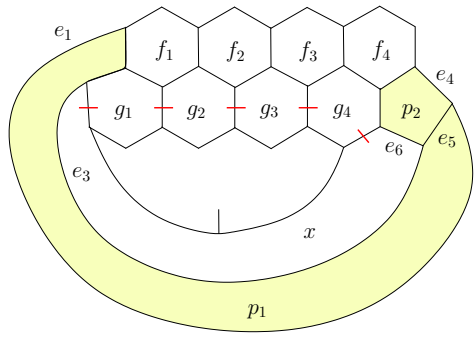


Figure 4.12: The initial patch of an L_2 expansion involving two neighbouring pentagons p_1 and p_2. This is the case of Figure 4.10 where $e_2 = e_5$. Since p_1 and p_2 share an edge, they must also share a face x. However this patch has a non-trivial cyclic 5-edge-cut (which is drawn in red), so it cannot be part of a reducible fullerene.

G have a reduction of length at most 2.

Proof. We may again assume that G' is IPR. It follows from Lemma 4.8 that G' has a reduction of length at most 3, so each canonical expansion has length at most 3. If there are three 5-vertices in the initial patch of the expansion, the result follows directly from Observation 4.7. So there is (at least) one reduction of length 2 so that none of its 5-vertices is contained in that initial patch. But then the path of length 2 between these 5-vertices still exists in the expanded graph and allows a reduction of length 2. □

Note that Lemma 4.9 does not hold if the sets of 5-vertices of the reductions of length 2 are not pairwise disjoint. This is illustrated in Figure 4.13 where the L_2 expansion from p_1 to p_2 destroys 4 reductions of length 2.

For two reductions R_1 and R_2 in a dual fullerene G we define the distance $d(R_1, R_2)$ to be $\min\{d(a_1, a_2) \mid a_i \text{ is a 5-vertex of } R_i\}$.

Lemma 4.10. *If a dual fullerene G has L_0 reductions R_1 and R_2 with $d(R_1, R_2) > 4$, all canonical children G' of G have an L_0 reduction.*

Proof. It follows from Lemma 4.6 that there is a reduction of length at most 2 in G'. The distance between vertices which are in the initial patch of an expansion of length 2 is at most 4. Therefore at least one of the two neighbouring 5-vertex pairs was not modified by the expansion and the neighbouring vertices are either

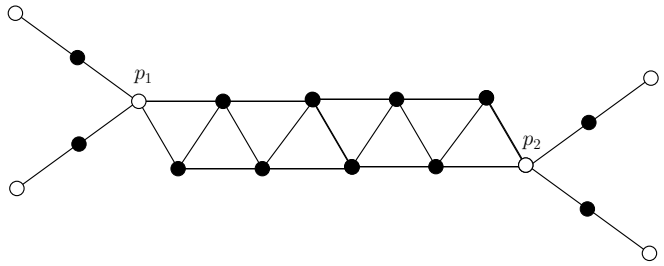


Figure 4.13: An L_2 expansion which destroys 4 reductions of length 2. The white vertices are 5-vertices and the black ones are 6-vertices.

unchanged or changed to 6-vertices. In either case the reduction will still be possible. □

For dual fullerenes with 152 vertices, Lemmas 4.5, 4.6, 4.8, 4.9 and 4.10 can be used to determine a bound on the length of canonical expansions in 93.9% of the cases.

In Section 4.4 we compare the performance of our algorithm with *fullgen*.

4.3 Generation of IPR fullerenes

Recall from Section 4.1.1 that IPR fullerenes are fullerenes where no two pentagons share an edge (or when using the dual representation: dual IPR fullerenes are dual fullerenes with no adjacent 5-vertices). IPR fullerenes tend to be chemically more stable and thus more likely to occur in nature [113, 1]. Therefore we also developed specialised algorithms for the generation of IPR fullerenes. In Section 4.3.1 we extend the generation algorithm for all fullerenes with some simple look-aheads and a filter to generate only IPR fullerenes. In Section 4.3.2 we describe a specialised generation algorithm for IPR fullerenes which recursively generates IPR fullerenes from smaller IPR fullerenes. The results obtained by these algorithms and a comparison with *fullgen* are given in Section 4.4.

4.3.1 Generator for all fullerenes with filter and look-aheads

The generation algorithm from Section 4.2 was developed for generating all fullerenes, but it can also be used to generate only IPR fullerenes by using a filter and some simple look-aheads:

When generating dual IPR fullerenes with n vertices one should obviously not perform expansions which lead to dual fullerenes with n vertices which contain adjacent 5-vertices. Figure 4.14 shows some examples of expansions which can lead to dual IPR fullerenes (cf. Figure 4.4 from Section 4.2). The solid white vertices are 5-vertices, the solid black vertices are 6-vertices and the dashed ones can be either. If any of the black vertices in the initial patch of the expansion would be a 5-vertex, the expanded dual fullerene would not be IPR.

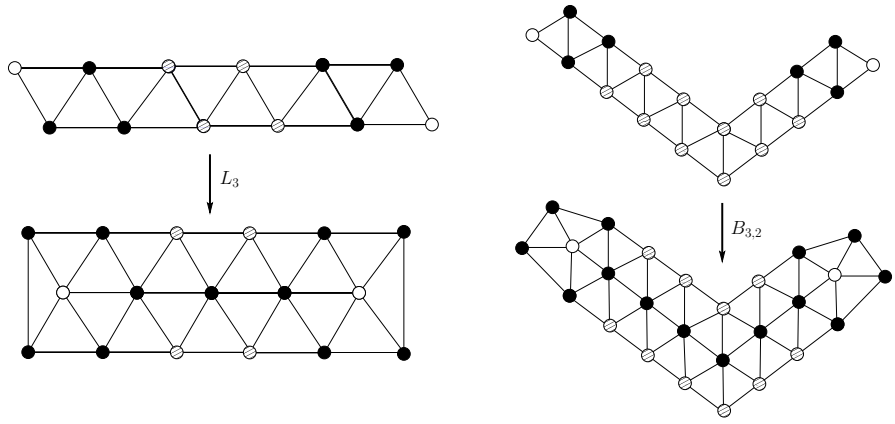


Figure 4.14: Examples of expansions which can lead to dual IPR fullerenes.

An L_0 expansion is the only expansion that increases the number of vertices in a dual fullerene by just 2 vertices, but the result of an L_0 expansion is never a dual IPR fullerene. So when constructing dual IPR fullerenes with n vertices, dual IPR fullerenes with $n-2$ vertices do not have to be constructed. The largest dual fullerenes to which an expansion is applied have $n-3$ vertices.

For a dual fullerene with $n-4$ vertices only expansions of length 3 (i.e. L_2 or $B_{1,0}$ expansions) can lead to dual IPR fullerenes with n vertices. However if a dual fullerene with $n-4$ vertices contains an L_0 reduction, it follows from Lemma 4.6 that expansions of length 3 are not canonical. Thus we can reject all dual fullerenes with $n-4$ vertices that contain an L_0 reduction and also avoid applying L_0 expansions to dual fullerenes with $n-6$ vertices.

As can be seen in Table 4.3 from Section 4.4, these simple look-aheads already result in a relatively efficient program (especially for fullerenes with a large number of vertices or when generating fullerenes for a range of vertices). More details can be found in Section 4.4.

4.3.2 Recursive generation of IPR fullerenes

We also developed a specialised algorithm for the recursive generation of IPR fullerenes. This generation algorithm stays entirely within the class of IPR fullerenes, that is: IPR fullerenes are constructed from smaller IPR fullerenes.

The construction operations are the same as the operations for the generation of all dual fullerenes from Section 4.2, but here only expansions for which the expanded dual fullerene is also IPR are performed.

We first classify the class of *irreducible* IPR fullerenes. These are IPR fullerenes which cannot be reduced to a smaller IPR fullerene by our reduction operations. We will show that this class consists of 36 fullerenes with up to 112 vertices and 4 infinite families of nanotube fullerenes. The generation then recursively applies the expansions to the irreducible IPR fullerenes using an adapted version for IPR fullerenes of the isomorphism rejection method from Section 4.2.2.

Definitions and preliminaries

Definition 4.1 (Cluster)**.** *A k-cluster C is a plane graph where all faces except one exterior face are triangles and has the following properties:*

- *All vertices of C have degree at most 6.*
- *Vertices which are not on the boundary of C have degree 5 or 6.*
- *C contains exactly k vertices with degree 5 which are not on the boundary.*
- *No two vertices with degree 5 which are not on the boundary are adjacent.*
- *Vertices with degree 5 which are not on the boundary are at distance at least 2 from the boundary.*
- *Between any two vertices a, b of C which have degree 5 and which are not on the boundary, there is a path P from a to b so that each edge on P contains exactly one vertex with degree 5 which is not in the boundary.*
- *No subgraph of C is a k-cluster.*

A k-cluster for which k is not specified is sometimes just called a *cluster*. We also assign a colour to the vertices of a cluster: vertices which are on the boundary of the cluster have colour 6 and the colour of the vertices which are not on the boundary is equal to their degree. We also call a vertex with colour 5 a 5-vertex and a vertex with colour 6 a 6-vertex.

We say that a dual fullerene G *contains* a cluster C if and only if C is a subgraph of G and every vertex on the boundary of C has degree 6 in G.

Definition 4.2 (Locally reducible cluster)**.** *A cluster is locally reducible if there exists an L or B-reduction where the active vertices of the reduction are part of the cluster such that the reduced cluster does not contain any adjacent 5-vertices.*

Note that the reduced cluster is not necessarily a cluster. Clusters which are not locally reducible are called *irreducible*.

Lemmas 4.11 and 4.12 are useful for the proof of Lemma 4.13.

Lemma 4.11. *Given a dual fullerene G and a reduction. If $v, w \in V(G)$ are at distance d in G and v nor w are active vertices of the reduction, then v and w are at distance at least $d - \lfloor \frac{d+1}{3} \rfloor$ in the reduced dual fullerene.*

Proof. Each time the reduction crosses the shortest path P between v and w, the distance between v and w decreases by at most one. All vertices involved in a reduction of a dual fullerene have to be distinct. Thus each time the reduction crosses P, 3 vertices of P become part of the reduction patch as can be seen in Figure 4.15. So if the reduction crosses P k times, P must contain at least $3k$ vertices. So a reduction can cross, enter or leave a path of length d at most $\lfloor \frac{d+1}{3} \rfloor$ times. An example for $d = 5$ is given in Figure 4.15. □

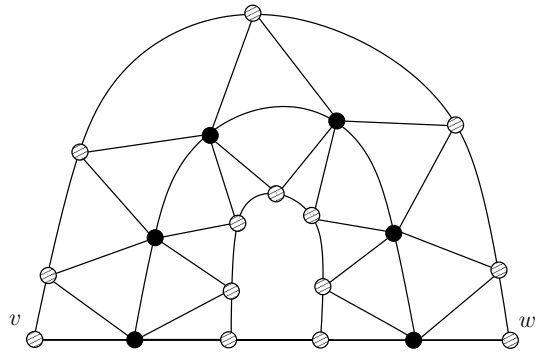


Figure 4.15: Example for Lemma 4.11 with $d = 5$. The black vertices are 6-vertices and the dashed vertices can be either 5- or 6-vertices. Here the reduction crosses the shortest path between v and w two times.

Lemma 4.12. *Given a dual fullerene G and a reduction. If $v, w \in V(G)$ are at distance d in G and v is a 6-vertex which becomes a 5-vertex after reduction and w is not an active vertex of the reduction, then v and w are at distance at least $d - \lfloor \frac{d}{3} \rfloor$ in the reduced dual fullerene.*

Proof. Each time the reduction crosses the shortest path between v and w the distance between v and w decreases by at most one. Since all vertices involved in a reduction of a dual fullerene have to be distinct and since v is involved in the reduction, it can cross a path of length d from v to w at most $\lfloor \frac{d}{3} \rfloor$ times. An example for $d = 3$ is given in Figure 4.16. □

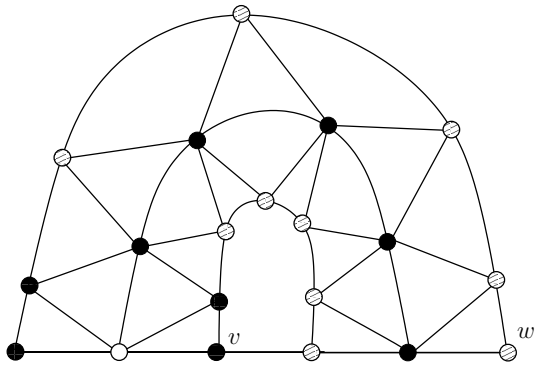


Figure 4.16: Example for Lemma 4.12 with $d = 3$. The white vertices are 5-vertices, the black vertices are 6-vertices and the dashed vertices can be either. Here the reduction crosses the shortest path between v and w once.

Lemma 4.13. *A dual IPR fullerene which contains a locally reducible cluster is reducible to a smaller dual IPR fullerene.*

Proof. Given a dual IPR fullerene G which contains a locally reducible cluster C. Let G' be the dual fullerene obtained by applying a reduction from C. The only possibility such that G' would not be IPR is that a 5-vertex which is part of C or a 6-vertex of C which becomes a 5-vertex after reduction would be adjacent to a 5-vertex which is not part of the cluster.

Let v be a 5-vertex of G which is not part of C. It follows from Definition 4.1 that 5-vertices which are not part of the cluster, are at distance at least 3 from 5-vertices which are part of the cluster.

Let w be a 5-vertex which is in C and which is not an active vertex of the reduction. It follows from Lemma 4.11 that v and w are at distance at least 2 in G'.

Now let w be a 6-vertex which becomes a 5-vertex after reduction. Since w is adjacent to a 5-vertex in C, it follows from Definition 4.1 that v and w are at distance at least 2 in G. Thus it follows from Lemma 4.12 that v and w are at distance at least 2 in G'.

Thus G' does not contain any adjacent 5-vertices. □

Note that if a dual fullerene contains multiple clusters, they are distinct in the sense that for every two clusters in a dual fullerene the set of 5-vertices is disjoint, but they may have some 6-vertices in common.

Generation of clusters

In this section we describe an algorithm to generate all k-clusters ($k > 0$). A vertex of a cluster-tree T_C of a cluster C is a k-vertex if it is a k-vertex in C.

Definition 4.3 (Cluster-tree)**.** *A cluster-tree T_C of a k-cluster C is a subgraph of C with the following properties:*

- *T_C is an unembedded tree.*
- *T_C contains k 5-vertices.*
- *Every $e \in E(T_C)$ contains exactly one 5-vertex.*
- *All vertices with degree 1 in T_C are 5-vertices*

Every k-cluster C contains a cluster-tree T_C, since such a cluster-tree can be obtained from C as follows:

- Let G be the graph consisting of all paths between any two 5-vertices of C such that each edge $e \in E(G)$ contains exactly one 5-vertex.
- Let S_G be a spanning tree of G.
- Remove all 6-vertices with degree 1 from S_G.

Cluster-trees are constructed by recursively applying construction operations I and II from Figure 4.17 to a single 5-vertex. The white vertices represent 5-vertices and the black ones are 6-vertices. These colour codes are also used in the

other figures in the remainder of this chapter. The operations are only applied to 5-vertices (respectively 6-vertices) which have degree less than 5 (respectively 6).

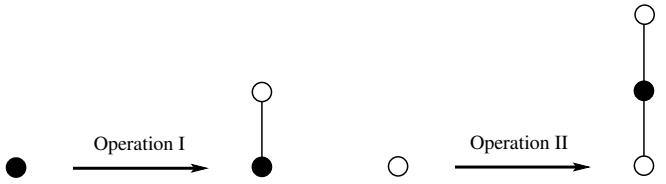


Figure 4.17: The construction operations for cluster-trees.

Lemma 4.14. *Every cluster-tree can be generated by recursively applying operations I and II to a single 5-vertex.*

Proof. We will prove this by showing that every cluster-tree can be reduced to a smaller cluster-tree by the inverse of the construction operations.

Given a cluster-tree T. Assume that $|V(T)| > 1$ as the tree with one vertex is clearly constructed. Take an arbitrary degree 1 vertex of T and call it v. It follows from Definition 4.3 that v is a 5-vertex and that its only neighbour w is a 6-vertex, so the degree of w is at least 2. If w has degree 2, the inverse of operation II can be applied and if its degree is larger than 2, the inverse of operation I can be applied. So T is reducible to a smaller cluster-tree. □

If 6-vertices have degree less than 6 in a cluster, we represent the remaining incident edges of those 6-vertices by semiedges. We will describe the generation algorithm for clusters in terms of the reduction. The reduction operations are shown in Figure 4.18 and are performed on embedded graphs. Vertices which can be either 5- or 6-vertices are drawn as dashed white vertices and the small white triangles represent one or more edges. The reduction procedure of a cluster C is as follows: every cluster contains a cluster-tree, so choose an arbitrary cluster-tree T_C of C and then recursively apply reduction operations (a) and (b). When the graph can no longer be reduced, all semiedges are removed. Reduction operation (a) is only applied if the edge e which is replaced by 2 semiedges is not in $E(T_C)$ and if it is part of the boundary of the current plane graph. Reduction (a) replaces an edge of a triangle by 2 semiedges. The vertices shown in reduction (a) from Figure 4.18 can be 5-vertices or 6-vertices, but the graph should not contain adjacent 5-vertices (so a triangle can contain at most one 5-vertex). Reduction (b) replaces a 6-vertex with 5 semiedges by a semiedge.

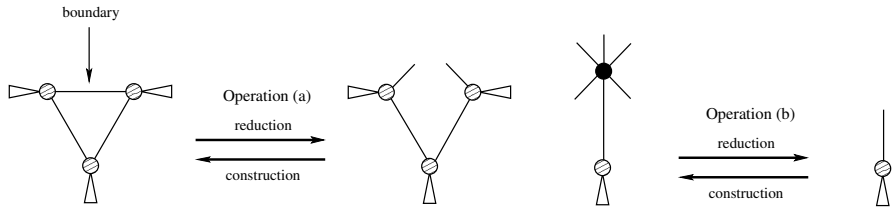


Figure 4.18: The operations for clusters.

Lemma 4.15. *The reduction procedure reduces C to T_C.*

Proof. The reduction procedure recursively applies operations (a) and (b) to C until the graph can no longer be reduced. We then remove all semiedges and call this irreducible graph G. We will show that G is T_C.

Since the reduction does not remove edges $e \in E(T_C)$, T_C is a subgraph of G.

If G contains cycles, it contains triangles since operation (a) only reduces edges which are part of the boundary. But if G contains triangles, it can be reduced by operation (a). Thus G does not contain cycles.

Let $V' = V(G) \setminus V(T_C)$. All vertices $v \in V'$ are 6-vertices and G does not contain 6-vertices of degree 1, else it would be reducible by operation (b). Since G and T_C are trees we have:

$$\begin{aligned} |E(G)| &= |V(G)| - 1 \\ &= |V(T_C)| + |V'| - 1 \\ &= |E(T_C)| + |V'| \end{aligned} \tag{4.1}$$

Let $E' = E(G) \setminus E(T_C)$. If $|V'| > 0$ and since all vertices in V' have degree at least 2, $|E'|$ is minimal if all vertices in V' have degree 2, so $|E'|$ is at least $\frac{2|V'|+2}{2}$. Thus we have: $|E(G)| \geq |E(T_C)| + \frac{2|V'|+2}{2} = |E(T_C)| + |V'| + 1$. But this is a contradiction to equation 4.1, so V' has to be emty and thus $V(G) = V(T_C)$.

We also have $|V(G)| = |E(G)| + 1 = |E(T_C)| + |E'| + 1$. Since $|V(G)| = |V(T_C)|$, we have $|V(T_C)| = |E(T_C)| + |E'| + 1$. So $|E'| = 0$ since T_C is a tree. Thus G is T_C. □

The construction algorithm for k-clusters is the inverse of the reduction procedure and is as follows:

- Generate all non-isomorphic k-cluster-trees.

- Given a k-cluster-tree T. Add semiedges to T such that every 5-vertex (respectively 6-vertex) of T has degree 5 (respectively degree 6).
- Embed T in all possible ways.
- Given an embedding of T. Recursively apply the inverse of reduction operations (a) and (b) from Figure 4.18 to T. The inverse of reduction operation (b) is only applied to vertices at distance at most 1 from a 5-vertex, such that the new 6-vertex is at distance at most 2 from a 5-vertex.

Since there are only a small number of k-cluster-trees, we use isomorphism rejection by lists to avoid isomorphic copies (see Section 1.3). More specifically we compute a canonical labelling of each generated cluster-tree and keep a list of the canonical labellings of the cluster-trees which were generated so far. The recursion is only continued for cluster-trees which were not generated before.

The construction algorithm only outputs clusters which do not contain adjacent 5-vertices. The generated k-clusters are then tested for local reducibility and only the irreducible ones are output. Since there are only few irreducible clusters, we apply isomorphism rejection by lists to avoid isomorphic copies.

Irreducibility of k-clusters $(1 \leq k \leq 6)$

Lemma 4.16. *All dual IPR fullerenes which contain only 1-clusters are reducible to a smaller dual IPR fullerene.*

Proof. In [65] it was proven that in a dual IPR fullerene, the shortest path between two 5-vertices forms a valid L or B-reduction (not necessarily to a dual IPR fullerene). Each cluster contains one 5-vertex, thus all vertices at distance at most 2 from each 5-vertex are 6-vertices.

Given a dual IPR fullerene G. Let G' be the graph obtained by applying the shortest reduction between two 5-vertices $a, b \in V(G)$. Let a' (respectively b') be the 6-vertex in G which is adjacent to a (respectively b) which is transformed into a 5-vertex by the reduction. It follows from Lemma 4.11 that the distance in G' between 5-vertices which were not involved in the reduction is at least 2. It follows from Lemma 4.12 that the distance in G' between a' (or b') and a 5-vertex which is not modified by the reduction is at least 2.

Suppose a and b are at distance d in G. Note that d is at least 3 since a and b are part of a different cluster. Since we performed the shortest reduction between a and b, a' and b' are at distance at least $d-2$ in G'. If $d > 3$ there is not a problem. If $d = 3$, a' and b' could be at distance 1 in G'. However by using

a similar argument as in the proof of Lemma 4.6 from Section 4.2.3 this would imply that G has a non-trivial cyclic 5-edge-cut and is thus a type-(5,0) nanotube. But this is not possible since G is IPR. Thus G' is a dual IPR fullerene. □

Using a program based on the algorithm from Section 4.3.2 we generated all k-clusters (for a given k) and tested the generated clusters for reducibility. Later in this section, we will describe how we tested the correctness of our programs. We obtained the following results:

Observation 4.17. *All k-clusters with $k \in \{2, 3, 5\}$ are locally reducible.*

Applying Lemma 4.13 to Observation 4.17 gives us the following corollary:

Corollary 4.18. *Every dual IPR fullerene which contains a k-cluster ($k \in \{2, 3, 5\}$) is reducible to a smaller dual IPR fullerene.*

Observation 4.19. *There is exactly one 4-cluster which is not locally reducible.*

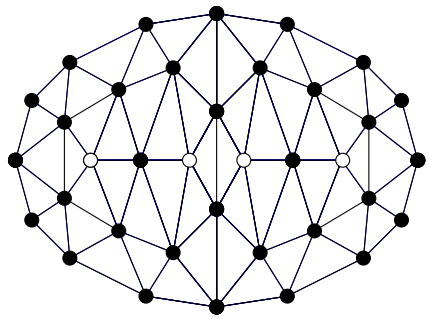


Figure 4.19: A locally irreducible 4-cluster.

This cluster is depicted in Figure 4.19. The four 5-vertices are white and the other vertices are 6-vertices. Every dual IPR fullerene which contains this cluster has a $B_{2,2}$-reduction to a smaller dual IPR fullerene unless the vertex x displayed in Figure 4.20a is a 5-vertex. The path of vertices which is going to be reduced by the $B_{2,2}$-reduction is drawn with dashed edges (assuming x is not a 5-vertex). In principle x can be a vertex which is part of the cluster, but this is not a problem for the reduction. If x is a 5-vertex, there is an L_2-reduction which yields a dual IPR fullerene. This is shown in Figure 4.20b. The reduced dual fullerene is IPR since y is a 6-vertex, otherwise the dual fullerene before reduction was not IPR. In principle y might be identical to one of the vertices which is part of the cluster. This gives us the following corollary:

Corollary 4.20. *Every dual IPR fullerene which contains a 4-cluster is reducible to a smaller dual IPR fullerene.*

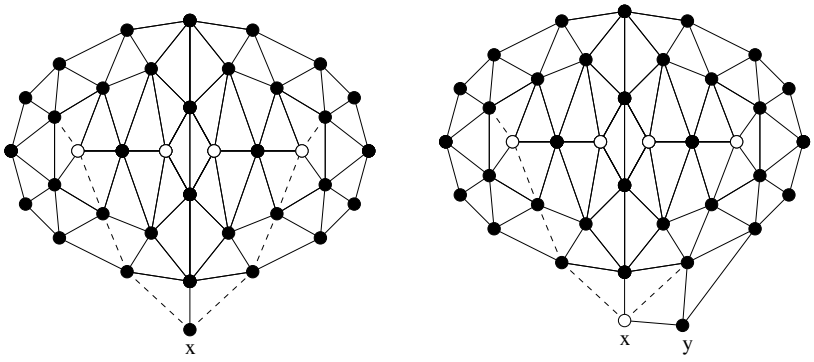


Figure 4.20: A locally irreducible 4-cluster which has a $B_{2,2}$-reduction (i.e. Figure 4.20a) or an L_2-reduction (i.e. Figure 4.20b).

Using the generator for k-clusters we also obtained the following result:

Observation 4.21. *There are exactly six 6-clusters which are not locally reducible.*

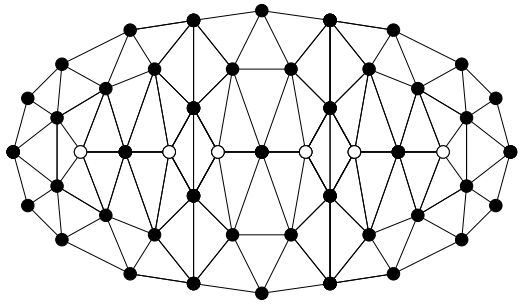


Figure 4.21: A locally irreducible 6-cluster, called *straight-cluster*.

The first cluster is depicted in Figure 4.21. The six 5-vertices are white and the other vertices are 6-vertices. We call this a *straight-cluster*. Every dual IPR fullerene which contains this cluster has an L_6-reduction to a smaller dual IPR

fullerene unless vertex a or b displayed in Figure 4.22a is a 5-vertex. This is shown in Figure 4.22a. Also here a and b may be part of the cluster. The path of vertices which is going to be reduced by the L_6-reduction is drawn with dashed edges. If a or b is a 5-vertex, there is an L_2-reduction which yields an IPR fullerene. This is shown in Figure 4.22b where it is assumed that a is a 5-vertex. The reduced dual fullerene is IPR since b is a 6-vertex, otherwise the original dual fullerene was not IPR. This gives us the following corollary:

Corollary 4.22. *Every dual IPR fullerene which contains a straight-cluster is reducible to a smaller IPR fullerene.*

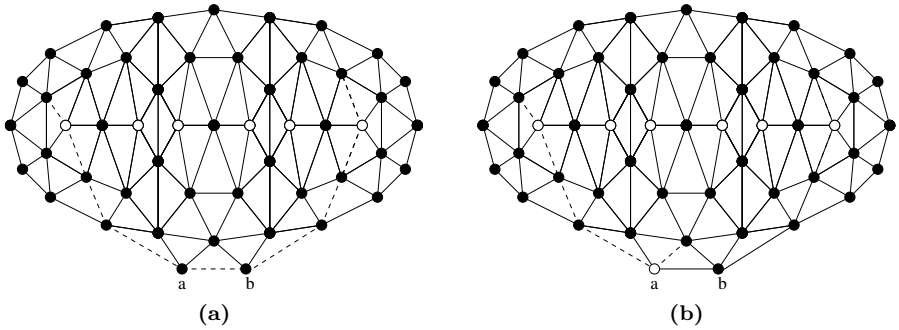


Figure 4.22: Straight-cluster which has an L_6-reduction (i.e. Figure 4.22a) or an L_2-reduction (i.e. Figure 4.22b).

We call the cyclic sequence of the degrees of the vertices in the boundary of a patch in clockwise or counterclockwise order the *boundary sequence* of a patch.

A *cap* is a fullerene patch which contains 6 pentagons and has a boundary sequence of the form $(23)^l(32)^m$. Such a boundary is represented by the parameters (l, m). In the literature, the vector (l, m) is also called the *chiral vector* (see [45]). When we speak about *caps* in the remainder of this text, we more specifically mean caps with a boundary sequence of the form $(23)^l(32)^m$. Not every patch of 6 pentagons can be completed with hexagons to a patch with a boundary sequence of the form $(23)^l(32)^m$. An example of such a patch is given in Figure 4.23a (this example comes from [74]): every way to complete this patch with hexagons leads to a conflict. Figure 4.23b gives an example of a partial filling which cannot be completed because the dashed edge (which is enforced) leads to an octagon while only hexagons are allowed. But the patches with 6 pentagons

which we will discuss in the remainder of this section all can be completed with hexagons to a boundary of the form $(23)^l(32)^m$.

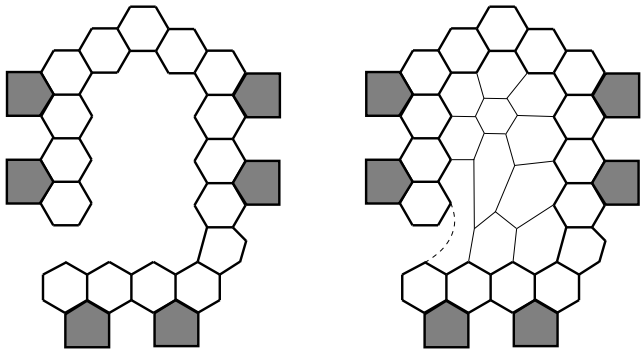


Figure 4.23: A patch with 6 pentagons which cannot be completed with hexagons to a patch with a boundary sequence of the form $(23)^l(32)^m$ (i.e. Figure 4.23a). Figure 4.23b gives an example of a partial filling which cannot be completed because there would be an octagon.

It follows from the results of Brinkmann [10] that a (fullerene) patch which contains 6 pentagons and which can be completed with hexagons to a boundary of the form $(23)^l(32)^m$ has unique boundary parameters, i.e. it cannot be completed to a boundary with parameters (l', m') where l' is different from l or m' is different from m. A cap with boundary parameters (m, l) is the mirror image of a cap with boundary (l, m). A cap has a valid reduction if and only if its mirror image is also reducible. Therefore we will assume that $l \geq m$.

The second irreducible 6-cluster is depicted in Figure 4.24. We call this a *distorted star-cluster*. By checking all possible reductions, it can be seen that for any dual IPR fullerene which contains this cluster there are no reductions to a smaller dual IPR fullerene where both 5-vertices of the reduction are in the distorted star-cluster.

Caps which contain the dual of a distorted star-cluster as a subgraph, have boundary parameters (6,5). Adding a ring of hexagons (or a ring of 6-vertices in the dual) to a cap does not change the boundary parameters of the cap (see Figure 4.25). Note that there are multiple ways of gluing together two caps with boundary parameters $(l, 0)$ to a fullerene. We call an (l, m) ring of hexagons of an IPR fullerene *reducible* if there is a way of removing that ring of hexagons

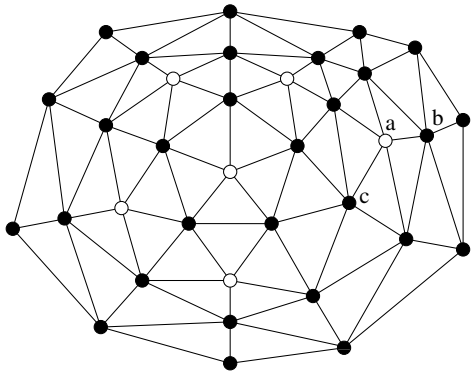


Figure 4.24: A locally irreducible 6-cluster, called *distorted star-cluster*.

such that the reduced fullerene is still IPR.

We call a cap which contains at least one pentagon in its boundary a *kernel*. Clearly, every cap has a kernel.

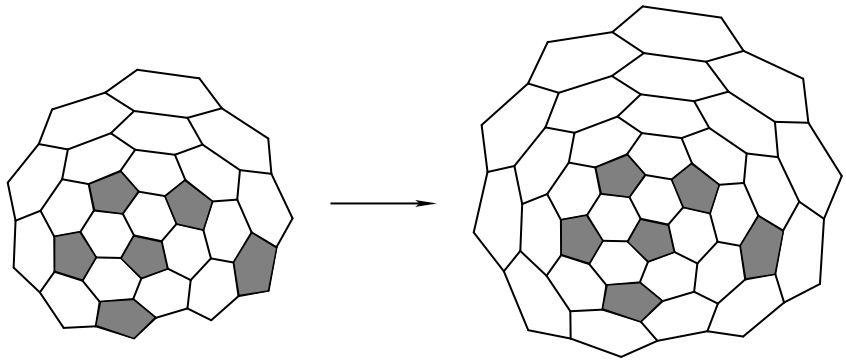


Figure 4.25: Adding a ring of hexagons to an IPR cap with boundary parameters (6,5).

The program from Brinkmann et al. described in [36] generates all nanotube caps which are non-isomorphic as infinite half-tubes. This is done by first generating all non-isomorphic nanotube caps and then filtering out the ones which are non-isomorphic as infinite half-tubes. We modified the program so it outputs all non-isomorphic nanotubes (thus also the ones which are isomorphic as infinite half-tubes). By using this modified version of the generator, we were able to gen-

erate all IPR (6,5) kernels. The largest one has 73 vertices, so an IPR fullerene which contains a (6,5) cap and has no reducible (6,5) hexagon rings has at most $2 \cdot 73 + 2 \cdot (6 + 5) = 168$ vertices. The $2 \cdot (6 + 5)$ represents a ring of hexagons, since the fullerene consisting of 2 IPR kernels may not be IPR.

Using the corrected version of *fullgen* (see Section 4.4), we determined all IPR fullerenes up to 168 vertices which have a (6,5) boundary and do not have any reducible (6,5) hexagon rings. There are 11 such fullerenes and each of them is reducible to a smaller IPR fullerene. The largest one has 106 vertices. These results have been independently confirmed by our generator for IPR fullerenes which uses a filter (see Section 4.3.1). All of the dual (6,5) caps in these 11 dual IPR fullerenes contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of the distorted star-cluster.

Consider the directed edge (a, b) from the distorted star-cluster from Figure 4.24. If a ring of 6-vertices is added to a dual (6,5) cap which contains (a, b), the straight path starting from (a, b) still exits the cap at the same relative position in the larger dual cap. Given a dual IPR fullerene F which has a (6,5) boundary. If there is an L or B-reduction which starts from (a, b) and where the second 5-vertex of the reduction is part of the other dual cap of F, then the dual fullerene F' obtained by adding a (6,5) ring of 6-vertices to F is still reducible by the same reduction (but which now has one additional 6-vertex). So if the reduction in F was an L_x reduction, it will be an L_{x+1} reduction in F'. (Note that a reduction where a is one of the 5-vertices involved in the reduction and where b is part of the reduction path can only produce a smaller dual IPR fullerene if vertex c (from Figure 4.24) is the 6-vertex which is transformed into a 5-vertex by the reduction.)

We then added (6,5) rings of 6-vertices to these 11 dual fullerenes which have a (6,5) boundary and do not have any reducible (6,5) rings of 6-vertices. When 5 rings of 6-vertices have been added, there is a reduction from (a, b) to the other dual cap in each of the 11 cases. So all dual fullerenes of these 11 types with at least 5 (6,5) rings of 6-vertices are reducible to a smaller dual IPR fullerene. We also verified that each of these 11 types of dual fullerenes with less than 5 rings of 6-vertices are reducible as well.

This gives us the following corollary:

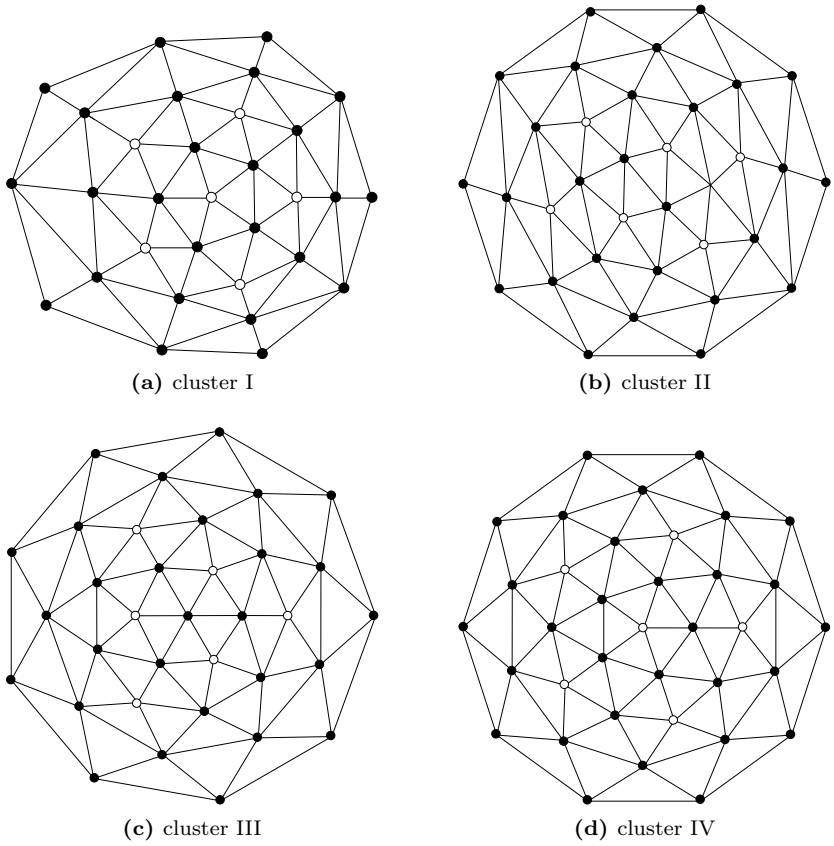
Corollary 4.23. *Every dual IPR fullerene which contains a (6,5) boundary is reducible to a smaller dual IPR fullerene.*

There is a dual (6,5) kernel which is a subgraph of the distorted star-cluster. So if a dual fullerene contains a distorted star-cluster, it also has a dual (6,5)

kernel and thus also a (6,5) boundary. This gives us:

Corollary 4.24. *Every dual IPR fullerene which contains a distorted star-cluster is reducible to a smaller dual IPR fullerene.*

The remaining 4 locally irreducible 6-clusters are depicted in Figure 4.26. We call them cluster I, II, III and IV respectively. Dual caps which contain cluster I, II, III or IV as a subgraph have boundary parameters (5,5), (8,2), (9,0) and (10,0) respectively.



(a) cluster I **(b)** cluster II

(c) cluster III **(d)** cluster IV

Figure 4.26: Four irreducible 6-clusters.

By checking all possible reductions which involve a 5-vertex which is part of one of these four clusters, it can be seen that dual IPR fullerenes which contain

one of these clusters do not have a reduction to a smaller dual IPR fullerene where at least one of the 5-vertices involved in the reduction is in one of these four clusters. We call clusters with this property *globally irreducible*. This gives us:

Corollary 4.25. *Every dual IPR fullerene which contains two 6-clusters c and d with $c, d \in \{I, II, III, IV\}$ is not reducible to a smaller dual IPR fullerene.*

Also note that dual caps which contain a connected subgraph of six 5-vertices which is isomorphic to a subgraph of a cluster $c \in \{I, II, III, IV\}$ have different boundary parameters for each different c. Therefore dual IPR fullerenes which contain two 6-clusters c and d with $c \in \{I, II, III, IV\}$ and $d \in \{I, II, III, IV\} \setminus \{c\}$ do not exist.

All dual caps which contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster I-IV are globally irreducible as well. So all IPR fullerenes which can be decomposed into 2 caps where both caps are globally irreducible are not reducible to a smaller IPR fullerene.

By using the generator for caps from Brinkmann et al. [36], we were able to determine that all dual IPR caps with boundary parameters (5,5) (respectively (8,2) and (9,0)) contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster I (respectively II and III). However there are caps with boundary parameters (10,0) which do not contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster IV. This gives us the following corollary:

Corollary 4.26. *Every IPR fullerene which contains a (5,5), (8,2) or (9,0) boundary is not reducible to a smaller IPR fullerene.*

We will now show that all dual IPR fullerenes which have a (10,0) boundary are reducible, except for dual fullerenes where both caps contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster IV and for a limited number of dual fullerenes which contain a 12-cluster.

By using the modified version of the generator for caps from Brinkmann et al. [36], we were able to generate all IPR (10,0) kernels. The largest one has 60 vertices, so an IPR fullerene which contains a (10,0) cap and has no reducible (10,0) hexagon rings has at most $2 \cdot 60 + 2 \cdot 10 = 140$ vertices. Using *fullgen* we determined all of these fullerenes. These results were also independently confirmed by our generator for IPR fullerenes which uses a filter.

All of these dual fullerenes are reducible, except the ones where both dual caps contain a connected subgraph with six 5-vertices isomorphic to a subgraph

of cluster IV and a limited number of dual fullerenes which contain a 12-cluster. Later in this section, we will show which dual fullerenes containing a 12-cluster are irreducible.

We verified that for each of these reducible fullerenes F there is an r such that the fullerenes obtained by adding r $(10,0)$ rings of hexagons to F have a reduction which is entirely within one cap. We also verified that all fullerenes obtained from F by adding less than r $(10,0)$ rings of hexagons are reducible as well. The irreducible dual fullerenes which contain a 12-cluster where the dual caps do not contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster IV also become reducible if a $(10,0)$ ring of 6-vertices is added. Also for these dual fullerenes there is an r such that the dual fullerenes obtained by adding r $(10,0)$ rings of 6-vertices have a reduction which is entirely within one dual cap (and all of these dual fullerenes obtained by adding less than r $(10,0)$ rings of 6-vertices are reducible as well).

This gives us the following corollary:

Corollary 4.27. *Every dual IPR fullerene which contains a (10,0) boundary is reducible to a smaller dual IPR fullerene, except for dual fullerenes where both dual caps contain a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster IV and for a limited number of dual fullerenes which contain a 12-cluster.*

Together with the other corollaries from this section, this gives us:

Corollary 4.28. *All dual IPR fullerenes which contain a 6-cluster are reducible to a smaller dual IPR fullerene, unless the dual fullerene contains 2 clusters c with $c \in \{I, II, III, IV\}$*

Irreducibility of k-clusters $(7 \leq k \leq 12)$

Now we will prove that all dual IPR fullerenes which contain a k-cluster with $7 \leq k \leq 11$ are reducible to a smaller dual IPR fullerene. We will also prove that there are only a limited number of dual fullerenes which contain a 12-cluster which are not reducible to a smaller dual IPR fullerene and determine them.

For a given patch with k pentagons $(7 \leq k \leq 12)$, we can compute an upper bound for the number of vertices of a fullerene which contains this patch by using the results from [7]. Suppose for example that we have a patch P with 7 pentagons, h_P hexagons and boundary length l. We can determine an upper bound for the number of hexagons h in a patch with the same boundary length and 5 pentagons by using Theorem 12 of [7] as follows:

$$\begin{aligned} \frac{l+1}{2} &\geq \lceil \sqrt{2h + \frac{113}{4}} + \frac{1}{2} \rceil \\ \frac{l}{2} &\geq \sqrt{2h + \frac{113}{4}} \\ h &\leq \frac{l^2 - 113}{8} \end{aligned}$$

So the number of faces in a fullerene containing P is at most $7+h_P+5+\frac{l^2-113}{8}$. For patches with k ($8 \leq k \leq 12$) pentagons, an upper bound for the number of faces of a fullerene which contains such a patch is obtained in a similar way. For a patch with a given number of pentagons, the upper bound increases as the number of hexagons in the patch and its boundary length increases.

Earlier in this section, we have proven that every cluster has a cluster-tree. For a k-cluster C and a cluster-tree T_C of C, let $deg_C : V(C) \to \mathbb{N}$ stand for the degree of a vertex in C and $deg_{T_C} : V(T_C) \to \mathbb{N}$ for the degree of a vertex in T_C. In a patch which is the dual of T_C, a face corresponding with a vertex v from T_C has $deg_C(v) - deg_{T_C}(v)$ edges on the boundary. When induced edges are added between vertices of T_C, the boundary length with never increase. Therefore an upper bound for the boundary length b_{T_C} of the dual of such a cluster-tree T_C of a k-cluster C is given by:

$$\begin{aligned} b_{T_C} &\leq \sum_{v \in V(T_C)} (deg_C(v) - deg_{T_C}(v)) \\ &= 5k + 6(V(T_C) - k) - \sum_{v \in V(T_C)} deg_{T_C}(v) \\ &= 5k + 6(V(T_C) - k) - 2(V(T_C) - 1) \\ &= 4V(T_C) - k + 2 \end{aligned}$$

Thus the boundary length is maximal if $V(T_C)$ is maximal.

Lemma 4.29. *A cluster-tree of a k-cluster has at most $2k-1$ vertices.*

Proof. Given a k-cluster-tree T. Let n_6 stand for the number of 6-vertices in T, so $|V(T)| = k + n_6$. Since T is a tree, we have $|E(T)| = k + n_6 - 1$. Since 6-vertices have degree at least 2 and since there are no adjacent 6-vertices, $|E(T)|$ is at least $2n_6$. This gives us:

$$\begin{aligned} 2n_6 &\leq k + n_6 - 1 \\ \iff n_6 &\leq k - 1 \\ \iff k + n_6 &\leq 2k - 1 \end{aligned}$$

□

So the boundary length of a cluster-tree of a k-cluster is at most $4(2k-1) - k + 2 = 7k - 2$. Using the results from [7], we can now determine an upper bound for the number of vertices of a fullerene which contains the dual of a given cluster-tree T_C of a 7-cluster C.

We already determined that a patch with 5 pentagons and boundary length l has at most $\frac{l^2-113}{8}$ hexagons. So a fullerene containing the dual of T_C has at most $|V(T_C)| + 5 + \frac{b_{T_C}^2-113}{8}$ faces. The fact that T_C is a cluster-tree allows us to improve the upper bound of $\frac{b_{T_C}^2-113}{8}$ on the number of hexagons h in a patch with 5 pentagons which has the same boundary as T_C. Since the leafs of a cluster-tree are 5-vertices and the neighbour of a leaf is a 6-vertex, a part of the boundary of T_C has the shape of Figure 4.27a. The complement $\bar{P}$ of this patch in a fullerene (i.e. the shaded region in Figure 4.27a) contains 5 pentagons and h hexagons. If $\bar{P}$ exists, a patch with one additional face as in Figure 4.27b also exists. The pentagon in the boundary of this patch can be modified to a hexagon as in Figure 4.27c. This patch $\bar{P}'$ contains 5 pentagons and $h+1$ hexagons and its boundary length is $b_{T_C} - 2$. So an upper bound for the number of hexagons in $\bar{P}'$ is given by: $h + 1 \leq \frac{(b_{T_C}-2)^2-113}{8}$ (which improves the old bound $h \leq \frac{b_{T_C}^2-113}{8}$).

Since T_C is a cluster-tree, all leafs are 5-vertices and since T_C contains at least 3 vertices, there are at least 2 leafs. So there are at least two parts of the boundary which have the shape of Figure 4.27a. This second place can also be modified as in Figure 4.27c, which gives us: $h + 2 \leq \frac{(b_{T_C}-4)^2-113}{8}$. Thus a fullerene containing the dual of T_C has at most $|V(T_C)| + 5 + \frac{(b_{T_C}-4)^2-113}{8} - 2$ faces.

Since $|V(T_C)| \leq 2k - 1$ and $b_{T_C} \leq 7k - 2$, such a fullerene has at most $2 \cdot 7 - 1 + 5 + \frac{(7 \cdot 7-2-4)^2-113}{8} - 2 = 233$ faces or $2 \cdot (233 - 2) = 462$ vertices. We computed the maximal number of vertices of a fullerene containing a k-cluster ($8 \leq k \leq 12$) in a similar way. The results are shown in Table 4.1.

Note that this upper bound is very coarse since the patches with the largest number of hexagons given in [7] for a given number of pentagons and boundary

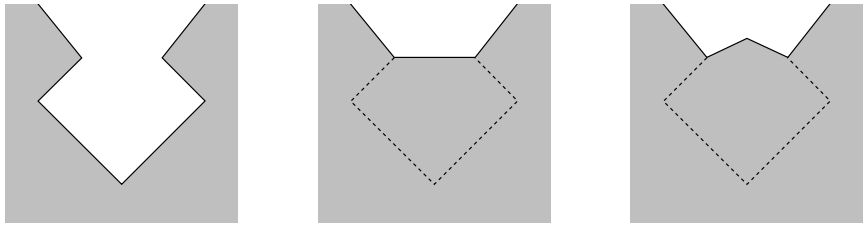


Figure 4.27: Part of the boundary of a k-cluster tree (i.e. Figure 4.27a) and how it can be modified to obtain a better bound on the number of faces in the complementary part (i.e. Figures 4.27b and 4.27c).

k	max nv
7	462
8	330
9	296
10	286
11	286
12	292

Table 4.1: Upper bound for the number of vertices of a fullerene containing the dual of a k-cluster.

length are not IPR if the patch contains at least 2 pentagons.

Using *fullgen* we generated all IPR fullerenes up to 330 vertices and tested them for reducibility. This was independently verified by our generator for IPR fullerenes from Section 4.3.1. We obtained the following results:

Observation 4.30. *All dual IPR fullerenes which contain a k-cluster* ($8 \leq k \leq 11$) *are reducible to a smaller dual IPR fullerene.*

Observation 4.31. *There are exactly 56 irreducible dual IPR fullerenes which contain a 12-cluster. The largest one has 58 vertices or* $2 \cdot (58 - 2) = 112$ *faces.*

Observation 4.32. *There are exactly 36 irreducible dual IPR fullerenes which contain a 12-cluster and which do not have a dual cap which contains a connected subgraph with six 5-vertices which is isomorphic to a subgraph of cluster I, II, III or IV.*

It was not feasible to generate all IPR fullerenes up to 462 vertices with *fullgen*. However, our generator for locally irreducible clusters was still fast enough to generate all locally irreducible 7-clusters. By using the dual patches P of these specific 7-clusters which have boundary length b_P in the formula $|V(P)| + 5 + \frac{b_P^2-113}{8}$, we were able to determine that fullerenes which contain the dual of one of these locally irreducible 7-clusters have at most 166 vertices. Using *fullgen* we generated all these fullerenes and tested them for reducibility. We obtained the following result (which was independently confirmed by our generator for IPR fullerenes which uses a filter):

Corollary 4.33. *All dual IPR fullerenes which contain a 7-cluster are reducible to a smaller dual IPR fullerene.*

Actually we only had to prove that dual IPR fullerenes which contain one 7-cluster and five 1-clusters (or one 8-cluster and four 1-clusters etc.) are reducible. Since e.g. a dual fullerene consisting of a 7-cluster and a 5-cluster is always reducible since all 5-clusters are locally reducible (see Observation 4.17).

Together, Corollaries 4.16, 4.17, 4.20, 4.28, 4.30, 4.31 and 4.33 give the following theorem:

Theorem 4.34. *The class of irreducible dual IPR fullerenes consists of an infinite family of dual IPR fullerenes which contain two 6-clusters c with $c \in \{I, II, III, IV\}$ and 36 dual fullerenes which contain a 12-cluster.*

Generation algorithm

In order to generate all IPR fullerenes with n vertices, the generation algorithm recursively applies the construction operations to all irreducible IPR fullerenes with at most n vertices. The construction operations are the same as the operations for the generation of all dual fullerenes from Section 4.2, but here only expansions for which the expanded dual fullerene is also IPR are performed. Unfortunately the lemmas from Section 4.2.3 which allow to determine a good bound on the length of the shortest reduction cannot be applied for the recursive generation of IPR fullerenes. The isomorphism rejection routines are also the same as in Section 4.2, but here only reductions to smaller IPR fullerenes are considered.

The four infinity families of irreducible dual IPR nanotube fullerenes which contain two 6-clusters c with $c \in \{I, II, III, IV\}$ consist of dual caps with boundary parameters $(5, 5)$, $(8, 2)$, $(9, 0)$ or $(10, 0)$, respectively. They are generated by

adding rings of 6-vertices with the respective parameters in all possible ways. Since there are only a small number of irreducible IPR fullerenes, we use isomorphism rejection by lists (see Section 1.3) to make sure that no isomorphic irreducible IPR fullerenes are output. For example there are 23 098 573 580 IPR fullerenes with up to 300 vertices while only 162 of them are irreducible.

In Section 4.4, we compare the running time of this generator with *fullgen* and the generator which uses a filter from Section 4.3.1.

Open questions

When classifying these irreducible IPR fullerenes we encountered some open questions. Future work might include solving these open questions:

- Can every fullerene be split into two *caps*? By performing a computer search, we verified that all fullerenes up to 200 vertices can be split into two caps.

- Does a 12-cluster uniquely determine a dual fullerene? Or equivalently: does a boundary sequence uniquely describe the interior of a subpatch of a fullerene which only consists of hexagons?

  It is known [18, 64] that the boundary of a hexagon patch determines the number of faces of the patch. It is also known that the boundary sequence uniquely describes the interior of a hexagonal patch if it is a subgraph of the hexagonal lattice and it has been shown by Guo et al. [64] that this is not the case if the patch is not necessarily a subgraph of the hexagon lattice. For hexagon patches which are subgraphs of fullerenes, it is unknown.

4.4 Testing and results

4.4.1 Running times and the number of fullerenes

The running times and a comparison with *fullgen* are given in Table 4.2. Our generator is called *buckygen*. The programs were compiled with gcc and executed on an Intel Xeon L5520 CPU at 2.27 GHz. The running times include writing the fullerenes to a null device.

As can be seen from that table, *buckygen* is significantly faster than *fullgen*. Our generator constructs larger fullerenes from smaller ones, so in order to generate all fullerenes with n vertices, all fullerenes with at most $n - 4$ vertices

number of vertices	time (s) (buckygen)	fullerenes/s (buckygen)	fullgen (s) / buckygen (s)
100	6.8	42 358	7.30
140	220	33 369	7.39
170	2 167	21 268	5.63
200	12 631	16 953	5.49
230	62 520	12 597	5.13
260	263 278	9 408	4.59
280	636 944	7 735	4.43
300	1 436 972	6 494	4.07
320	3 078 995	5 502	3.67
20–100	9.1	159 365	24.96
102–150	671	157 736	33.04
152–200	17 733	115 625	32.08
202–250	235 108	82 813	32.08

Table 4.2: Running times and generation rates for fullerenes.

have to be generated as well (recall that an L_0 expansion increases the number of vertices by 4). *Fullgen* does not construct fullerenes from smaller ones and therefore the speedup of *buckygen* is decreasing. For an extremely large number of vertices (more than 600 vertices), *fullgen* would probably be faster than our generator. However generating all fullerenes with 600 vertices is certainly far outside the scope of what is computationally feasible as e.g. generating all fullerenes with 300 vertices takes more than 1 CPU year with *buckygen* and generating all fullerenes with 320 vertices already requires more than 2 CPU years. So for the sizes which are computationally feasible, our generator is at least 3 times faster than *fullgen*.

Since *buckygen* constructs larger fullerenes from smaller ones, generating all fullerenes with at most n vertices gives only a small overhead compared to generating all fullerenes with exactly n vertices. In *fullgen* the overhead is considerably larger as it does not construct fullerenes from smaller fullerenes. For example, *buckygen* can generate all fullerenes with $n \in [290, 300]$ vertices more than 15 times faster than *fullgen*. More comparisons can be found in the bottom part of Table 4.2.

A comparison of the running times for generating IPR fullerenes is given in Table 4.3. *Buckygen IPR* is the specialised algorithm from Section 4.3.2 which re-

number of vertices	time (s) (bg IPR)	fullerenes/s (bg IPR)	fg IPR (s) / bg IPR (s)	fg IPR (s) / bg filter (s)	bg filter (s) / bg IPR (s)
200	4 110	3 809	1.88	0.80	2.34
230	22 481	3 836	2.14	0.96	2.23
260	104 831	3 456	2.18	1.03	2.21
280	274 748	3 066	2.19	1.10	2.00
300	678 331	2 686	2.19	1.16	1.88
320	1 591 041	2 329	1.99	1.14	1.75
340	3 613 915	1 981	1.73	1.09	1.60
360	8 135 063	1 625	1.51	1.05	1.43
0–140	17.5	33 055	19.62	1.99	9.85
290–300	776 910	11 753	7.83	4.11	1.91

Table 4.3: Running times and generation rates for IPR fullerenes. *Bg* stands for *buckygen* and *fg* stands for *fullgen*.

cursively generates IPR fullerenes from smaller fullerenes. *Buckygen filter* is the algorithm from Section 4.3.1 which generates all fullerenes with some look-aheads and a filter for IPR fullerenes. *Buckygen IPR* is considerably faster than *fullgen* for generating IPR fullerenes, but also in this case the speedup is decreasing. This is again because *fullgen* constructs fullerenes directly instead of constructing them from smaller fullerenes as in *buckygen*. On the other hand, generating all fullerenes for a range of vertices is much more efficient with *buckygen*. For example, *buckygen IPR* can generate all IPR fullerenes with $n \in [290, 300]$ vertices more than 7 times faster than *fullgen*. The reason that the speedup of *buckygen IPR* compared to *buckygen filter* is decreasing, is because the lemmas from Section 4.2.3 which allow to determine a good bound on the length of the shortest reduction cannot be applied to the specialised generation algorithm which recursively generates IPR fullerenes from smaller ones. Furthermore the ratio of IPR fullerenes among all fullerenes is increasing (see Table 4.4), thus the ratio of fullerenes which are rejected by *buckygen filter* because they are not IPR is decreasing. So for generating IPR fullerenes with an extremely large number of vertices, *buckygen filter* would be most efficient.

Buckygen was used to generate all fullerenes up to 400 vertices. Contradicting results with *fullgen* led to the detection of a programming error in *fullgen*. Due to this error some fullerenes were missed starting from 136 vertices and IPR fullerenes starting from 254 vertices. In the meantime this bug has been fixed and now the results of the two programs are in complete agreement up to at least

nv	fullerenes	IPR fullerenes	percentage IPR
60	1 812	1	0.06
100	285 914	450	0.16
150	14 059 174	335 569	2.39
200	214 127 742	15 655 672	7.31
250	1 712 934 069	230 272 559	13.44
300	9 332 065 811	1 821 766 896	19.52
350	38 580 626 759	9 768 511 147	25.32
400	132 247 999 328	40 286 153 024	30.46

Table 4.4: Percentage of IPR fullerenes. nv stands for the number of vertices.

380 vertices, which is a good check of both. We also generated all IPR fullerenes up to 400 vertices by using *buckygen IPR*. These results were independently confirmed by *buckygen filter* and *fullgen IPR* up to 380 vertices. We give the counts in Tables 4.5–4.9, which correct those in the article of Brinkmann and Dress [19] where they overlap. By applying the program *plantri_md6* from [34] to the generated fullerenes, we were also able to generate all cubic plane graphs with maximum face size 6 and minimum face size 3 up to 176 vertices and all cubic plane graphs with maximum face size 6 and minimum face size 4 up to 316 vertices. Thurston [118] proved that the number of fullerenes with $2n$ vertices grows at the rate of n^9. The fullerenes themselves can be downloaded from [26] for small sizes. We decided to run independent tests up to 380 vertices with *fullgen* in order to also have an independent check of the minimality of the counterexample to the spiral conjecture (see Section 4.4.2). Independent computations for more than 380 vertices or using *buckygen* for more than 400 vertices would be possible, but would require an enormous amount of computing time.

nv	nf	min. face 3	min. face 4	fullerenes	IPR fullerenes
4	4	1	0	0	0
6	5	1	0	0	0
8	6	1	1	0	0
10	7	4	1	0	0
12	8	8	2	0	0
14	9	11	4	0	0
16	10	23	7	0	0
18	11	34	10	0	0
20	12	54	22	1	0
22	13	83	32	0	0
24	14	125	58	1	0
26	15	174	92	1	0
28	16	267	151	2	0
30	17	365	227	3	0
32	18	509	368	6	0
34	19	706	530	6	0
36	20	963	805	15	0
38	21	1 270	1 158	17	0
40	22	1 708	1 695	40	0
42	23	2 204	2 373	45	0
44	24	2 876	3 354	89	0
46	25	3 695	4 595	116	0
48	26	4 708	6 340	199	0
50	27	5 925	8 480	271	0
52	28	7 491	11 417	437	0
54	29	9 255	15 049	580	0
56	30	11 463	19 832	924	0
58	31	14 083	25 719	1 205	0
60	32	17 223	33 258	1 812	1
62	33	20 857	42 482	2 385	0
64	34	25 304	54 184	3 465	0
66	35	30 273	68 271	4 478	0
68	36	36 347	85 664	6 332	0
70	37	43 225	106 817	8 149	1
72	38	51 229	132 535	11 190	1
74	39	60 426	163 194	14 246	1
76	40	71 326	200 251	19 151	2
78	41	83 182	244 387	24 109	5
80	42	97 426	296 648	31 924	7
82	43	113 239	358 860	39 718	9
84	44	131 425	431 578	51 592	24
86	45	151 826	517 533	63 761	19
88	46	175 302	617 832	81 738	35

Table 4.5: Cubic plane graphs with maximum face size 6 listed with respect to their minimum face size. Cubic plane graphs with maximum face size 6 and with minimum face size 5 are fullerenes. nv is the number of vertices and nf is the number of faces.

nv	nf	min. face 3	min. face 4	fullerenes	IPR fullerenes
90	47	200 829	735 257	99 918	46
92	48	231 042	870 060	126 409	86
94	49	263 553	1 029 114	153 493	134
96	50	300 602	1 209 783	191 839	187
98	51	341 960	1 420 472	231 017	259
100	52	388 673	1 659 473	285 914	450
102	53	438 795	1 937 509	341 658	616
104	54	496 961	2 249 285	419 013	823
106	55	559 348	2 612 410	497 529	1 233
108	56	629 807	3 015 386	604 217	1 799
110	57	706 930	3 483 289	713 319	2 355
112	58	792 703	4 002 504	860 161	3 342
114	59	885 137	4 600 343	1 008 444	4 468
116	60	990 929	5 257 856	1 207 119	6 063
118	61	1 102 609	6 019 580	1 408 553	8 148
120	62	1 227 043	6 849 385	1 674 171	10 774
122	63	1 363 825	7 805 813	1 942 929	13 977
124	64	1 513 612	8 846 570	2 295 721	18 769
126	65	1 673 568	10 041 875	2 650 866	23 589
128	66	1 853 928	11 335 288	3 114 236	30 683
130	67	2 045 154	12 821 597	3 580 637	39 393
132	68	2 255 972	14 415 241	4 182 071	49 878
134	69	2 485 363	16 248 586	4 787 715	62 372
136	70	2 732 106	18 211 371	5 566 949	79 362
138	71	2 998 850	20 454 114	6 344 698	98 541
140	72	3 295 090	22 845 387	7 341 204	121 354
142	73	3 606 102	25 587 469	8 339 033	151 201
144	74	3 944 923	28 486 985	9 604 411	186 611
146	75	4 316 999	31 808 776	10 867 631	225 245
148	76	4 711 038	35 313 026	12 469 092	277 930
150	77	5 135 794	39 315 258	14 059 174	335 569
152	78	5 599 065	43 529 295	16 066 025	404 667
154	79	6 091 434	48 339 505	18 060 979	489 646
156	80	6 621 013	53 361 979	20 558 767	586 264
158	81	7 198 926	59 117 693	23 037 594	697 720
160	82	7 800 960	65 110 208	26 142 839	836 497
162	83	8 460 776	71 938 170	29 202 543	989 495
164	84	9 168 333	79 041 733	33 022 573	1 170 157
166	85	9 917 772	87 147 815	36 798 433	1 382 953
168	86	10 711 603	95 517 631	41 478 344	1 628 029
170	87	11 590 680	105 090 752	46 088 157	1 902 265
172	88	12 491 734	114 936 807	51 809 031	2 234 133
174	89	13 479 003	126 169 808	57 417 264	2 601 868
176	90	14 518 882	137 732 548	64 353 269	3 024 383

Table 4.6: Cubic plane graphs with maximum face size 6 listed with respect to their minimum face size (continued). nv is the number of vertices and nf is the number of faces.

nv	nf	min. face 4	fullerenes	IPR fullerenes
178	91	150 895 768	71 163 452	3 516 365
180	92	164 343 840	79 538 751	4 071 832
182	93	179 752 024	87 738 311	4 690 880
184	94	195 420 760	97 841 183	5 424 777
186	95	213 287 269	107 679 717	6 229 550
188	96	231 489 614	119 761 075	7 144 091
190	97	252 233 869	131 561 744	8 187 581
192	98	273 226 069	145 976 674	9 364 975
194	99	297 264 792	159 999 462	10 659 863
196	100	321 450 554	177 175 687	12 163 298
198	101	349 098 672	193 814 658	13 809 901
200	102	376 999 869	214 127 742	15 655 672
202	103	408 774 872	233 846 463	17 749 388
204	104	440 627 726	257 815 889	20 070 486
206	105	477 200 827	281 006 325	22 606 939
208	106	513 632 380	309 273 526	25 536 557
210	107	555 304 108	336 500 830	28 700 677
212	108	596 974 072	369 580 714	32 230 861
214	109	644 526 803	401 535 955	36 173 081
216	110	691 786 828	440 216 206	40 536 922
218	111	746 085 995	477 420 176	45 278 722
220	112	799 648 739	522 599 564	50 651 799
222	113	861 133 064	565 900 181	56 463 948
224	114	922 082 216	618 309 598	62 887 775
226	115	991 650 902	668 662 698	69 995 887
228	116	1 060 208 550	729 414 880	77 831 323
230	117	1 139 239 947	787 556 069	86 238 206
232	118	1 216 496 915	857 934 016	95 758 929
234	119	1 305 306 936	925 042 498	105 965 373
236	120	1 392 596 607	1 006 016 526	117 166 528
238	121	1 492 525 091	1 083 451 816	129 476 607
240	122	1 590 214 959	1 176 632 247	142 960 479
242	123	1 702 998 124	1 265 323 971	157 402 781
244	124	1 812 247 954	1 372 440 782	173 577 766
246	125	1 938 356 975	1 474 111 053	190 809 628

Table 4.7: Triangle-free cubic plane graphs with maximum face size 6 listed with respect to their minimum face size. nv is the number of vertices and nf is the number of faces.

nv	nf	min. face 4	fullerenes	IPR fullerenes
248	126	2 061 311 003	1 596 482 232	209 715 141
250	127	2 202 202 308	1 712 934 069	230 272 559
252	128	2 338 869 735	1 852 762 875	252 745 513
254	129	2 497 257 527	1 985 250 572	276 599 787
256	130	2 649 382 974	2 144 943 655	303 235 792
258	131	2 825 361 014	2 295 793 276	331 516 984
260	132	2 995 557 818	2 477 017 558	362 302 637
262	133	3 191 292 821	2 648 697 036	395 600 325
264	134	3 379 722 482	2 854 536 850	431 894 257
266	135	3 598 542 661	3 048 609 900	470 256 444
268	136	3 806 922 124	3 282 202 941	512 858 451
270	137	4 049 087 424	3 501 931 260	557 745 670
272	138	4 281 540 754	3 765 465 341	606 668 511
274	139	4 549 259 510	4 014 007 928	659 140 287
276	140	4 805 073 991	4 311 652 376	716 217 922
278	141	5 103 457 703	4 591 045 471	776 165 188
280	142	5 385 296 261	4 926 987 377	842 498 881
282	143	5 713 728 893	5 241 548 270	912 274 540
284	144	6 026 548 238	5 618 445 787	987 874 095
286	145	6 388 285 729	5 972 426 835	1 068 507 788
288	146	6 731 485 975	6 395 981 131	1 156 161 307
290	147	7 132 734 985	6 791 769 082	1 247 686 189
292	148	7 508 699 038	7 267 283 603	1 348 832 364
294	149	7 948 994 131	7 710 782 991	1 454 359 806
296	150	8 365 304 423	8 241 719 706	1 568 768 524
298	151	8 847 679 520	8 738 236 515	1 690 214 836
300	152	9 302 042 370	9 332 065 811	1 821 766 896
302	153	9 835 862 103	9 884 604 767	1 958 581 588
304	154	10 332 102 625	10 548 218 751	2 109 271 290
306	155	10 915 020 041	11 164 542 762	2 266 138 871
308	156	11 462 133 758	11 902 015 724	2 435 848 971
310	157	12 098 825 145	12 588 998 862	2 614 544 391
312	158	12 694 519 224	13 410 330 482	2 808 510 141
314	159	13 396 207 247	14 171 344 797	3 009 120 113
316	160	14 043 402 497	15 085 164 571	3 229 731 630

Table 4.8: Triangle-free cubic plane graphs with maximum face size 6 listed with respect to their minimum face size (continued). nv is the number of vertices and nf is the number of faces.

nv	nf	fullerenes	IPR fullerenes
318	161	15 930 619 304	3 458 148 016
320	162	16 942 010 457	3 704 939 275
322	163	17 880 232 383	3 964 153 268
324	164	19 002 055 537	4 244 706 701
326	165	20 037 346 408	4 533 465 777
328	166	21 280 571 390	4 850 870 260
330	167	22 426 253 115	5 178 120 469
332	168	23 796 620 378	5 531 727 283
334	169	25 063 227 406	5 900 369 830
336	170	26 577 912 084	6 299 880 577
338	171	27 970 034 826	6 709 574 675
340	172	29 642 262 229	7 158 963 073
342	173	31 177 474 996	7 620 446 934
344	174	33 014 225 318	8 118 481 242
346	175	34 705 254 287	8 636 262 789
348	176	36 728 266 430	9 196 920 285
350	177	38 580 626 759	9 768 511 147
352	178	40 806 395 661	10 396 040 696
354	179	42 842 199 753	11 037 658 075
356	180	45 278 616 586	11 730 538 496
358	181	47 513 679 057	12 446 446 419
360	182	50 189 039 868	13 221 751 502
362	183	52 628 839 448	14 010 515 381
364	184	55 562 506 886	14 874 753 568
366	185	58 236 270 451	15 754 940 959
368	186	61 437 700 788	16 705 334 454
370	187	64 363 670 678	17 683 643 273
372	188	67 868 149 215	18 744 292 915
374	189	71 052 718 441	19 816 289 281
376	190	74 884 539 987	20 992 425 825
378	191	78 364 039 771	22 186 413 139
380	192	82 532 990 559	23 475 079 272
382	193	86 329 680 991	24 795 898 388
384	194	90 881 152 117	26 227 197 453
386	195	95 001 297 565	27 670 862 550
388	196	99 963 147 805	29 254 036 711
390	197	104 453 597 992	30 852 950 986
392	198	109 837 310 021	32 581 366 295
394	199	114 722 988 623	34 345 173 894
396	200	120 585 261 143	36 259 212 641
398	201	125 873 325 588	38 179 777 473
400	202	132 247 999 328	40 286 153 024

Table 4.9: Counts of fullerenes and IPR fullerenes. nv is the number of vertices and nf is the number of faces.

4.4.2 Testing conjectures related to fullerenes

We also tested some conjectures related to fullerenes, more specfically Barnette's conjecture and the spiral conjecture.

Barnette's conjecture

There is a famous conjecture from Barnette which conjectures the following:

Conjecture 4.35 (Barnette, 1969)**.** *Every 3-connected cubic planar graph with maximum face size 6 is hamiltonian.*

All 3-connected cubic planar graphs with up to n vertices and maximum face size 6 can be obtained by applying the program *plantri_md6* from [34] to all fullerenes with up to n vertices. We checked Barnette's conjecture up to 316 vertices. For fullerenes (which have minimum face size 5), we checked it up to 336 vertices: no counterexamples were found. We did not test this conjecture for larger fullerenes, since not the generation but the hamiltonicity test is the bottleneck.

The number of cubic plane graphs with maximum face size 6 with respect to their minimum face size is listed in Tables 4.5-4.8. This corrects and extends the results of the computation reported by Brinkmann et al. [34], which relied on the faulty version of *fullgen*. In that paper they had verified Barnette's conjecture up to 250 vertices.

The spiral conjecture

Recall from Section 4.1.2 that the first algorithm to generate fullerenes was the spiral algorithm of Manolopoulos et al. [86]. It constructs fullerenes by enumerating *spiral codes* – that are encodings of *face spirals*.

A clockwise (respectively counterclockwise) face spiral (see Figure 4.28) of a fullerene with k faces is a sequence of distinct faces $(f_1, f_2, \ldots, f_k)$ with the property that f_1 and f_2 share an edge and that for $3 \leq i \leq k$ face f_i has a connected intersection with $\{f_1 \cup f_2 \cup \cdots \cup f_{i-1}\}$ and shares an edge e with f_{i-1} that is the last of those edges of f_{i-1} in clockwise (respectively counterclockwise) orientation around f_{i-1} that belong to no face in $\{f_1, f_2, \ldots, f_{i-2}\}$.

Such face spirals can be encoded as sequences of the face sizes, or more compactly as a sequence of length 12 which contains the positions of the 12 pentagons in the spiral. Thus fullerenes which have a face spiral can be encoded in a very compact way. Figure 4.28 shows an example of a face spiral of the C_{60} buckyball.

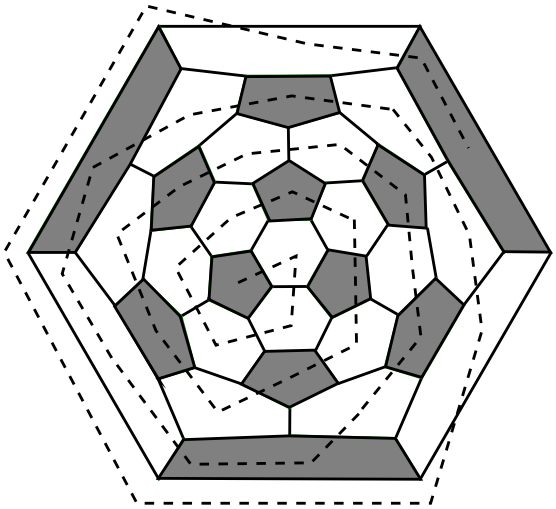


Figure 4.28: A face spiral of the C_{60} buckyball with encoding $1, 7, 9, 11, 13, 15, 18, 20, 22, 24, 26, 32$.

The *spiral conjecture* (see [53]) stated that every fullerene can be described by a face spiral.

Already in [85] Manolopoulos and Fowler give an example of a fullerene with 380 vertices which does not allow any face spirals and would therefore be missed by their algorithm. The question was whether this is the smallest counterexample to the spiral conjecture. In [12] it is shown that in the more general class of all cubic polyhedra, the smallest element which does not allow any face spirals has only 18 vertices. So one might have expected that there is a smaller counterexample in the class of fullerenes.

For efficiency reasons, Manolopoulos and Fowler restricted the generation of fullerene spiral codes to spirals starting at a pentagon (see [53]). It is known that the smallest fullerene that is missed in this way has 100 vertices. The smallest IPR fullerene that does not have a spiral starting at a pentagon has 206 vertices.

Later, the *International Union of Pure and Applied Chemistry* (IUPAC) also recommended [57] to use face spirals as basis for fullerene nomenclature.

Thus it would be interesting to know the smallest fullerene which does not allow a face spiral. Therefore we used *buckygen* to generate all fullerenes up to 400 vertices and tested them for face spirals. The result is that the fullerene with 380 vertices depicted in Figure 4.29 is the smallest fullerene without a face

spiral. This is exactly the counterexample from Manolopoulos and Fowler [85], but at that time its minimality could not be proven. The second smallest fullerene without a spiral has 384 vertices and is shown in Figure 4.30. Again, this example was already obtained before by construction (by Yoshida and Fowler [124]).

The two fullerenes from Figure 4.29 and 4.30 are the only fullerenes without spirals up to 400 vertices. The counts of fullerenes which do not have a spiral starting at a pentagon can be found in [30]. We have independently confirmed these results up to 380 vertices by running *fullgen*.

Recently Fowler et al. [52] determined graphs which do not allow face spirals for all of the remaining classes of cubic polyhedra with maximum face size 6.

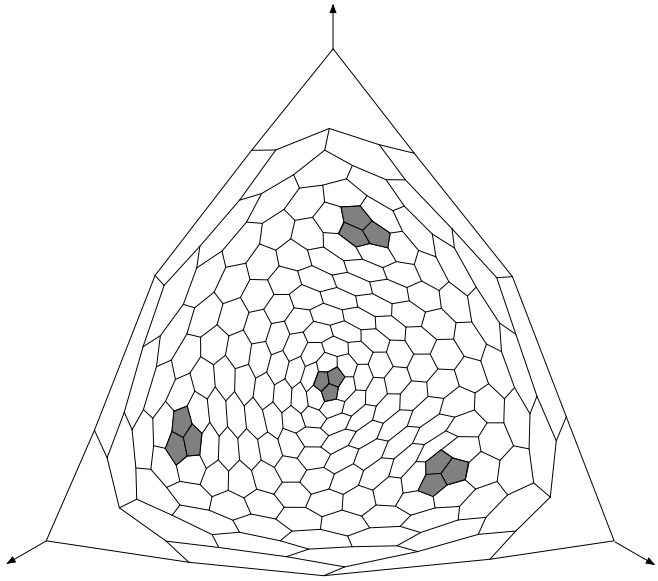


Figure 4.29: The smallest fullerene without a spiral. In order to show the rotational symmetry with vertices as centres of rotation, one vertex has to be chosen at infinity.

4.5 Closing remarks

We have described a new fullerene generator *buckgen* which is considerably faster than *fullgen*, which is the only previous generator capable of going beyond 100

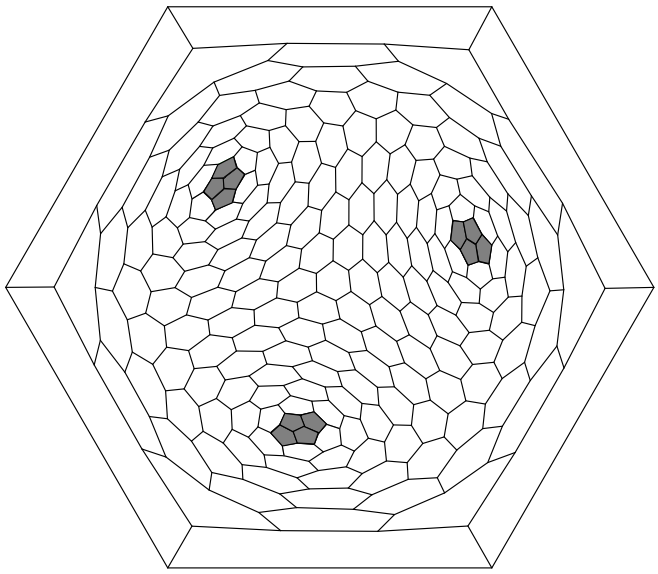


Figure 4.30: The second smallest fullerene without a spiral.

vertices. The generation cost is now likely to be lower than that of any significant computation performed on the generated structures (such as e.g. testing if the fullerenes are hamiltonian or performing energy computations).

After correction of an error in *fullgen*, we now have two independent counts of fullerenes up to 380 vertices which are in complete agreement, and counts up to 400 vertices from *buckgen*.

The latest version of *buckygen* can be downloaded from [25]. *Buckygen* is also part of the *CaGe* software package [17].

Chapter 5

Ramsey numbers

In the first part of this chapter we describe an optimised algorithm for the generation of maximal triangle-free graphs and triangle Ramsey graphs. This enabled us to compute all triangle Ramsey numbers $R(K_3, G)$ up to 30 for graphs of order 10. We also prove several theoretical results which allow us to determine several triangle Ramsey numbers larger than 30. By combining the new computational and theoretical results, we determine the Ramsey numbers $R(K_3, G)$ of nearly all of the 12 005 168 graphs of order 10, except for 10 of the hardest cases.

Remark: this was joint work also with Jan-Christoph Schlage-Puchta, who contributed the theoretical results. The research is also described in [31]. Various parts of this chapter were adopted from that article.

In the second part of this chapter we derive six new upper bounds for the classical triangle Ramsey numbers $R(3, k)$ by using specialised computational techniques: $R(3, 10) \leq 42$, $R(3, 11) \leq 50$, $R(3, 13) \leq 68$, $R(3, 14) \leq 77$, $R(3, 15) \leq 87$ and $R(3, 16) \leq 98$. All of them are improvements by one over the previously best published bounds. We also determine all *critical* triangle Ramsey graphs for K_8 (i.e. triangle Ramsey graphs for K_8 with $R(3, 8) - 1$ vertices) and prove that the known critical triangle Ramsey graph for K_9 with 35 vertices is unique up to isomorphism.

Remark: this second part was joint work with Stanisław P. Radziszowski and it is also described in [59]. Various parts of this chapter were adopted from that article.

5.1 Introduction

Ramsey theory is often described as the study of how order emerges from randomness. The *Ramsey number* $R(G, H)$ of two graphs G and H is the smallest integer r such that every assignment of two colours (e.g. red and blue) to the edges of K_r contains G as a red subgraph or H as a blue subgraph. Or equivalently, $R(G, H)$ is the smallest integer r such that every graph F with at least r vertices contains G as a subgraph, or its complement F^c contains H as a subgraph. A graph F is a *Ramsey graph* for a pair of graphs (G, H) if F does not contain G as a subgraph and its complement F^c does not contain H as a subgraph. This concept also generalises to n colours (i.e. $R(G_1, G_2, ..., G_n)$), but in this thesis we focus on two colours.

The existence of $R(G, H)$ follows from Ramsey's theorem [103] from 1930. The *classical* Ramsey numbers (where both G and H are complete graphs) are known to be extremely difficult to determine. It is even difficult to obtain narrow bounds when G or H have a large order. Therefore only few exact results are known. The last exact result was obtained by McKay and Radziszowski [91] in 1995 when they proved that $R(K_4, K_5) = 25$.

For a good overview of the results and bounds of Ramsey numbers which are currently known, we refer the reader to Radziszowski's dynamic survey [99]. We also denote the classical Ramsey number $R(K_m, K_n)$ by $R(m, n)$. In Section 5.3, we derive new upper bounds for the Ramsey numbers $R(3, n)$ for $10 \leq n \leq 16$ by using specialised computational techniques.

The requirement that G^c contains K_n as a subgraph is equivalent to the requirement that G contains an independent set of order n. Therefore the classical Ramsey number $R(m, n)$ can also be interpreted as a solution to the *party problem*: what is the minimum number of guests that must be invited to a party so that it is guaranteed that at least m people will be mutual acquaintances or at least n people will be mutual strangers?

Since Ramsey numbers are difficult to determine, straightforward methods usually do not work. Therefore advanced computational techniques are required for solving Ramsey-type problems. It is likely that these techniques will also be applicable in other areas of combinatorics which require extensive computations with a very large number of graphs or with very large graphs. Ramsey theory has amongst others applications in theoretical computer science. We refer the reader to [108] and [55] for a survey of applications.

In this thesis, we focus on *triangle* Ramsey numbers: that is Ramsey numbers $R(G, H)$ where $G = K_3$. When we speak about Ramsey numbers or Ramsey

graphs in the remainder of this chapter, we always mean triangle Ramsey numbers or triangle Ramsey graphs, respectively.

Already in 1980 all triangle Ramsey numbers for graphs of order 6 were determined by Faudree, Rousseau and Schelp [51]. In 1993 the Ramsey numbers for connected graphs of order 7 were computed by Jin Xia [123]. Unfortunately, some of his results turned out to be incorrect. These were later corrected by Brinkmann [13] who computationally determined all triangle Ramsey numbers for graphs of order 7 and 8. Independently, Schelten and Schiermeyer also determined Ramsey numbers of graphs of order 7 by hand [111, 112].

Brandt, Brinkmann and Harmuth [8] gave all triangle Ramsey numbers for connected graphs of order 9 and all Ramsey numbers $R(K_3, G) \leq 24$ for connected graphs of order 10. For 2001 graphs of order 10 the Ramsey number remained open. As not only the number of graphs is increasing very fast, but also the difficulty to determine Ramsey numbers, it is very likely that the table of all triangle Ramsey numbers for graphs of order 10 is the last complete list that can possibly be determined for a very long time.

In Section 5.2 we describe an efficient algorithm for the generation of all non-isomorphic maximal triangle-free graphs and triangle Ramsey graphs together with new theoretical results. This allowed us to determine the Ramsey number of all graphs of order 10, except for 10 of the hardest cases.

In Section 5.3, we derive new upper bounds for the Ramsey numbers $R(3, n)$ by using computational techniques.

5.2 Generalised triangle Ramsey numbers

5.2.1 Introduction

In this section we describe an algorithm for generating maximal triangle-free graphs and triangle Ramsey graphs. A *maximal* triangle-free graph (in short, an *mtf graph*) is a triangle-free graph such that the insertion of each new edge introduces a triangle. For graphs with more than 2 vertices this is equivalent to being triangle-free and having diameter 2.

Note that adding edges to a graph removes edges from its complement. So if F is a triangle Ramsey graph for some graph G, all triangle-free graphs (with $|V(F)|$ vertices) obtained from F by adding edges are also triangle Ramsey graphs for G. If an mtf graph M is not a triangle Ramsey graph for G, graphs obtained from M by removing edges are not triangle Ramsey graphs for G. Hence, there

is a triangle Ramsey graph of order r for G if and only if there is an mtf graph of order r that is a Ramsey graph for G (in short, an *mtf Ramsey graph*).

Thus in order to prove that $R(K_3, G) = r$, we have to show that:

- There are no mtf Ramsey graphs for G with r vertices (which implies $R(K_3, G) \leq r$).
- There is an mtf Ramsey graph for G with $r - 1$ vertices (which implies $R(K_3, G) > r - 1$).

Even though only a very small portion of the triangle-free graphs are also maximal (see Table 5.1), the number of mtf graphs still grows very fast (see Table 5.2). Thus it is computationally not feasible for large r to generate all mtf graphs with r vertices and test if they are Ramsey graphs for a given G. Therefore it is necessary to include the restriction to Ramsey graphs already in the generation process.

$\|V(G)\|$	triangle-free graphs	mtf graphs	percentage mtf
1	1	1	100
2	2	1	50
3	3	1	33.333333
4	7	2	28.571429
5	14	3	21.428571
6	38	4	10.526316
7	107	6	5.607477
8	410	10	2.439024
9	1 897	16	0.843437
10	12 172	31	0.254683
11	105 071	61	0.058056
12	1 262 180	147	0.011647
13	20 797 002	392	0.001885
14	467 871 369	1 274	0.000272
15	14 232 552 452	5 036	0.000035
16	581 460 254 001	25 617	0.000004

Table 5.1: Counts of triangle-free and maximal triangle-free graphs.

In Section 5.2.3 we describe an algorithm for the generation of all non-isomorphic mtf graphs. This algorithm uses the same basic approach as the

algorithm by Brandt et al. [9], but also uses some structural information obtained from [9] to significantly speed up the generation. In Section 5.2.4 we describe how we extended this algorithm to generate only mtf Ramsey graphs for a given graph G. In Section 5.2.5 we describe how we used the generator for Ramsey graphs to determine the Ramsey numbers $R(K_3, G)$. The main difference between the approach used in this thesis and the approach described in [8] is that the approach used here is optimised for small lists of graphs with large Ramsey numbers while the approach in [8] is optimised for large lists with comparatively small Ramsey numbers.

Next to these computational results, we also list new theoretical results in Section 5.2.2. By combining these computational and theoretical results, we were able to determine the Ramsey numbers $R(K_3, G)$ of nearly all of the 12 005 168 graphs of order 10, except for 10 of the hardest cases. Our results are presented in Section 5.2.6.

5.2.2 Theoretical results

In this section we list several general results on Ramsey numbers of the form $R(K_3, G)$, where G is close to a complete graph. These results were proven by Jan-Christoph Schlage-Puchta, therefore we refer the reader to [31] for the proofs.

The first result is a slight modification of Theorem 1 from [8]. We give the proof for completeness. The proofs of the other lemmas in this section use a similar argument and can be found in [31].

Lemma 5.1. *Let M be a triangle-free graph with r vertices, such that M^c contains K_{n-1}, and let s be an integer satisfying $1 \leq s < n$ and $(r-n)(s+1) > (n-1)(n-2)$. Then M^c contains $K_n - K_{1,s}$.*

Proof. Suppose otherwise. If M contains a vertex with degree at least n, then M^c contains K_n since M is triangle-free. So assume $\Delta(M) \leq n-1$. Now assume that M contains a vertex v with degree $n-1$. Then the neighbourhood of v is an independent set of order $n-1$. Thus if we delete this vertex, we obtain a graph M' with $r-1$ vertices, which contains an independent set of order $n-1$, such that each vertex in this independent set has degree at most $n-2$. If M contains no vertex of degree $n-1$, we delete an arbitrary vertex not contained in some specified independent set of order $n-1$. In each case we obtain an induced subgraph M' of M with $r-1$ vertices, which contains an independent set S of size $n-1$, such that every vertex in S has degree at most $n-2$. From each vertex w in $V(M') \setminus S$ there are at least $s+1$ edges connecting w with an element of S,

otherwise $S \cup \{w\}$ would induce a supergraph of $K_n - K_{1,s}$ in M^c. Hence, there exists a vertex $v \in S$, which has degree at least $\frac{(r-n)(s+1)}{n-1}$. By assumption this quantity is larger than $n-2$, contradicting the choice of M'. Hence our claim follows. □

Let T_{s+} denote the tree obtained from $K_{1,s}$ by adding an extra vertex and connecting it to a vertex with degree 1 in $K_{1,s}$. Let Δ_s be the graph obtained from $K_{1,s}$ by adding one edge connecting two vertices with degree 1 in $K_{1,s}$.

Lemma 5.2. *Let r, n, s be integers such that $1 \leq s < n$ and $(r-n)(s+1) > (n-1)(n-2)$. Then for every triangle-free graph M with r vertices, such that M^c contains $K_n - \Delta_{s+1}$, we have that M^c contains $K_n - T_{s+}$.*

Since T_{s+} is a subgraph of Δ_{s+1}, we have $R(K_3, K_n - \Delta_{s+1}) \leq R(K_3, K_n - T_{s+})$. Together with Lemma 5.2 this gives:

Corollary 5.3. *Let r, n, s be integers such that $1 \leq s < n$ and $(r-n)(s+1) > (n-1)(n-2)$. Then $R(K_3, K_n - \Delta_{s+1}) = R(K_3, K_n - T_{s+})$.*

Lemma 5.4. *Let n, r, s be integers, such that $(r-n+1)s > (n-2)(n-3)$ and $(r-n)(s+1) > (n-1)(n-2)$. Then every graph with r vertices containing $K_{n-1} - e$, also contains $K_n - T_{s+}$.*

Lemma 5.5. *Let n, r, s, t be integers, such that $s+t+2 \leq n$, $s \geq t > 0$, $(r-n)(s+1) > (n-1)(n-2)$ and $(r-(n-1))(s+1) > (n+2(s-t)-2)(n-3)$. Then every graph with r vertices containing $K_{n-1} - e$, also contains $K_n - K_{1,s} - K_{1,t}$.*

Applying these results to the case of graphs of order 10, we obtain the following:

Corollary 5.6.

- *For $2 \leq s \leq 9$, we have $R(K_3, K_{10} - K_{1,s}) = R(K_3, K_9) = 36$.*
- *For $3 \leq s \leq 8$, we have $R(K_3, K_{10} - T_{s+}) = R(K_3, K_{10} - \Delta_{s+1}) = R(K_3, K_{10} - K_{1,s} - e) = R(K_3, K_9 - e) = 31$.*

Proof. The upper bounds follow from the lemmas, while the lower bounds are implied by $R(K_3, K_9) = 36$ and $R(K_3, K_9 - e) = 31$, respectively. □

5.2.3 Generation of maximal triangle-free graphs

Mtf graphs with $n+1$ vertices are generated from mtf graphs with n vertices using the same construction method as in [9], but different isomorphism rejection routines. To describe the construction, we first introduce the concept of *good dominating sets*.

Definition 5.1. *$S \subseteq V(G)$ is a dominating set of G if $S \cup \{N(s) \mid s \in S\} = V(G)$.*

A dominating set S of an mtf graph G is good if after removing all edges with both endpoints in S, the distance from s to v is at most two for every $s \in S$ and $v \in V(G) \setminus S$.

The basic construction operation removes all edges between vertices of a good dominating set S and connects all vertices of S to a new vertex v. This is also illustrated in Figure 5.1. Edges with both endpoints in S are called *internal edges* of S. This basic construction operation is a recursive structure for the class of all mtf graphs [9].

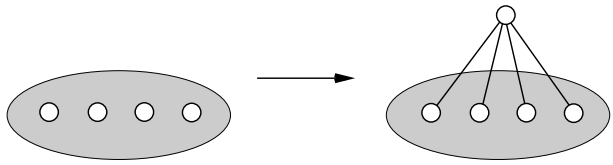


Figure 5.1: The basic construction operation for mtf graphs.

In [9] it was observed that a surprisingly large number of mtf graphs have an automorphism group of order 2. For example more than 33% of the mtf graphs with 21 vertices have an automorphism group of order 2. This is nearly always caused by two vertices with identical neighbourhoods. We exploit this observation to improve the efficiency of the isomorphism rejection routines. To this end we distinguish between 3 types of good dominating sets:

type 0: A set $S = N(v)$ for some $v \in V$. Note that for every vertex v in an mtf graph, the set $N(v)$ is a good dominating set without internal edges.

type 1: A good dominating set S without internal edges, but $S \neq N(v)$ for every vertex v in V.

type 2: A good dominating set S with internal edges.

We also refer to construction operations as *expansions* and call the inverse operations *reductions*. We speak about reductions or expansions of type 0, 1 and 2 if the good dominating sets involved are of this type. If G' is obtained from G by an expansion, we call G' the child of G and G the parent of G'.

We use the canonical construction path method (see Section 1.3) to make sure that only pairwise non-isomorphic mtf graphs are generated. Two reductions of mtf graphs G and G' (which may be identical) are called equivalent if there is an isomorphism from G to G' mapping the vertices that are removed onto each other and inducing an isomorphism of the reduced graphs. Note that a reduction does not only depend on the vertex which is removed, since the graph obtained after removing that vertex is not necessarily mtf (i.e. in case of a reduction of type 2). In that case, one also has to describe which edges must be inserted to make the graph mtf. In order to use the canonical construction path method, we first have to define which of the various possible reductions of an mtf graph G to a smaller mtf graph is the *canonical reduction* of G. This canonical reduction must be uniquely determined up to equivalence. We call the graph obtained by applying the canonical reduction to G the *canonical parent* of G and an expansion that is the inverse of a canonical reduction a *canonical expansion*.

Furthermore, we also define an equivalence relation on the set of possible expansions of a graph G. Note that the expansions are uniquely determined by the good dominating set S to which they are applied. Therefore we define two expansions of G to be equivalent if and only if there is an automorphism of G mapping the two good dominating sets onto each other.

The two rules of the canonical construction path method are:

1. Only accept a graph if it was constructed by a canonical expansion.
2. For every graph G to which construction operations are applied, perform exactly one expansion from each equivalence class of expansions of G.

If we start with K_1 and recursively apply these rules to each graph until the output size is reached (cf. Algorithm 1.1 from Section 1.3), exactly one graph of each isomorphism class of mtf graphs is generated. We refer the reader to [9] for a proof.

For deciding whether or not a reduction is canonical, we use a two-step strategy. First we decide which vertex should be removed by the canonical reduction. In case the graph obtained after removing this vertex is not an mtf graph, we determine the canonical way to insert edges (i.e. the second step). If the graph obtained after removing this vertex is an mtf graph, there is no need to perform

the second step. A 5-tuple $t(v) = (x_0(v), \ldots, x_4(v))$ represents the vertex v which is removed by the reduction in such a way that two vertices have the same 5-tuple if and only if they are in the same orbit of the automorphism group. The canonical reduction is a reduction which removes the vertex with the lexicographically smallest 5-tuple.

The first entry $x_0(v)$ is the type of the neighbourhood of v in the reduced graph (i.e. 0, 1 or 2). The most expensive part in computing the canonical reduction is the computation of how edges have to be inserted between the former neighbours of the removed vertex. Therefore we assigned the lowest priority to operations of type 2. If the graph has vertices with identical neighbourhood, a reduction with $x_0 = 0$ is always possible and other reductions do not have to be considered in order to find the one with minimal 5-tuple. In case there are exactly two vertices with identical neighbourhood, the canonical reduction is already found after this step: no matter how the remaining entries of the 5-tuple are defined, removing one of these two vertices is the canonical reduction as they are the only ones with minimal value of x_0. Furthermore, there is an automorphism exchanging these two vertices and fixing the other vertices, so the two reductions are equivalent and both are canonical.

The way the remaining values are chosen is the result of a lot of performance tests comparing different choices. The value of $x_1(v)$ is $-deg(v)$ of the vertex v that is going to be removed by the reduction in case $x_0(v) \in \{0, 1\}$ and $deg(v)$ in case $x_0(v) = 2$. Furthermore $x_2(v) = -\sum_{w \in N(v)} deg(w)$ and $x_3(v)$ can be described as $-\sum_{w \in N(v)} |V|^{deg(w)}$ (where V denotes the set of vertices of the mtf graph which was constructed). In the program x_3 is in fact implemented as a sorted string of degrees, but it results in the same ordering.

We call a vertex v *eligible* for position j if it is among the vertices for which $(x_0(v), ..., x_{j-1}(v))$ is minimal among all possible reductions. For the first rule of the canonical construction path method we do not have to find the canonical reduction, but only have to determine whether the last expansion producing vertex w is canonical. Therefore each x_i is only computed if the vertex w is still eligible for position i and only for vertices which are eligible for position i. If $x_0(w) \in \{0, 1\}$ and w is the only vertex eligible for position i, we know that the expansion was canonical. If $x_0(w) = 2$, we still have to determine whether the edges that have been removed are equivalent to the edges that would be inserted by a canonical reduction.

If there are also other vertices eligible for position 4, we canonically label the graph G using the program *nauty* [88]. We define $x_4(v)$ to be the negative of

the largest label in the canonical labelling of G of a vertex which is in the same orbit of $\mathrm{Aut}(G)$ as v. The discriminating power of $x_0, ..., x_3$ is usually enough to decide whether or not a reduction is canonical. For generating all mtf graphs with $n = 20$ vertices for example, the more expensive computation of x_4 is only required in 7.8% of the cases. This fraction is decreasing with the number of vertices to e.g. 6.2% for $n = 22$. After computing x_4, the vertex in the canonical reduction is uniquely defined up to isomorphism.

In case $x_0 = 2$, the canonical reduction is not completely determined by the vertex v which is removed by the canonical reduction, as there can be multiple ways to insert the edges in the former neighbourhood of v. In this case we use the same method as Brandt et al. used in [9], which is essentially a canonical choice of a set of edges that can be inserted and give priority to sets of small size. As this routine is not often used, this part has hardly any impact on the computation time: for generating mtf graphs with $n = 18$ vertices, only about 2.5% of the time is spent on the routines dealing with this part and already for $n = 20$ this decreases to 1.5%. Therefore we decided not to develop any improvements for this part and refer the reader to [9] for details.

The priority of the operations expressed in the 5-tuple also allows look-aheads for deciding whether or not an expansion can be canonical before actually performing it. This is also an advantage when constructing good dominating sets for expansion as it often allows to reduce the number of sets that have to be constructed.

A vertex which has the same neighbourhood as another vertex is called a *double vertex*. An mtf graph with double vertices can be reduced by a reduction of type 0. If two vertices have the same neighbourhood in an mtf graph, each good dominating set without internal edges either contains both vertices or none. So after an operation of type 0 or 1, the vertices still have identical neighbourhoods allowing a reduction of type 0. Thus if a graph G contains a reduction of type 0, we do not have to apply expansions of type 1. Furthermore we only have to apply expansions of type 0 to neighbourhoods of vertices v of G for which $deg(v)$ is at least as large as the degree of the canonical double vertex in G, otherwise the new vertex will not have the minimal value of x_1. If G did not contain any double vertices, we have to apply operations of type 0 to the neighbourhoods of all vertices. In this case, we don't have to check canonicity for the graphs constructed.

After a canonical operation of type 2, no reductions of type 0 are possible. So we only have to apply operations of type 2 that make sure that afterwards

no vertices with identical neighbourhoods exist. Therefore the good dominating sets to which an operation of type 2 is applied must contain at least one vertex from the neighbourhood of each double vertex. If no vertex of the common neighbourhood of a pair of double vertices is included, both vertices must be contained in the dominating set themselves (since the expanded graph must have diameter 2). But then they would still have identical neighbourhoods after the operation. Each good dominating set must also contain a vertex from each set of vertices with identical neighbourhood. In the program we use this in its strongest form only if there is just one common neighbourhood, else we use a weaker form. This is not a problem for the efficiency as there is usually only one common neighbourhood.

If a graph has at least 3 vertices which have the same neighbourhood, every graph obtained by applying an expansion of type 2 to G has a reduction of type 0: let v_1, v_2 and v_3 be vertices which have the same neighbourhood. In order to destroy an operation of type 0, the good dominating set S to which an operation of type 2 is applied must contain at least one vertex of the set of vertices with identical neighbourhood. Suppose w.l.o.g. that S contains v_1. If S also contains v_2 (respectively v_3), v_1 and v_2 (respectively v_3) also have identical neighbourhoods in the expanded graph. If S contains neither v_2 nor v_3, v_2 and v_3 have identical neighbourhoods in the expanded graph. So in each case the expanded graph contains a reduction of type 0. Thus we do not have to apply expansions of type 2 to graphs which contain at least 3 vertices which have the same neighbourhood.

The simplified structure of the algorithm which searches for possible expansions and uses these optimisations is listed as pseudocode in Algorithm 5.1.

Due to the choice of x_1, the degree of the vertex which is removed by the reduction is minimal for canonical reductions of type 2. If we apply an operation of type 2 to a good dominating set S, the new vertex v will have degree $|S|$. If the minimum degree of a graph is m, we only have to apply operations of type 2 to good dominating sets of size at most m (or size $m+1$ if the good dominating set contains all vertices of minimum degree).

Recall that we also have to compute the equivalence classes of expansions of a graph in order to comply with the second rule of the canonical construction path method. We use *nauty* to compute the automorphism group of the graph and then compute the orbits of good dominating sets using the generators of the group. In case we know that only an operation of type 0 can be canonical, we actually compute the orbits of vertices representing the good dominating sets

Algorithm 5.1 Find_expansions(graph G, type t)

```
if t == 0 then
  search for expansions of type 0
else if t == 1 then
  if G contains no double vertices then
    search for expansions of type 1
  end if
else
  if G contains no double vertices then
    search for expansions of type 2
  else
    if G contains at least 3 vertices with the same neighbourhood then
      return  no type 2 expansions possible
    else
      search for expansions where the good dominating set contains at least
      one vertex from the neighbourhood of every double vertex
    end if
  end if
end if
```

formed by their neighbourhoods (which is computationally cheaper).

In some cases we do not have to call *nauty* to compute the automorphism group. For example, if G has a trivial automorphism group and we apply an operation of type 0 by inserting a vertex v' with the same neighbourhood as v, the expanded graph G' will have an automorphism group of order 2 generated by the automorphism exchanging v and v' and fixing all other vertices.

Testing and results

We used our program to generate all mtf graphs up to 23 vertices. The number of graphs generated were in complete agreement with the results obtained by running the program from Brandt et al. [9] (which is called *MTF*). The graph counts, running times and a comparison with *MTF* are given in Table 5.2. Our program is called *triangleramsey*. Both programs were compiled by gcc and the timings were performed on an Intel Xeon L5520 CPU at 2.27 GHz. The timings for $|V(G)| \geq 20$ include a small overhead due to parallelisation. We can conclude from Table 5.2 that *triangleramsey* is significantly faster than *MTF* for generating

mtf graphs.

Table 5.3 gives an overview how many graphs are constructed by canonical operations of the different types. This table shows that operations of type 2 are by far the least common canonical operations.

$\|V(G)\|$	number of graphs	MTF (s)	triangleramsey (s)	speedup
17	164 796	4.0	0.8	5.00
18	1 337 848	30.5	6.2	4.92
19	13 734 745	315	67	4.70
20	178 587 364	4 390	972	4.52
21	2 911 304 940	75 331	17 109	4.40
22	58 919 069 858	1 590 073	373 417	4.26
23	1 474 647 067 521	40 895 299	10 431 362	3.92

Table 5.2: Counts and generation times for mtf graphs.

number of vertices	number of mtf graphs	num. generated by an operation of type 0	num. generated by an operation of type 1	num. generated by an operation of type 2
4	2	2	0	0
5	3	2	0	1
6	4	4	0	0
7	6	6	0	0
8	10	9	0	1
9	16	15	0	1
10	31	29	1	1
11	61	57	3	1
12	147	139	4	4
13	392	368	15	9
14	1 274	1 183	75	16
15	5 036	4 595	391	50
16	25 617	22 889	2 420	308
17	164 796	142 718	19 577	2 501
18	1 337 848	1 105 394	213 743	18 711
19	13 734 745	10 674 672	2 855 176	204 897
20	178 587 364	129 333 325	46 244 514	3 009 525

Table 5.3: The number of mtf graphs which were generated by operations of each type.

5.2.4 Generation of Ramsey graphs

The construction operations for mtf graphs never add edges between vertices of the parent and thus never remove edges from the complement. So, if G is contained in the complement of an mtf graph M, G will also be contained in the complement of all descendants of M. If M is not a Ramsey graph for G, its descendants won't be Ramsey graphs. This means we can prune the generation process when an mtf graph is generated which is not a Ramsey graph for G.

The same pruning was already used by Brandt et al. in [8]. Nevertheless, we optimised our algorithm which tests if G is contained in the complement of an mtf graph for dense graphs as the graphs with 10 vertices whose Ramsey number could not be determined in [8] are all very dense. These optimisations are described in the remainder of this section.

For a graph G and an mtf graph M the following criteria are equivalent:

(i) G is subgraph of M^c

(ii) M contains a spanning subgraph of G^c as an induced subgraph

If G is dense, G^c has relatively few edges and therefore in this case it is easier to test (ii) instead of (i). By just applying this simple algorithm, even with the faster generator we were not able to go much further than Brandt et al. did in [8]. Therefore we designed and applied several optimisations specifically for dense test graphs G. These optimisations are crucial for the efficiency of the algorithm.

The bottleneck of the algorithm is the procedure which tests if the generated mtf graphs contain a spanning subgraph of G^c as induced subgraph. This procedure basically constructs all possible sets with $|V(G)|$ vertices and an upper bound of $|E(G^c)|$ on the number of edges and tests for each set if the graph induced by this set is a subgraph of G^c. If none of these sets induces a spanning subgraph of G^c, the current mtf graph is a Ramsey graph for G. Various bounding criteria are used to avoid the construction of sets which cannot be a subgraph of G^c. The simplified structure of this algorithm (where some optimisations are omitted to improve the readability) is sketched as pseudocode in Algorithm 5.2. In the pseudocode $M[S]$ denotes the subgraph of M induced by the set of vertices $S \subseteq V(M)$. In the actual implementation we represent S and P by bitvectors.

To test if S induces a spanning subgraph of G^c, we first use some cheap heuristics. For example, if the graph induced by S contains more vertices with degree at least 1 than G^c, it cannot be a spanning subgraph of G^c. If these cheap heuristics are passed, we try to map the vertices of the graph induced by S to

the vertices of G^c in all possible ways. The vertices are mapped in descending order of their degrees. This is a simple algorithm, but it is not a bottleneck.

Algorithm 5.2 Contains_spanning_subgraph(mtf graph M, test graph G, set S, possible vertices P)

```
if S contains |V(G)| elements then
  test if S induces a spanning subgraph of G^c
  return result
end if
P' := P
for every v ∈ P' do
  remove v from P'
  add v to S
  if |E(M[S])| ≤ |E(G^c)| and Δ(M[S]) ≤ Δ(G^c) then
    if Contains_spanning_subgraph(M, G, S, P') then
      return true
    end if
  end if
  remove v from S
end for
return false
```

If the algorithm as described so far is applied and the order of the mtf graphs is sufficiently large, by far most of the mtf graphs that are generated are rejected as they turn out to be no Ramsey graphs for the test graph G. For example for $G = K_{10} - P_5$ and $|V(M)| = 28$ (without other optimisations) approximately 99% of the mtf graphs which were generated are no Ramsey graphs (and are thus rejected). So most of the tests for making spanning induced subgraphs give a positive result – that is: there is an induced subgraph of M that is a spanning subgraph of G^c. We take this into account by first using some heuristics to try to find a set of vertices which is a spanning subgraph of G^c quickly. If such a set is found, we can abort the search.

More specifically: when an mtf graph is rejected because it is not a Ramsey graph for G, we store the set of vertices which induces a spanning subgraph of G^c. For each order n, we store up to 100 sets of vertices which caused an mtf graph with n vertices to be rejected. When a graph with n vertices is generated, we first investigate if one of those 100 sets of vertices induces a spanning subgraph of G^c. Only if this is not the case, we continue the search. Experimental results

showed that storing 100 sets seems to be a good compromise between cost to test if a set induces a spanning subgraph of G^c and the chance to have success. Without other optimisations this already makes the program 5 times faster for e.g. $G = K_{10} - P_5$ and $|V(M)| = 26$.

The second step in trying to prove that M is not a Ramsey graph is a greedy heuristic. We construct various sets of $|V(G)|$ vertices which have as few common neighbours as possible. These sets are good candidates to induce a subgraph of G^c. Only if none of these sets induces a subgraph of G^c, we have to continue to investigate the graph. This results in an additional speedup of approximately 10%.

These heuristics allow to find a set of vertices which induces a spanning subgraph of G^c quickly in about 98% of the cases. If these heuristics did not yield such a set of vertices, we start a complete search. In about 70% of the cases the graphs passing the heuristic search are actually Ramsey graphs for G. The coarse pseudocode of the procedure which tests if an mtf graph M is a Ramsey graph for G is given in Algorithm 5.3.

Algorithm 5.3 Is_Ramsey_graph(mtf graph M, test graph G)

```
for each stored set S with n = |V(M)| do
  if S induces a spanning subgraph of G^c in M then
    return M is not a Ramsey graph for G
  end if
end for
construct sets of |V(G)| vertices in a greedy way
if set found which induces a spanning subgraph of G^c in M then
  store set
  return M is not a Ramsey graph for G
end if
construct all possible sets of |V(G)| vertices // cf. Algorithm 5.2
if set found which induces a spanning subgraph of G^c in M then
  store set
  return M is not a Ramsey graph for G
else
  return M is a Ramsey graph for G
end if
```

The construction of all possible sets of $|V(G)|$ vertices can also be improved. Since most of the graphs passing the heuristical search are actually Ramsey

graphs for G, we try to construct sets of $|V(G)|$ in a well-defined order rather than in a random order. More specifically, we first try to add the vertices which have the largest degrees and most vertices in common with the vertices in the current set of vertices S to S. This allows to bound earlier in the recursion of Algorithm 5.2.

Also recall that our algorithm constructs Ramsey graphs from smaller Ramsey graphs. Therefore if an mtf graph M was constructed by operations of type 0 or 1 (i.e. no edges were removed), we only have to investigate sets of vertices which contain the new vertex which was added by the construction. The subgraphs induced by the other sets did not change and are already proven not to induce a spanning subgraph of G^c. Moreover if M was constructed by an operation of type 0, we only have to investigate sets of vertices which contain the new vertex and all other vertices which have the same neighbourhood as the new vertex. Since if a set does not contain a vertex v which has the same neighbourhood as the new vertex, we can swap v and the new vertex.

Similarly, if M was constructed by an operation of type 2 and one edge $\{v_1, v_2\}$ was removed, we only have to investigate sets of vertices which contain the new vertex or which contain both v_1 and v_2. Similar optimisations can also be used when more edges are removed, but this does not speed up the program as in most cases such operations turn out to be not canonical. In these cases the graph is already rejected before it is tested whether or not this graph is a Ramsey graph.

We also try to avoid constructing mtf graphs that are no Ramsey graphs for G in advance. This is of course even better than efficiently rejecting graphs after they are constructed. More specifically, each time a new mtf Ramsey graph M for a graph G was constructed, we search and store *approximating* sets of vertices. We call a set of vertices *approximating* if it induces a spanning subgraph of G^c_δ, where G^c_δ is a graph obtained by removing a vertex of minimum degree from G^c. For most graphs G with 10 vertices whose Ramsey number could not be determined in [8], G^c has minimum degree 0.

If for a graph M' which is constructed from M there is an approximating set S of M for which no vertex $s \in S$ is a neighbour of the new vertex v, the graph induced by $S \cup \{v\}$ in M' is a spanning subgraph of G^c. So graphs constructed from M can only be Ramsey graphs if the good dominating set of M contains at least one vertex from each approximating set in M. Since searching for all approximating sets is expensive, we search for them during the search for sets of vertices which induce a spanning subgraph of G^c: when a set of $|V(G^c_\delta)|$ vertices was formed, we store it. We store up to 300 such sets, but only store them if they

are sufficiently distinct from the sets which were already stored. If it turns out that the graph which was under investigation is a Ramsey graph, we test the list of stored sets and determine the approximating sets among them. On average this optimisation avoids the construction of more than 90% of the children.

5.2.5 Computation of Ramsey numbers

To determine Ramsey numbers with our algorithm, we also use the same basic strategy as Brandt et al. used in [8] (combined with several new optimisations):

Assume we have a list of all graphs G with Ramsey number $R(K_3, G) \geq r$. We want to split this list into those with $R(K_3, G) = r$ and those with $R(K_3, G) > r$. We have a (possibly empty) list of MAXGRAPHs, these are graphs which have Ramsey number r. We also have a (possibly empty) list of RAMSEYGRAPHs, which are triangle-free graphs with r vertices which are (or might be) Ramsey graphs for some of the remaining graphs.

The procedure to test whether the remaining graphs have Ramsey number r or at least $r + 1$ works as follows:

for $k = \binom{n}{2}$ **downto** $n - 1$ **do**
  for every connected graph G with k edges in the list **do**
    if G is not contained in any MAXGRAPH **then**
      if G is contained in the complement of every RAMSEYGRAPH **then**
        if *triangleramsey* applied to G finds a Ramsey graph of order r **then**
          add this Ramsey graph to the list of RAMSEYGRAPHs
          $R(K_3, G) > r$
        else
          add G to the list of MAXGRAPHs
          $R(K_3, G) \leq r$
        end if
      else
        $R(K_3, G) > r$
      end if
    else
      $R(K_3, G) = r$
    end if
  end for
end for

The graphs are tested in descending order of edges. So if $G' \subset G$, then G is tested before G'. If G has Ramsey number $R(K_3, G) = r$, all subgraphs of G

which are in the list of graphs with Ramsey number at least r also have Ramsey number r. The algorithm used in this procedure to test if G is contained in the complement of an mtf graph M, is the same as the algorithm used in *triangleramsey* to test if M contains a spanning subgraph of G^c as induced subgraph. The results from the subgraph tests used in this procedure were independently confirmed by performing the same subgraph tests using programs from Brandt et al. [8].

For large orders of r (i.e. $r \geq 26$), the bottleneck of the procedure is the computation of individual Ramsey graphs by *triangleramsey*. Here we used some additional optimisations: if r is close to $R(K_3, G)$, there are usually only very few Ramsey graphs of order r for G. Therefore for certain graphs G where we expected r to be close to $R(K_3, G)$, we used *triangleramsey* to compute all Ramsey graphs of order r for G, instead of aborting the program as soon as one Ramsey graph was found. *Triangleramsey* constructs Ramsey graphs from smaller Ramsey graphs, so in order to construct all Ramsey graphs for G of order $r+1$, we can start the program from the Ramsey graphs of order r. This avoids redoing the largest part of the work. Of course this approach only works if there are not too many Ramsey graphs of order r to be stored. We used this strategy amongst others to generate all Ramsey graphs of order 28 for $K_{10} - P_5$ and $K_{10} - 2P_3$ (where P_x stands for the path with x vertices). Computing all Ramsey graphs with 28 vertices for $K_{10} - 2P_3$, for example, required almost 4 CPU years. This yielded 7 Ramsey graphs and constructing the Ramsey graphs with 29 vertices from these 7 graphs took less than 2 seconds.

Let H be a subgraph of G for which we know that $R(K_3, H) \geq r$. If we have the list of all Ramsey graphs of order r for G and none of these Ramsey graphs is a Ramsey graph for H, we know that $R(K_3, H) = r$. Amongst others, this allowed us to determine that several subgraphs of $K_{10} - P_5$ and $K_{10} - 2P_3$ have Ramsey number 28.

5.2.6 Testing and results

By using the algorithm described in Section 5.2.5, we were able to compute all Ramsey numbers $R(K_3, G) = r$ for connected graphs of order 10 for which $r \leq 30$ and to determine which Ramsey graphs have Ramsey number larger than 30. Since *triangleramsey* is more than 20 times faster than *MTF* for generating triangle Ramsey graphs of large order r, we could only compute Ramsey numbers up to $r = 26$ with the original version of *MTF*. In order to be able to compare our results for larger r, we added several of the optimisations which were described

in Section 5.2.4 to *MTF*. With this improved version of *MTF* we were also able to determine all Ramsey numbers up to $r = 30$ and to determine which graphs have Ramsey number larger than 30 (though this optimised version of *MTF* still required significantly more CPU-time than *triangleramsey*). All results were in complete agreement. In the cases where we generated all Ramsey graphs of order r for a given test graph, the results obtained by *triangleramsey* and *MTF* were also in complete agreement. For each Ramsey graph which was generated, we also used an independent program to confirm that the Ramsey graph does not contain G in its complement.

There are 34 connected graphs G for which $R(K_3, G) > 30$. In Section 5.2.2, we proved that $R(K_3, K_{10} - T_{3+}) = R(K_3, K_{10} - K_{1,3} - e) = 31$. So the graphs with $R(K_3, G) > 30$ which are a subgraph of $K_{10} - T_{3+}$ or $K_{10} - K_{1,3} - e$ also have Ramsey number 31. In that section, we also proved that $R(K_3, K_{10} - K_{1,s}) = 36$ (for $2 \leq s \leq 9$). This leaves us with 10 connected graphs G with $R(K_3, G) > 30$ for which we were unable to determine their exact Ramsey number. The complements of these 10 remaining graphs are depicted in Figure 5.2.

Actually we have some additional knowledge about most of these graphs: if G^c contains a triangle, an mtf graph cannot contain G^c as induced subgraph. So if G^c contains a triangle and H^c is the only graph which can be obtained by removing an edge from that triangle of G^c, then $R(K_3, G) = R(K_3, H)$. Thus among the 10 remaining graphs, $R(K_3, K_{10} - K_3 - e) = R(K_3, K_{10} - P_3 - e)$ and $R(K_3, K_{10} - K_4) = R(K_3, K_{10} - K_4^-)$ (where K_4^- stands for K_4 with one edge removed). It is also known that $40 \leq R(K_3, K_{10}) \leq 42$ and $37 \leq R(K_3, K_{10} - e) \leq 38$ (see Section 5.3 and [99, 59] for more information). Seven of the ten remaining graphs are subgraphs of $K_{10} - P_3$ and since it follows from the theoretical results from Section 5.2.2 that $R(K_3, K_{10} - P_3) = 36$, the Ramsey number of these seven graphs is at most 36.

Table 5.4 contains the number of connected graphs G of order 10 which have $R(K_3, G) = r$. Since no connected graph of order n can be contained in the complement of $K_{n-1,n-1}$, the smallest possible Ramsey number of a connected graph of order n is at least $2n - 1$. The triangle Ramsey numbers of connected graphs of order 10 are given in Appendix B.

Previously only the Ramsey numbers for disconnected graphs of order at most 8 were known (see [13]). We independently verified these results for order 8 and also determined all Ramsey numbers for disconnected graphs of order 9 and 10. These results are listed in Table 5.5. The Ramsey numbers smaller than 28 were obtained computationally. We also independently confirmed these

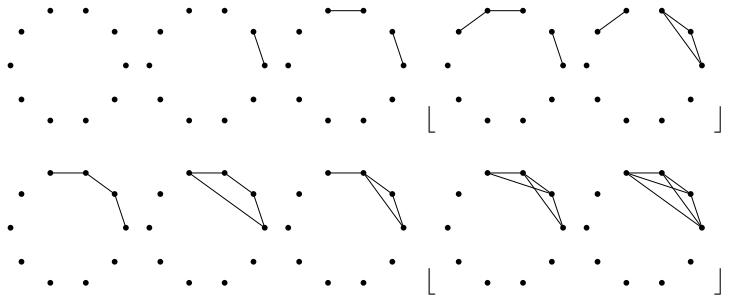


Figure 5.2: The complements G^c of the 10 remaining graphs which have $R(K_3, G) > 30$ for which we were unable to determine the Ramsey number. Graphs which must have the same Ramsey number are grouped by $\lfloor$ and $\rfloor$.

computational results by using *MTF*. The other Ramsey numbers were obtained by some simple reasoning. More specifically, if a disconnected graph is the union of 2 connected graphs G_1 and G_2 and $R(K_3, G_1) - |V(G_1)| \geq R(K_3, G_2)$, then $R(K_3, G_1 \cup G_2) = R(K_3, G_1)$.

Unfortunately, we did not succeed to compute new values of the functions $f(), g()$ and $h()$ given in Table 2 of [8]. Nevertheless we could confirm all the values given in Table 2 of [8] with *triangleramsey*.

5.2.7 Closing remarks

Since all computational results were independently obtained by both *MTF* and *triangleramsey*, the chance of incorrect results caused by errors in the implementation is extremely small.

We believe that specialised algorithms and/or new theoretical results will be required to determine the triangle Ramsey number of the 10 remaining graphs. It is likely that the specialised algorithms from Section 5.3 which are used to obtain better upper bounds for classical Ramsey numbers can also be adapted and applied to obtain better bounds for some of the remaining graphs.

As not only the number of graphs is increasing very fast, but also the difficulty to determine Ramsey numbers, it is very likely that this is the last list of Ramsey numbers that can be completed for a very long time. Therefore we hope that the challenge of determining the Ramsey number of the 10 remaining graphs will be taken up by the mathematical community.

As human intuition and insight is often based on examples, data about small

	$\|G\| = 3$	$\|G\| = 4$	$\|G\| = 5$	$\|G\| = 6$	$\|G\| = 7$	$\|G\| = 8$	$\|G\| = 9$	$\|G\| = 10$
$r = 5$	1							
$r = 6$	1							
$r = 7$		5						
$r = 8$								
$r = 9$		1	18					
$r = 10$								
$r = 11$			2	98				
$r = 12$				6				
$r = 13$				2	772			
$r = 14$			1	4	40			
$r = 15$						9 024		
$r = 16$					13	1 440		
$r = 17$				1	19	498	242 773	
$r = 18$				1	7	119	16 024	
$r = 19$							311	10 101 711
$r = 20$								504
$r = 21$					1	28	1 809	1 602 240
$r = 22$							22	3 155
$r = 23$					1	6	98	6 960
$r = 24$								
$r = 25$						1	26	1 384
$r = 26$							5	316
$r = 27$							3	92
$r = 28$						1	7	142
$r = 29$								30
$r = 30$								3
$r = 31$							1	16 + ?
$r = 36$							1	8 + ?

Table 5.4: Number of connected graphs G with Ramsey number $R(K_3, G) = r$. Note that the 10 graphs with $R(K_3, G) > 30$, for which we were unable to determine the Ramsey number are not included in the table.

graphs, like complete lists of Ramsey numbers, can help to discover mathematical theorems, suggest conjectures and give insight into the structure of mathematical problems. An example is given in [16], where a large amount of computational data about *alpha*-labellings gave insight into the structure of *alpha*-labellings of trees so that new theorems could be proven and some unexpected conjectures

	$\|G\|=3$	$\|G\|=4$	$\|G\|=5$	$\|G\|=6$	$\|G\|=7$	$\|G\|=8$	$\|G\|=9$	$\|G\|=10$
$r=3$	2							
$r=4$		2						
$r=5$		2	4					
$r=6$		1	3	7				
$r=7$			5	11	18			
$r=8$				3	5	23		
$r=9$			1	20	50	60	83	
$r=10$						36	68	151
$r=11$				2	102	225	427	596
$r=12$					6	12	144	168
$r=13$					2	776	1 552	3 734
$r=14$				1	6	52	107	447
$r=15$							9 024	18 048
$r=16$						13	1 466	2 933
$r=17$					1	21	540	243 856
$r=18$					1	9	137	16 301
$r=19$								311
$r=20$								
$r=21$						1	30	1 869
$r=22$								22
$r=23$						1	8	114
$r=24$								
$r=25$							1	28
$r=26$								5
$r=27$								3
$r=28$							1	9
$r=31$								1
$r=36$								1

Table 5.5: Number of disconnected graphs G with Ramsey number $R(K_3, G) = r$.

were suggested. In order not to be misled by too small examples, it is important to have as much data as possible to develop a good intuition. This is confirmed by the following example: if $K_n - m \cdot e$ denotes the graph obtained by removing m disjoint edges from K_n, then the previously existing lists of triangle Ramsey numbers for graphs of order at most 9 have the property that for fixed n, the values of $R(K_3, K_n - m \cdot e)$ are the same for all $2 \leq m \leq n/2$. This may be considered as a hint that it could be true in general. But as the list in Appendix B

shows, this equation does not hold for $n = 10$: $R(K_3, K_{10} - m \cdot e) = 30$ for all $(3 \leq m \leq 5)$, while $R(K_3, K_{10} - 2 \cdot e) > 30$. The complements of these graphs are also depicted in Figure 5.3.

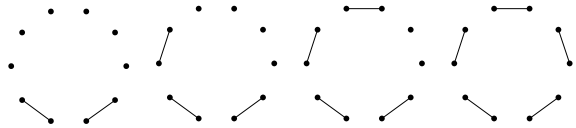


Figure 5.3: The complements of $K_{10} - m \cdot e$ $(2 \leq m \leq 5)$. $R(K_3, K_{10} - m \cdot e) = 30$ for all $(3 \leq m \leq 5)$, while $R(K_3, K_{10} - 2 \cdot e) > 30$.

Another interesting observation in the list of Ramsey numbers in Appendix B is that while for $7 \leq n \leq 9$ the graph $K_n - P_5$ has a smaller Ramsey number than $K_n - 2P_3$, they have the same Ramsey number (i.e. 29) for $n = 10$.

These observations are in a certain sense similar to the results obtained with the new lists of snarks from Chapter 3 which yielded counterexamples to several conjectures. These conjectures hold for snarks with less than 34 vertices, so this might have suggested that they also hold in general, whereas this is not always the case. Thus it is important to have as much data as possible.

The latest version of *triangleramsey* can be downloaded from [21]. The list of the Ramsey graphs used in this research can be obtained from *House of Graphs* [15] by searching for the keywords "ramsey * order 10" and the Ramsey numbers can be obtained from [22].

5.3 Classical triangle Ramsey numbers

In this section we describe several specialised techniques and algorithms which we used to determine new upper bounds for classical Ramsey numbers.

5.3.1 Definitions and preliminaries

A $(k, l; n, e)$-*graph* is a K_k-free graph with n vertices and e edges with independence number $\alpha(G) < l$. We denote the set of all $(k, l; n, e)$-graphs by $\mathcal{R}(k, l; n, e)$. We omit the parameter e, or both e and n, or give a range to either of these parameters, when referring to special $(k, l; n, e)$-graphs or unions of $\mathcal{R}(k, l; n, e)$. For example, a (k, l)-graph is a $(k, l; n, e)$-graph for some n and e, and the set $\mathcal{R}(3, 9; 35, \leq 139)$ consists of all triangle-free graphs with 35 vertices and $\alpha(G) \leq 8$

with at most 139 edges (later we will prove that this set is empty). A $(k,l;n)$-graph is called *critical* for (k,l) if $n = R(k,l) - 1$.

Let $e(k,l,n)$ denote the minimum number of edges in any $(k,l;n)$-graph (or ∞ if no such graph exists). The sum of the degrees of all neighbours of v in G is denoted by $Z_G(v)$ (or $Z(v)$ if G is fixed), i.e.

$$Z(v) = Z_G(v) = \sum_{\{u,v\}\in E(G)} deg(u). \tag{1}$$

Let G be a $(3,k;n,e)$-graph. For any vertex $v \in V(G)$, G_v denotes the graph induced in G by the set $V(G) \setminus (N(v) \cup \{v\})$. G_v is a $(3,k-1)$-graph, since if G_v contained an independent set S of order $k-1$, the set $S \cup \{v\}$ would be an independent set of order k in G. Thus if $d = deg_G(v)$, G_v is a $(3,k-1;n-d-1,|E(G)| - Z_G(v))$-graph. If $\gamma(v,k,G)$ (or $\gamma(v)$ if k and G are fixed) is defined as $e - Z_G(v) - e(3,k-1,n-d-1)$, this implies that:

$$\gamma(v) \geq 0, \tag{2}$$

where $\gamma(v)$ is the so called *deficiency* of vertex v (see [61]). Finally, the deficiency $\gamma(G)$ of a graph G is defined as $\sum_{v\in V(G)} \gamma(v,k,G)$. Since all $\gamma(v,k,G)$'s are non-negative, this implies:

$$\gamma(G) \geq 0. \tag{3}$$

The condition that $\gamma(G)$ is non-negative is often sufficient to derive good lower bounds for $e(k,l,n)$, though sometimes the stronger condition that all $\gamma(v,k,G)$'s in $\sum_{v\in V(G)} \gamma(v,k,G)$ have to be non-negative implies even better bounds. $\gamma(G)$ can be computed from the degree sequence of G [61, 63]. If a $(3,k;n,e)$-graph G has n_i vertices of degree i, then:

$$\gamma(G) = ne - \sum_{i=0}^{k-1} n_i\big(i^2 + e(3,k-1,n-i-1)\big) \geq 0, \tag{4}$$

where $n = \sum_{i=0}^{k-1} n_i$ and $2e = \sum_{i=0}^{k-1} i n_i$.

5.3.2 Summary of prior and new results

In 1995, Kim [78] obtained a breakthrough result by proving that $R(3,k) = \Theta(n^2/\log n)$ using probabilistic arguments. For a good overview of the results related to the asymptotic behaviour of $R(3,k)$, we refer the reader to [115]. But,

as already mentioned in Section 5.1, even though the asymptotic behaviour of $R(3,k)$ is known, computing the exact values of $R(3,k)$ remains an extremely difficult problem. Therefore only a few exact results are known.

In 1955 Greenwood and Gleason [62] proved that $R(3,4) = 9$ and $R(3,5) = 14$. Kéry [77] proved that $R(3,6) = 18$ in 1964. A few years later Kalbfleisch [75] established that $R(3,7) \geq 23$ and $R(3,9) \geq 36$ in his Ph.D. thesis. In 1968 Graver and Yackel [61] established that $R(3,7) \leq 23$, thus proving that $R(3,7) = 23$. In 1982 Grinstead and Roberts [63] proved that $R(3,9) = 36$ and established a lower bound of 28 for $R(3,8)$. Ten years later McKay and Zhang [94] proved that $R(3,8) = 28$. The exact value of $R(3,k)$ for $k \geq 10$ is still unknown. Exoo [47] established a lower bound of 40 for $R(3,10)$ and Radziszowski and Kreher [101] proved that $R(3,10) \leq 43$. In this chapter we improve this upper bound to 42. These results are also summarised in Table 5.6. The known values and bounds for $R(m,n)$ for $m > 3$ or $n > 10$ can be found in [99].

Most of the results from Table 5.6 were obtained by using knowledge about $e(3,k,n)$. We also use this direction here: we compute new exact values of $e(3,k,n)$ in several cases and give improved lower bounds for many other cases, which in turn permits us to prove new upper bounds for $R(3,k)$ for $k = 10, 11, 13, 14$, 15 and 16. Likely, more new upper bounds could also be obtained for $k \geq 17$, but we did not perform these computations.

k	$R(3,k)$	year	reference (upper / lower bound)
3	6	-	Folklore
4	9	1955	Greenwood and Gleason [62]
5	14	1955	Greenwood and Gleason [62]
6	18	1964	Kéry [77]
7	23	1966 / 1968	Kalbfleisch [75] / Graver and Yackel [61]
8	28	1982 / 1992	Grinstead and Roberts [63] / McKay and Zhang [94]
9	36	1966 / 1982	Kalblfeisch [75] / Grinstead and Roberts [63]
10	40-43	1989 / 1988	Exoo [47] / Radziszowski and Kreher [101]

Table 5.6: Known values and bounds for $R(3,k)$.

There are still several other open problems related to $R(3,k)$. For example Erdős and Sós [46] conjectured that:

Conjecture 5.7 (Erdős and Sós [46])**.** *$R(3,n+1) - R(3,n) \to \infty$, for $n \to \infty$.*

They also asked to prove or disprove that $R(3,n+1) - R(3,n) = o(n)$. Both problems are still open. Erdős has formulated various other problems related to

Ramsey numbers which are still unsolved.

General formulas for $e(3,k,n)$ are known for all $n \le 13k/4 - 1$ and for $n = 13k/4$ when $k = 0 \bmod 4$. Theorem 5.8 is a cumulative summary of various contributions [61, 63, 100, 101, 102]. It captures many of the small cases, for example it gives the exact values of $e(3,9,n)$ for all $n \le 26$, of $e(3,10,n)$ for $n \le 28$, and of $e(3,13,n)$ for all $n \le 39$.

Theorem 5.8 ([100, 102]). *For all $n, k \ge 1$, for which $e(3,k+1,n)$ is finite,*

$$e(3,k+1,n) = \begin{cases} 0 & \text{if } n \le k, \\ n-k & \text{if } k < n \le 2k, \\ 3n-5k & \text{if } 2k < n \le 5k/2, \\ 5n-10k & \text{if } 5k/2 < n \le 3k, \\ 6n-13k & \text{if } 3k < n \le 13k/4-1. \end{cases}$$

Furthermore, $e(3,k+1,n) = 6n-13k$ for $k = 4t$ and $n = 13t$, and the inequality $e(3,k+1,n) \ge 6n-13k$ holds for all n and k. All critical graphs have been characterized whenever the equality in the theorem holds for $n \le 3k$.

In 2001 Lesser [81] stated in her thesis the theorem that $e(3,k+1,n) \ge (40n-91k)/6$ (which is better than $e(3,k+1,n) \ge 6n-13k$ for larger parameters) and a number of other improvements and characterisations of graphs realising a specific number of edges. These improvements are credited to an unpublished manuscript of her supervisor Backelin [3]. As of 2012, Backelin's manuscript already exceeds 500 pages and it contains numerous additional related results [3, 4], but it still needs more work before it can be published. Therefore we will not rely on these unpublished results in the remainder of this chapter. However, in several places we will cite the bounds obtained there for reference.

In summary, the behaviour of $e(3,k+1,n)$ is known for $n \le 13k/4 - 1$. It seems regular but very difficult to deal with for n slightly larger than $13k/4$. And it becomes hopelessly hard for even larger n.

In the remainder of this chapter we apply computational techniques to establish lower bounds on $e(3,k,n)$ for larger n, for $k \le 15$. Our results immediately imply better upper bounds for $R(3,k)$ in several cases. We also hope that they may contribute to further progress in understanding the general behaviour of $e(3,k,n)$.

The complete enumeration of the sets $\mathcal{R}(3,k)$ with $k \le 6$ was established in [100, 94]. The exact values of $e(3,7,n)$ were determined by Radziszowski and Kreher in [100] and the values of $e(3,8,\le 26)$ in [101]. In this chapter,

we determine the last missing value for $\alpha(G) < 8$, namely $e(3, 8, 27) = 85$. We also match or improve all lower bounds for $e(3, k, n)$ from Radziszowski and Kreher [101]. Lesser's thesis [81] contains many lower bounds for $e(3, k, n)$ which are better than those in [101]. We match or improve these lower bounds in all cases for $k \leq 10$. For $k \geq 11$ and n slightly exceeding $13k/4 - 1$, Lesser's bounds (which are in part credited to Backelin's unpublished manuscript [3]) are better than our bounds in some cases. However we obtain significantly better bounds for larger n.

Our general method is to compute the exact value of $e(3, k, n)$ for concrete k and n, if this is feasible. Else we derive a lower bound for $e(3, k, n)$ by using a combination of inequalities (2), (3) and (4), and computations. Better lower bounds for $e(3, k-1, m)$ for $m = n-d-1$ and various d, lead in general to better lower bounds for $e(3, k, n)$. If we manage to show that $e(3, k, n) = \infty$, i.e. no $(3, k; n)$-graph exists, then we obtain an upper bound $R(3, k) \leq n$. An additional specialised algorithm was needed to establish $R(3, 10) \leq 42$.

Section 5.3.3 describes the extension algorithms which we used to exhaustively construct all $(3, k; n, e)$-graphs for several n and e, for $k \leq 10$. The details of these algorithms are presented in Section 5.3.4. This leads to many new lower bounds for $e(3, k, n)$ and the enumeration of all $(3, k; n)$-graphs for which the number of edges is equal to or a little larger than $e(3, k, n)$ for $k \leq 10$. These results are presented in Section 5.3.5 (and Appendix C) and are then used in Section 5.3.6 to prove that there is only one critical graph for the Ramsey number $R(3, 9)$. It is known that $40 \leq R(3, 10) \leq 43$ (see Table 5.6). In Section 5.3.6 we establish that $R(3, 10) = 43$ if and only if $e(3, 10, 42) = 189$, or equivalently: that $R(3, 10) = 43$ if and only if every critical graph in this case is 9-regular. Then, using computations, we prove in Section 5.3.7 that the latter do not exist and thus show that $R(3, 10) \leq 42$. In Section 5.3.10 we describe how we tested the correctness of our programs.

Finally, in Section 5.3.8 we describe the second stage of our computations which imply many new lower bounds for $e(3, \geq 11, n)$. This stage only uses degree sequence analysis of potential $(3, k; n, e)$-graphs which have to satisfy inequality (4). This in turn leads to the new upper bounds for the Ramsey numbers which are marked in bold in Table 5.7. This table presents the values and best bounds for the Ramsey numbers $R(3, k)$ for $k \leq 16$. All of our new bounds improve the previously best known bounds (listed in the 2011 revision of Radziszowski's dynamic survey on small Ramsey numbers [99]) by one. All references to the lower bounds and the previous upper bounds can be found in that survey. In

November 2012, Exoo [49] improved the lower bound of $R(3, 11)$ by one to 47. The bound $R(3, 16) \leq 98$ was also obtained by Backelin in 2004, though it was not published [3, 4]. In Section 5.3.9 we describe how we attempted to improve lower bounds for $R(3, k)$, but unfortunately this did not lead to any new bounds.

k	$R(3,k)$	k	$R(3,k)$
3	6	10	40–**42**
4	9	11	47–**50**
5	14	12	52–59
6	18	13	59–**68**
7	23	14	66–**77**
8	28	15	73–**87**
9	36	16	79–**98**

Table 5.7: Values and best known bounds for Ramsey numbers $R(3, k)$, for $k \leq 16$.

5.3.3 Methods for computing $e(3, k, n)$

We use three methods to compute $e(3, k, n)$: the maximum triangle-free method, the minimum degree extension method and the neighbourhood gluing extension method. These methods are described in the following.

Maximum triangle-free method

One method to determine $e(3, k, n)$ is by first generating all *maximal* triangle-free (in short: mtf) $(3, k; n)$-graphs. Recall from Section 5.2 that there exists a $(3, k; n)$-graph if and only if there is an mtf $(3, k; n)$-graph. We use the algorithm from Section 5.2.4 to generate all mtf $(3, k; n)$-graphs efficiently (our implementation of this algorithm is called *triangleramsey*). Using this algorithm, it is much easier to generate all mtf $(3, k; n)$-graphs instead of all $(3, k; n)$-graphs, because the number of the former is in most cases much smaller. For example, there are 477 142 $(3, 8; 27)$-graphs, but only 21 798 of them are mtf. Table 5.8 shows the percentage of mtf Ramsey graphs for $R(3, 6)$. By recursively removing edges in all possible ways from these mtf $(3, k; n)$-graphs and testing if the resulting graphs G still satisfy $\alpha(G) < k$, the complete set $\mathcal{R}(3, k; n)$ can be obtained. We use isomorphism rejection by lists (see Section 1.3) to make sure no isomorphic copies are output.

$\|V(G)\|$	Ramsey graphs	mtf Ramsey graphs	percentage mtf
7	100	5	5.00
8	356	8	2.25
9	1 407	10	0.71
10	6 657	18	0.27
11	30 395	28	0.09
12	116 792	50	0.04
13	275 086	74	0.03
14	263 520	108	0.04
15	64 732	96	0.15
16	2 576	38	1.48
17	7	2	28.57

Table 5.8: Counts of Ramsey and mtf Ramsey graphs for $R(3,6)$ with at least 7 vertices.

In 1992 McKay and Zhang [94] proved that $R(3,8) = 28$, but the complete set of critical graphs for $R(3,8)$ was not yet known. When present research began, 430 215 $(3,8;27)$-graphs were known (most of these were generated by McKay).

We used *triangleramsey* to compute all mtf $(3,8;27)$-graphs. This yielded 21 798 graphs. The counts of all mtf $(3,8;\geq 15)$-graphs are given in Table 5.9. We also used the program *MTF* (see Section 5.2) to independently generate these mtf Ramsey graphs and obtained the same results. We then recursively removed edges in all possible ways from these mtf $(3,8;27)$-graphs to obtain the complete set $\mathcal{R}(3,8;27)$. This yielded 477 142 such graphs. As a test we verified that all of the 430 215 previously known $(3,8;27)$-graphs are indeed included in our list. The list can be downloaded from [87] and Table 5.10 contains the counts of these graphs according to their number of edges. So this proves that $e(3,8;27) = 85$ and hereby we have determined the last missing value of $e(3,8,n)$.

We also applied this method to generate the sets $\mathcal{R}(3,7;21), \mathcal{R}(3,7;22)$ and $\mathcal{R}(3,8;26,\leq 77)$ (see Appendix C for detailed results). The set of all $(3,7;22)$- and $(3,7;n,e(3,k,n))$-graphs were already determined by Radziszowski and Kreher [100] and our results are in complete agreement with theirs. Other enumerations are new.

This mtf method is infeasible for generating $(3,\geq 9;n)$-graphs for n which were needed in this work. Nevertheless, we used this method to verify the correctness

Number of vertices	Number of mtf Ramsey graphs
15	4 012
16	19 513
17	114 506
18	806 842
19	6 684 059
20	60 771 178
21	540 848 076
22	3 943 287 721
23	17 440 394 271
24	28 776 353 233
25	8 322 160 988
26	127 313 531
27	21 798

Table 5.9: Counts of all mtf $(3, 8; n)$-graphs for $n \geq 15$.

of our other programs for small numbers. The results agreed in all cases in which more than one method was used (see Section 5.3.10 for more details).

Minimum degree extension method

In their 1992 paper establishing $R(3, 8) = 28$, McKay and Zhang [94] proved that the set $\mathcal{R}(3, 8; 28)$ is empty by generating several sets $\mathcal{R}(3, k; n, e)$ with additional restrictions on the minimum degree $\delta(G)$. Suppose that one wants to generate all $(3, k; n, e)$-graphs. If G is such a graph and one considers its minimum degree vertex v, then we can reconstruct G given all possible graphs G_v. McKay and Zhang described such dependencies, designed an algorithm to reconstruct G, and completed the proof of $R(3, 8) = 28$ using this algorithm. We illustrate this method by an example. Note that if G is a $(3, k; n, e)$-graph and $v \in V(G)$ is a vertex of minimum degree, then G_v is a $(3, k-1; n-\delta(G)-1, \leq e-\delta(G)^2)$-graph.

Example: Suppose we want to construct all $(3, 8; 22, \leq 43)$-graphs. The average degree of a $(3, 8; 22, \leq 43)$-graph G is $\frac{2 \cdot 43}{22} < 4$, so $\delta(G) \leq 3$. Thus if we perform the following computations, we obtain all $(3, 8; 22, \leq 43)$-graphs:

Number of edges	Number of Ramsey graphs
85	4
86	92
87	1 374
88	11 915
89	52 807
90	122 419
91	151 308
92	99 332
93	33 145
94	4 746

Table 5.10: Counts of all $(3, 8; 27)$-graphs according to their number of edges.

- Construct all G's with $\delta(G) = 0$.
  - These G's can be obtained from $(3, 7; 21, \leq 43)$-graphs, however none exist since $e(3, 7; 21) = 51$.
- Construct all G's with $\delta(G) = 1$.
  - These G's can be obtained from $(3, 7; 20, \leq 42)$-graphs, however none exist since $e(3, 7; 20) = 44$.
- Construct all G's with $\delta(G) = 2$.
  - These G's can be obtained from $(3, 7; 19, \leq 39)$-graphs. (Remark: $e(3, 7; 19) = 37$).
- Construct all G's with $\delta(G) = 3$.
  - These G's can be obtained from $(3, 7; 18, \leq 34)$-graphs. (Remark: $e(3, 7; 18) = 30$).

The details of this algorithm are given in Section 5.3.4. The smaller Ramsey graphs on which the expansions are performed, can be obtained by recursively applying this minimum degree extension method until one reaches a size where all initial graphs were already known. If feasible, one can also generate the smaller Ramsey graphs by using the maximal triangle-free method. We implemented and

used this minimum degree extension method, and in all cases where more than one algorithm was used, it agreed with the results obtained by other algorithms (see Section 5.3.10 for details). However, using this method it was not feasible to generate most classes of graphs with higher parameters which were needed to improve upper bounds of $R(3,k)$. For example, we could not generate all $(3,9;28,\leq 69)$-graphs with this method, as the graphs with $\delta(G)=4$ are obtained from $(3,8;23,\leq 53)$-graphs, but there are already 10 691 100 $(3,8;23,\leq 52)$-graphs (see Table C.2 in Appendix C).

Neighbourhood gluing extension method

The most successful method which we used for our computations is the neighbourhood gluing extension method. Most of our computational results were obtained by this method.

Our general extension algorithm for an input $(3,k;m)$-graph H produces all $(3,k+1;n,e)$-graphs G, often with some specific restrictions on n and e, such that H is isomorphic to G_v for some vertex $v \in V(G)$. Thus the algorithm connects or *glues* the neighbours of a vertex v to H in all possible ways (see Figure 5.4 for an example). Note that each neighbour of v is glued to an independent set, otherwise the extended graph would contain triangles. This algorithm is also used in the minimum degree extension method and is described in more detail in Section 5.3.4.

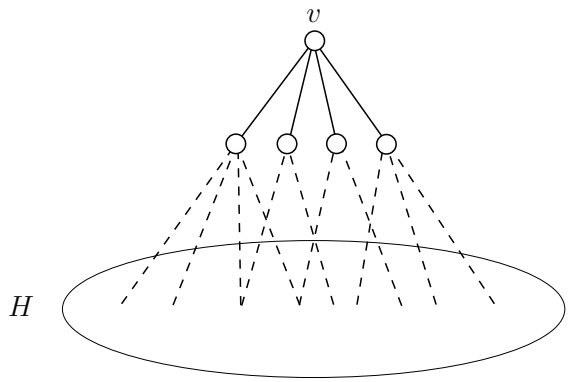


Figure 5.4: The operation of gluing the neighbours of v in all possible ways to H.

We used the following strategy to determine if the parameters of the input graphs to which our extender program was applied were such that the output

was guaranteed to contain all $(3, k+1; n, \leq e)$-graphs. Let $m_i = n - i - 1$, where i ranges over possible degrees in any graph G we look for, thus $\delta(G) \leq i \leq \Delta(G)$. In the broadest case we have $\delta(G) = \max\{n - R(3,k), 0\}$ and $\Delta(G) = k$. If G would contain a vertex v with degree less than $n - R(3,k)$, G_v would be a $(3,k)$-graph with at least $R(3,k)$ vertices, while no such graphs exist. If $\Delta(G)$ would be larger than k, G would contain an independent set of order $k+1$. We also identified a number of special cases where this range was more restricted.

Let t_i be an integer such that we have extended all $(3, k; m_i, < e(3,k,m_i)+t_i)$-graphs as potential G_v's of G. Now, if we use $e(3,k,m_i)+t_i$ instead of $e(3,k,m_i)$ in the formula for $\gamma(G)$ in inequality (4) from Section 5.3.1 for all possible values of i, we get:

$$\bar{\gamma}(G) = ne - \sum_{i=\delta(G)}^{\Delta(G)} n_i\big(i^2 + e(3,k,n-i-1) + t_i\big), \tag{5}$$

where n_i stands for the number of vertices of degree i in G and $n = \sum_{i=\delta(G)}^{\Delta(G)} n_i$ and $2e = \sum_{i=\delta(G)}^{\Delta(G)} i n_i$. If there are no degree sequences $n_{\delta(G)}, ..., n_{\Delta(G)}$ for which $\bar{\gamma}(G)$ is non-negative, we can conclude that all $(3, k+1; n, \leq e)$-graphs were already generated.

Example: Table 5.11 lists specific parameters of the general process used to obtain all $(3, 8; 25, \leq 65)$-graphs. Every vertex v in any $(3, 8; 25, \leq 65)$-graph has degree i, for some $2 \leq i \leq 7$. The corresponding graph G_v is of type $(3, 7; m_i, |E(G_v)|)$. The values of $e(3,7,m)$ are included in Table C.1 from Appendix C, and let the t_i's be as in Table 5.11. When we use these values of $e(3,7,m_i)$ and t_i in equality (5) for $\bar{\gamma}(G)$, there are no degree sequences of $(3, 8; 25, \leq 65)$-graphs for which $\bar{\gamma}(G)$ is non-negative. Thus, if we run the extender program for all possible graphs G_v with the number of edges listed in the last column of Table 5.11, we will obtain all $(3, 8; 25, e)$-graphs for $e \leq 65$.

The set of increments t_i accomplishing this goal is not unique, there are others which work. We just tried to minimise the amount of required computations in a greedy way (i.e. we make an initial guess for the values of t_i and then increase them one by one till there are no degree sequences for which $\bar{\gamma}(G)$ is non-negative). Note that the largest increments t_i in Table 5.11 occur for i's which are close to the average degree of G.

$i = deg_G(v)$	$m_i = \|V(G_v)\|$	$e(3,7,m_i)$	t_i	$\|E(G_v)\| = e - Z(v)$
2	22	60	1	60
3	21	51	1	51
4	20	44	2	44, 45
5	19	37	3	37, 38, 39
6	18	30	2	30, 31
7	17	25	1	25

Table 5.11: Obtaining all $(3, 8; 25, \leq 65)$-graphs.

5.3.4 Algorithmic details

In this section we present details about the algorithm which produces all $(3, k+1; n, e)$-graphs G for an input $(3, k; m)$-graph H such that for some vertex $v \in V(G)$, H is isomorphic to G_v. This algorithm is used in the minimum degree and neighbourhood gluing extension method. The implementation of the algorithm to generate maximal triangle-free Ramsey graphs is described in Section 5.2.4.

Given a $(3, k; n, f)$-graph G' as input and an expansion degree d, a desired maximum number of edges e and a minimum degree d_m as parameters. Our program constructs all $(3, k+1; n+d+1, \leq e)$-graphs G with $\delta(G) \geq d_m$ for which there is a vertex $v \in V(G)$ such that $deg(v) = d$ and $G_v \cong G'$. More specifically, the program adds a vertex v with neighbours $u_1, ..., u_d$ to G' and connects them to independent sets of G' in all possible ways, so that the resulting graph is a $(3, k+1; n+d+1, \leq e)$-graph with $\delta(G) \geq d_m$. We also use this notation in the remainder of this section, i.e. the extension procedure always adds a new vertex v with d neighbours $u_1, ..., u_d$ and connects these neighbours to independent sets $S_1, ..., S_d$ of G' (so u_i is connected to S_i, for $1 \leq i \leq d$) to construct a $(3, k+1; n+d+1, \leq e)$-graph G. (∗)

The extension program first determines all independent sets of G' of orders t that are possible, namely for which $d_m - 1 \leq t \leq k-1$. The program then recursively assigns the d neighbours of v to the eligible independent sets of G' and adds the edges joining the u_i's to their associated independent sets. It is then tested if the resulting graph G is a valid $(3, k+1; n+d+1, \leq e)$-graph. If it is, we output it. This general process is greatly accelerated by the techniques which are described in the remainder of this section.

We bound the recursion if a given partial assignment of neighbours of v to independent sets cannot lead to any $(3, k+1; n+d+1, \leq e)$-graphs. Suppose

that i neighbours $u_1, ..., u_i$ of v have already been assigned to independent sets $S_1, \ldots, S_i$. If $V(G') \setminus (S_1 \cup ... \cup S_i)$ induces an independent set I of order $k+1-i$, this assignment cannot lead to any output since $I \cup \{u_1, \ldots, u_i\}$ would form an independent set of order $k+1$ in G. We could test this property for all subsets of S_i's, but we found it to be most efficient to do it only for all pairs. Namely, if $u_1, ..., u_i$ have already been assigned to independent sets $S_1, \ldots, S_i$ and we consider to assign the next neighbour of v to an independent set S, we test if for all j $(1 \leq j \leq i)$, $V(G') \setminus (S_j \cup S)$ does not induce any independent set of order $k-1$. The list of independent sets which can still be assigned is dynamically updated.

For the efficiency of the algorithm it is vital that determining the indepence number of the graph induced by $V(G') \setminus (S_1 \cup ... \cup S_i)$ is fast. Hence we precompute the independence numbers of all induced subgraphs of G'. This precomputation also needs to be done very efficiently. We represent a set of vertices $S \subset V(G')$ by a bitvector. The array `indep_number[]` of 2^n `unsigned chars` stores the independence number of the graph induced by S in G'. It is very important that `indep_number[]` fits into the memory. On the computers on which we performed the expansions this was still feasible up to $n = 31$ (this requires approximately 2 GB of memory). We investigated various approaches to precompute `indep_number[]`. The algorithm which is given as pseudocode in Algorithm 5.4 was by far be the most efficient one. In the pseudocode `bitv(S')` represents the bitvector corresponding with set S'.

Algorithm 5.4 Precomputing the independence number

for $i = 0$ **upto** $2^n - 1$ **do**
  set `indep_number[i]` $= 0$
end for
for $j = k-1$ **downto** $k+1-d$ **do**
  for all independent sets S of order j in G' **do**
    Recursively make all supersets S' of S, and
    if `indep_number[bitv(S')]` $= 0$ **then** set `indep_number[bitv(S')]` $= j$
    else break making supersets of S
  end for
end for

The key idea is that if the superset S' of S already has `indep_number[bitv(S')]` $\geq j$, we can break the recursion of making the supersets. Usually one can break very quickly. For small extension degrees $d \leq 3$, it is more efficient not to precompute these independence numbers, but instead to compute them as needed.

If a neighbour u_i of v $(1 \leq i \leq d)$ has been assigned to an independent set S_i (still using the notation of $(*)$), we also update the degrees of the vertices in G'. If u_i is being connected to S_i, the degree of every vertex of S_i increases by one. If the degree of a vertex w of G' becomes k, other neighbours of v cannot be assigned to independent sets which contain w. We call such vertices which are no longer eligible *forbidden vertices*. We store all of them in a dynamically updated bitvector. We also dynamically update the list of independent sets to which u_i's can still be assigned. Independent sets which contain forbidden vertices are removed from the list of *eligible* independent sets (i.e. independent sets to which u_i's can still be assigned). We test this by performing bitvector operations. If no eligible independent sets are left, we can bound the recursion. Note that we cannot break the recursion when the number of eligible independent sets is smaller than the number of neighbours of v that still have to be assigned to independent sets, since they can be assigned to the same independent set. If i neighbours of v are already assigned and the forbidden vertices form an independent of set order $k+1-(d-i)$, the recursion can also be bounded, though in general this criterion is weak.

We assign the neighbours $u_1, ..., u_d$ of v to independent sets $S_1, ..., S_d$ in ascending order of the independent sets, i.e. $|S_j| \leq |S_{j+1}|$ for all $1 \leq j < d$. Doing this rather than in descending order allows us to eliminate many candidate independent sets early in the recursion. If an independent set S is small, it is very likely that $V(G') \setminus S$ induces a large independent set. So if a neighbour of v is assigned to S, it is also very likely that S cannot be assigned to another neighbour of v or that assigning S eliminates many eligible independent sets.

Assigning the neighbours of v to independent sets of ascending order also gives us an easy lower bound for the number of edges in any potential output graph which can be obtained from the current graph and assignment. If the the neighbours $u_1, ..., u_i$ of v $(1 \leq i \leq d)$ have already been assigned to independent sets $S_1, \ldots, S_i$ and the current minimal order of eligible independent sets is t, then any expanded graph will have at least $f = |E(G')| + d + |S_1| + ... + |S_i| + t(d-i)$ edges. If $f > e$, we can bound the recursion as well since we are constructing $(3, k+1; n+d+1, \leq e)$-graphs.

The pseudocode of the recursive extension procedure is listed in Algorithm 5.5. It is assumed that `indep_number[]` (see Algorithm 5.4) and the list of eligible independent sets are already computed. The procedure `Construct_Ramsey()` has two parameters. The first parameter is the number of neighbours l of v $(0 \leq l \leq d)$ which are already assigned to an independent set of G'. The second parameter

is the minimum order of the independent set to which the next neighbour u_{l+1} of v can be assigned. The recursion is bounded if any of the bounding criteria described above can be applied. When all neighbours $u_1, ..., u_d$ of v have been assigned to independent sets $S_1, ..., S_d$ of G', we add edges connecting u_i with every vertex of S_i for every u_i $(1 \leq i \leq d)$.

Algorithm 5.5 Construct_Ramsey(num_assigned, current_order)

```
if num_assigned == d then
  expand graph G' to G
  if G is a (3, k+1; n+d+1, ≤ e)-graph then
    output G
  end if
else
  for every eligible set S of order current_order do
    assign S to u_{num_assigned+1}
    update the set of eligible independent sets
    Construct_Ramsey(num_assigned + 1, current_order)
  end for
  if current_order < k − 1 then
    Construct_Ramsey(num_assigned, current_order + 1)
  end if
end if
```

Our extension program does not perform any isomorphism rejection. We canonically label the output graphs with *nauty* [88] and remove the isomorphic copies. This is not a bottleneck as there are usually only a few $(3, k+1; n+d+1, \leq e)$-graphs which are constructed by our program. The results obtained by our extension algorithms are described in Sections 5.3.5 and 5.3.7. In Section 5.3.10 we describe how we tested the correctness of our programs.

Degree Sequence Feasibility

Suppose we know the values or lower bounds for $e(3, k, m)$ for some fixed k and we wish to know all feasible degree sequences of $(3, k+1; n, e)$-graphs G. We construct the system of integer constraints consisting of $n = \sum_{i=0}^{k} n_i$, $2e = \sum_{i=0}^{k} i n_i$, and inequality (4) from Section 5.3.1 (where n_i stands for the number of vertices of degree i in G). If it has no solutions (i.e. there are no degree sequences $n_0, ..., n_k$ for which $\gamma(G)$ is non-negative), we conclude that $e(3, k+1, n) > e$. Otherwise, we obtain possible degree sequences for G. This algorithm is similar in functionality to the package FRANK which was developed by Lesser [81].

5.3.5 Progress on computing small $e(3, k, n)$

As already mentioned in Section 5.3.2, the complete enumeration of the sets $\mathcal{R}(3, k)$ with $k \leq 6$ was established in [100, 94]. Radziszowski and Kreher determined the exact values of $e(3, 7, n)$ in [100] and those of $e(3, 8, \leq 26)$ in [101].

By performing computations using the algorithms which are described in Section 5.3.3, we determined the last missing value for $\alpha(G) < 8$, namely $e(3, 8, 27) = 85$. We also complete the computation of exact values for $e(3, 9, n)$ for all n and for all $n \leq 33$ for $k = 10$. Previously only the values for $e(3, 9, \leq 26)$ and $e(3, 10, \leq 28)$ were known (these are given by Theorem 5.8). We also establish new lower bounds for $e(3, 10, \geq 34)$. All of these improvements are obtained by performing computations using the algorithms from Section 5.3.3.

The exact counts of $(3, k; n, e)$-graphs for $k = 7, 8, 9, 10$ are listed in Tables C.1,C.2, C.3 and C.4, respectively, in Appendix C. All $(3, \leq 9; n, e(3, k, n))$-graphs which were constructed by our programs can be obtained from *House of Graphs* [15] by searching for the keywords "minimal ramsey graph" or from [58].

Exact values of $e(3, 9, n)$

The values of $e(3, 9, \leq 26)$ are determined by Theorem 5.8. The values of $e(3, 9, n)$ for $27 \leq n \leq 34$ were obtained by computations, mostly by the neighbourhood gluing extender algorithm described in Section 5.3.3. The results are presented in Table 5.12. All of these values improve the previously best known lower bounds for $e(3, 9, n)$ reported by Radziszowski and Kreher [101] and Lesser [81]. The equality $e(3, 9, 35) = 140$ will be established by Theorem 5.9 in Section 5.3.6.

n	$e(3, 9, n)$	comments
27	61	
28	68	the same as in [3]
29	77	
30	86	
31	95	
32	104	not enough for $R(3, 10) \leq 42$
33	118	just enough for Theorem 5.10
34	129	122 required for $R(3, 10) \leq 43$
35	140	Theorem 5.9
36	∞	hence $R(3, 9) \leq 36$, old bound

Table 5.12: Exact values of $e(3, 9, n)$, for $n \geq 27$

Values and lower bounds for $e(3, 10, n)$

The values of $e(3, 10, \leq 28)$ are determined by Theorem 5.8. The values for $29 \leq n \leq 34$ were obtained by the neighbourhood gluing extender algorithm described in Section 5.3.3 and are listed in Table 5.13. The lower bounds for $e(3, 10, \geq 35)$ are included in the second column of Table 5.13. They are based on solving integer constraints of inequalities (3) and (4) from Section 5.3.1, using the exact values of $e(3, 9, n)$ listed in Table 5.12, and results from the neighbourhood gluing extender algorithm used similarly as in the example of Section 5.3.3. Our bounds for $e(3, 10, n)$ improve the previously best known bounds [101, 81] for all $n \geq 30$.

By Theorem 5.10 (see Section 5.3.6) we know that any $(3, 10; 42)$-graph must be 9-regular (and thus have 189 edges). Hence all its graphs G_v are necessarily of type $(3, 9; 32, 108)$. There exist a large number of these graphs. Their generation, extensions to possible $(3, 10; 42, 189)$-graphs and the implied nonexistence of any $(3, 10; 42)$-graphs is described in Section 5.3.7.

n	$e(3, 10, n) \geq$	comments
29	58	exact, the same as in [81]
30	66	exact
31	73	exact
32	81	exact
33	90	exact
34	99	exact, $(3, 10; 34, 99)$-graph constructed by Backelin [4]
35	107	
36	117	
37	128	
38	139	146 required for $R(3, 11) \leq 49$
39	151	as required for $R(3, 11) \leq 50$, Theorem 5.13
40	161	
41	172	184 maximum
42	∞	hence $R(3, 10) \leq 42$, new bound, Theorem 5.12
43	∞	hence $R(3, 10) \leq 43$, old bound

Table 5.13: Values and lower bounds on $e(3, 10, n)$, for $n \geq 29$.

All lower bounds in Tables 5.12 and 5.13 required computations of our neighbourhood gluing extender algorithm. We did not perform any such computations in an attempt to improve the lower bounds of $e(3, \geq 11, n)$ as this was computa-

tionally infeasible. All results presented in Section 5.3.8 for $k \geq 11$ depend only on the degree sequence analysis and the results for $k \leq 10$.

5.3.6 Improved lower bounds for $e(3, 9, 35)$ and $e(3, 10, 42)$

Sometimes we can improve the lower bounds for $e(3, k, n)$ implied by inequalities (3) and (4) from Section 5.3.1 by a more detailed analysis of feasible degree sequences. Such improvements can typically be made in cases for which inequality (4) gives a small number of possible degree sequences $n_0, ..., n_{k-1}$ with only one strongly dominating degree (i.e. a degree sequence with a large value of n_i ($0 \leq i \leq k-1$), while the other n_j's are small), but where the graphs represented by the degree sequences are not regular. We have such a situation in the proofs of the two following theorems. These theorems were proven by Stanisław P. Radziszowski in [59].

Theorem 5.9. *There exists a unique* $(3, 9; 35)$*-graph, and* $e(3, 9, 35) = 140$.

Proof. Any $(3, 9; 35)$-graph G has $\Delta(G) \leq 8$, hence we have $|E(G)| \leq 140$. Suppose $G \in \mathcal{R}(3, 9; 35, 140 - s)$ for some $s \geq 0$. Since $R(3, 8) = 28$, the degrees of vertices in G are 7 or 8, and let there be n_7 and n_8 of them, respectively. We have $n_7 + n_8 = 35$, $n_7 = 2s$. In this case there are five solutions to inequality (4) with $0 \leq s \leq 4$. In particular, this shows that $e(3, 9, 35) \geq 136$. If $n_7 > 0$ (equivalently $s > 0$), then consider graph H induced in G by n_7 vertices of degree 7. Observe that $\delta(H) \leq s$, since H is a triangle-free graph with $2s$ vertices. Let v be a vertex in $V(G)$ of degree 7 connected to at most s other vertices of degree 7. Thus we have $Z_G(v) \geq 7s + 8(7 - s) = 56 - s$, and $|E(G_v)| \leq (140 - s) - (56 - s) = 84$. However G_v is a $(3, 8; 27)$-graph which contradicts the fact that $e(3, 8, 27) = 85$.

The computations extending all $(3, 8; 26, 76)$-graphs, using the neighbourhood gluing extension method described in Section 5.3.3, established that there exists a unique (up to isomorphism) 8-regular $(3, 9; 35)$-graph. We note that it is a circulant graph with 35 vertices with circular distances {1,7,11,16}, which was already found by Kalbfleisch [75] in 1966. This graph is depicted in Figure 5.5. Clearly, any $(3, 9; 35, 140)$-graph must be 8-regular, and thus the theorem follows. □

Theorem 5.10. $R(3, 10) = 43$ *if and only if* $e(3, 10, 42) = 189$.

Proof. It is known that $R(3, 10) \leq 43$ [101], i.e. there are no $(3, 10; 43)$-graphs. We will prove the theorem by showing that any $(3, 10; 42)$-graph must be 9-regular.

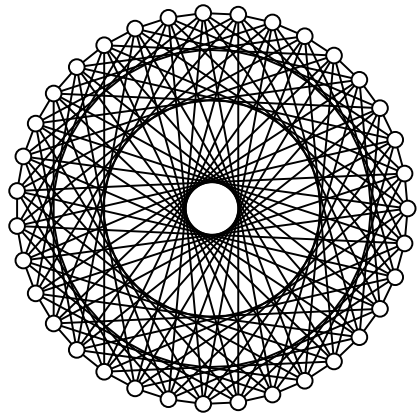


Figure 5.5: The unique $(3, 9; 35)$-graph.

Suppose $G \in \mathcal{R}(3, 10; 42, 189 - s)$ for some $s \geq 0$. Computations using the algorithm described in Section 5.3.4 established that G cannot have the unique $(3, 9; 35)$-graph as one of its G_v's. Hence, $7 \leq deg_G(v) \leq 9$ for all vertices $v \in V(G)$. The solutions to inequality (4) for which all degrees of the possible degree sequences are between 7 and 9 are presented in Table 5.14.

Note that for all $0 \leq s \leq 4$ we have $0 \leq n_7 \leq s$, $n_8 + 2n_7 = 2s$, $n_9 = 42 - n_8 - n_7$, and $|E(G)| = 189 - s$. Since $e(3, 9, 34) = 129$, using inequality (2) from Section 5.3.1 we see that $Z(v) \leq 60 - s$ for every vertex v of degree 7. Similarly, since $e(3, 9, 33) = 118$, $Z(v) \leq 71 - s$ for every vertex v of degree 8. If $s = 0$, then we are done. Otherwise consider the graph H induced in G by the $2s - n_7$ vertices of degree 7 or 8. Observe that $\delta(H) \leq s - n_7/2$, since H is triangle-free.

Case 1: $n_7 = 0$**.** Let v be a vertex in $V(G)$ of degree 8 connected to at most s other vertices of degree 8. This gives $Z_G(v) \geq 8s + 9(8 - s) = 72 - s$, which is a contradiction.

Case 2: $n_8 = 0$**.** Let v be a vertex in $V(G)$ of degree 7 connected to at most $s/2$ other vertices of degree 7 (in this case $|V(H)| = s$). This gives $Z_G(v) \geq 7s/2 + 9(7 - s/2) = 63 - s$, which is a contradiction.

Case 3: $n_7 = 1$**.** If v is the only vertex of degree 7, then $n_8 = 2s - 2$ and we easily have $Z_G(v) \geq 8n_8 + 9(7 - n_8) = 65 - 2s > 60 - s$, which again is a contradiction.

n_7	n_8	n_9	$\|E(G)\|$	$\gamma(G)$	s
0	8	34	185	24	4
1	6	35	185	25	4
2	4	36	185	26	4
3	2	37	185	27	4
4	0	38	185	28	4
0	6	36	186	60	3
1	4	37	186	61	3
2	2	38	186	62	3
3	0	39	186	63	3
0	4	38	187	96	2
1	2	39	187	97	2
2	0	40	187	98	2
0	2	40	188	132	1
1	0	41	188	133	1
0	0	42	189	168	0

Table 5.14: Solutions to inequality (4) for $(3, 10; 42, 189 - s)$-graphs.

Case 4: $n_7 = 2$**.** Both vertices of degree 7 must have $Z_G(v) \geq 7 + 8n_8 + 9(7 - n_8 - 1) = 61 - (2s - 2n_7) = 65 - 2s$, which is a contradiction.

Case 5: $n_7 > 2$**.** The only remaining degree sequence not covered by previous cases is $n_7 = 3$ and $n_8 = 2$, for $s = 4$ and $e = 185$. There is a vertex v of degree 7 connected to at most one other of degree 7, and thus $Z_G(v) \geq 7+2\cdot 8+4\cdot 9 > 60-s$, which is a contradiction. □

5.3.7 $R(3, 10) \leq 42$

Theorem 5.10 implies that any $(3, 10; 42)$-graph G must be 9-regular (and thus have 189 edges). Removing any vertex v with its neighbourhood from G yields a $(3, 9; 32, 108)$-graph G_v. Hence, our first task is to obtain all $(3, 9; 32, 108)$-graphs.

We used the neighbourhood gluing extension method to generate $(3, 9; 32, 108)$-graphs H which have a vertex v for which H_v is one of the following types: $(3, 8; 27)$, $(3, 8; 26, \leq 77)$, $(3, 8; 25, \leq 68)$, $(3, 8; 24, \leq 59)$ or $(3, 8; 23, 49)$. These extensions yielded a set $\mathcal{X}$ of 2 104 151 $(3, 9; 32, 108)$-graphs. When we use the notation of the example in Section 5.3.3, we have $4 \leq i \leq 8$, $m_i = 31 - i$ and

$t_i = 10, 5, 4, 4, 1$, respectively. When we use these values of $e(3, 8, m_i)$ and t_i in equality (5) for $\bar{\gamma}(G)$ for a $(3, 9; 32, 108)$-graph G, the only degree sequence for which $\bar{\gamma}(G)$ is non-negative is $n_6 = 8, n_7 = 24$.

Potentially, the complete set of $(3, 9; 32, 108)$-graphs could be obtained by performing additional extensions of degree 6 to $(3, 8; 25, 69)$-graphs or extensions of degree 7 to $(3, 8; 24, 60)$-graphs. However, there are already 12 581 543 $(3, 8; 25, \leq 68)$-graphs and 3 421 512 $(3, 8; 24, \leq 59)$-graphs (see Table C.2 in Appendix C), and there are many more with one additional edge. So extending all $(3, 8; 25, 69)$- or $(3, 8; 24, 60)$-graphs to $(3, 9; 32, 108)$-graphs using the algorithm from Section 5.3.4 is computationally infeasible. Hence, further refinement of the construction method to generate the $(3, 9; 32, 108)$-graphs which are not in $\mathcal{X}$ is needed.

Lemma 5.11 describes which $(3, 9; 32, 108)$-graphs are possibly not in $\mathcal{X}$.

Lemma 5.11. *Every* $(3, 9; 32, 108)$*-graph* $H \notin \mathcal{X}$ *has* $n_6 = 8, n_7 = 24$*, and furthermore in such* H *every vertex of degree* 6 *has exactly* 3 *neighbours of degree* 7 *and every vertex of degree* 7 *has exactly* 1 *neighbour of degree* 6*.*

Proof. As stated after the definition of $\mathcal{X}$ in the beginning of this section, the fact that $\bar{\gamma}(G)$ from equality (5) must be non-negative implies the specified degree sequence of $H \notin \mathcal{X}$. Suppose that H contains a vertex v of degree 6 with at least 4 neighbours of degree 7. As $Z_H(v) \geq 40$ and thus $|E(H_v)| \leq 68$. However, all such graphs were included in the set of inputs producing $\mathcal{X}$, so we have a contradiction. Similarly, suppose that H has a vertex v of degree 7 with no neighbours of degree 6. Then $Z_H(v) = 49$ and $|E(H_v)| = 59$, but all such graphs were used as inputs producing $\mathcal{X}$, hence again we have a contradiction. So for any $H \notin \mathcal{X}$, every vertex of degree 6 has at most 3 neighbours of degree 7 and every vertex of degree 7 has at least 1 vertex of degree 6. So there are exactly 24 edges connecting vertices of distinct degrees, and we can easily conclude that every vertex of degree 6 must have exactly 3 neighbours of degree 7 and every vertex of degree 7 exactly 1 neighbour of degree 6. □

We adapted the extension algorithm from Section 5.3.4 to generate this very restricted set of $(3, 9; 32, 108)$-graphs by performing extensions on all 64 233 886 $(3, 8; 24, 60)$-graphs (see Table C.2 in Appendix C). The result is that there are no $(3, 9; 32, 108)$-graphs which are not in $\mathcal{X}$. In Section 5.3.10 we describe how we tested the correctness of this adapted extension algorithm.

Theorem 5.12. $R(3,10) \leq 42$.

Proof. Suppose that G is a $(3,10;42)$-graph. By Theorem 5.10 it must be a 9-regular $(3,10;42,189)$-graph. Thus for any $v \in V(G)$, G_v is a $(3,9;32,108)$-graph. By Lemma 5.11 and the computations described above, we have proved there are exactly 2 104 151 such graphs. Since it is computationally not feasible to use the extension algorithm described in Section 5.3.4 to extend all $(3,9;32,108)$-graphs to $(3,10;42,189)$-graphs G, we implemented a modified version of this algorithm which was specialised for this task. The specialised algorithm goes as follows: the neighbours of v have to be connected to independent sets of order 8 in G_v. For every pair of (possibly equal) independent sets $\{S_i, S_j\}$ of order 8, we test if they can be assigned to two neighbours of v by checking if $V(G_v) \setminus (S_i \cup S_j)$ induces an independent set of order 8 in G_v. If so, we can bound the recursion. We cannot use the precomputed array `indep_number[]` of 2^n elements (see Section 5.3.4) for this task, since this array does not fit into the memory for $n \geq 32$. Instead we precompute a two-dimensional array which stores for every pair of independent sets $\{S_i, S_j\}$ of order 8 if $V(G_v) \setminus (S_i \cup S_j)$ induces an independent set of order 8 in G_v. The concept of eligible candidates (see Section 5.3.4) was also used, and the condition $\Delta(G) = 9$ turned out to be particularly strong in pruning the recursion. No 9-regular $(3,10;42,189)$-graphs were produced by this algorithm, and thus $R(3,10) \leq 42$. □

Theorem 5.12 improves the bound $R(3,10) \leq 43$ obtained in 1988 by Radziszowski and Kreher [101]. In Section 5.3.10 we describe how we tested the correctness of our implementations and the computational effort required for various parts of the computations.

5.3.8 New upper bounds on $R(3,k)$ for $k \geq 11$

We establish five further new upper bounds on the Ramsey numbers $R(3,k)$, for $k \geq 11$ as listed in Theorem 5.13. All of the new bounds improve the results listed in the 2011 revision of the Radziszowski's dynamic survey on small Ramsey numbers [99] by 1. The bound $R(3,16) \leq 98$ was also obtained by Backelin, though it was not published [3, 4]. Note that we don't improve the upper bound for $R(3,12)$.

Theorem 5.13. *The following upper bounds hold:* $R(3,11) \leq 50$, $R(3,13) \leq 68$, $R(3,14) \leq 77$, $R(3,15) \leq 87$, *and* $R(3,16) \leq 98$.

Proof. Each of the new upper bounds $R(3,k) \leq n$ can be obtained by showing that $e(3,k,n) = \infty$. The details of the intermediate stages of computations for all k are presented in the tables and comments of the remainder of this section. For $k = 16$ no data is shown except some comments in Table 5.19. In particular the data in this table implies $e(3,16,98) = \infty$ by inequality (4) from Section 5.3.1. □

Tables 5.15-5.19 list our lower bounds for $e(3,k,n)$, for $11 \leq k \leq 15$. These results were obtained by using our new bounds for $e(3,10,n)$ from Section 5.3.5 and degree sequence analysis of potential $(3,k;n,e)$-graphs which have to satisfy inequality (4). So here we did not perform any computations with the algorithms described in Section 5.3.3 as this is computationally infeasible.

All of our lower bounds for $e(3,k,n)$ are at least as good as those of Radziszowski and Kreher [101]. In Tables 5.15, 5.16 and 5.17, for $k = 11, 12$ and 13, respectively, we list some cases in the comments column, where the lower bounds for $e(3,k,n)$ listed in Lesser's thesis [81] (some of them credited to [3]) are better than our results. This is the case for n slightly larger than $13k/4 - 1$, mostly due to the theorems claimed in Backelin's unpublished manuscript [3, 4]. Our lower bounds for $e(3,k,n)$ and the implied upper bounds for $R(3,k)$, do not rely on these results. We have checked that assuming the results from [3, 4, 81] would not imply, using the methods of this paper, any further improvements on the upper bounds on $R(3,k)$ for $k \leq 16$, but they may for $k \geq 17$. Hence, if the results in [3, 81] are published, then using them jointly with our results may lead to better upper bounds on $R(3,k)$, at least for some $k \geq 17$.

Lower bounds for $e(3, 11, n)$

The exact values of $e(3, 11, \leq 31)$ are determined by Theorem 5.8. The bounds for $n = 32, 33$ which are marked with a 't' in Table 5.15 are implied by Theorem 5.8. The lower bounds for $e(3, 11, \geq 32)$ are included in the second column of Table 5.15. They are based on solving inequality (4) using known values and lower bounds for $e(3, 10, n)$ listed in Table 5.13 in Section 5.3.5. They are better than those in [81] for all $36 \leq n \leq 50$.

n	$e(3, 11, n) \geq$	comments
32	62t	63 in [81], 63 is exact [3, 4]
33	68t	69 in [81], 70 is exact [3, 4]
34	75	76 in [81], 77 is exact [3, 4]
35	83	84 in [81], credit to [3]
36	92	the same as in [3, 4]
37	100	
38	109	
39	117	unique degree sequence solution, 6-regular
40	128	
41	138	
42	149	
43	159	
44	170	
45	182	
46	195	199 required for $R(3, 12) \leq 58$
47	209	
48	222	unique solution: $n_9 = 36, n_{10} = 12$, 215 required for $R(3, 12) \leq 59$, old bound
49	237	245 maximum
50	∞	hence $R(3, 11) \leq 50$, new bound, Theorem 5.13
51	∞	hence $R(3, 11) \leq 51$, old bound

Table 5.15: Lower bounds on $e(3, 11, n)$, for $n \geq 32$.

The maximum number of edges in any $(3, 11; 49)$-graph is that of a 10-regular graph, so a proof of $e(3, 11, 49) > 245$ would imply $R(3, 11) \leq 49$. Observe that any graph G_v of any 10-regular $(3, 11; 50)$-graph must be a $(3, 10; 39, 150)$-graph. Thus, our improvement of the upper bound on $R(3, 11)$ from 51 to 50 is mainly due to the new lower bound $e(3, 10, 39) \geq 151$.

Lower bounds for $e(3, 12, n)$

The exact values of $e(3, 12, \leq 34)$ are determined by Theorem 5.8. The bounds for $35 \leq n \leq 37$ which are marked with a 't' in Table 5.16 are implied by Theorem 5.8. The lower bounds for $e(3, 12, \geq 35)$ are included in the second column of Table 5.16. They are based on solving inequality (4) using known values and lower bounds for $e(3, 11, n)$ given in Table 5.15. They are better than those in [81] for all $43 \leq n \leq 58$.

An improvement of the upper bound on $R(3, 12)$ obtained by Lesser [81] from 60 to 59 is now immediate (it formed a significant part of her thesis), but a further improvement from 59 to 58 would require an increase of the lower bound for $e(3, 12, 58)$ by 4.

n	$e(3, 12, n) \geq$	comments
35	67t	68 in [81], 68 is exact [3, 4]
36	73t	74 in [81], 75 is exact [3, 4]
37	79t	81 in [81], 82 is exact [3, 4]
38	86	88 in [81], 89 [3]
39	93	95 in [81], 96 [3]
40	100	102 in [81], 103 [3]
41	109	111 in [81]
42	119	119 in [81], 120 in [3]
43	128	the same as in [3]
44	138	
45	148	
46	158	
47	167	168, proof based on Table 7 [4]
48	179	180, proof based on Table 7 [4]
49	191	
50	203	
51	216	
52	229	
53	241	
54	255	259 required for $R(3, 13) \leq 67$
55	269	265 required for $R(3, 13) \leq 68$, Theorem 5.13
56	283	
57	299	
58	316	319 maximum
59	∞	hence $R(3, 12) \leq 59$, old bound

Table 5.16: Lower bounds on $e(3, 12, n)$, for $n \geq 35$.

Lower bounds for $e(3, 13, n)$

The exact values of $e(3, 13, \leq 39)$ are determined by Theorem 5.8. The bound for $n = 40$ is implied by Theorem 5.8. The lower bounds for $e(3, 13, \geq 40)$ are included in the second column of Table 5.17. They are based on solving inequality (4) using lower bounds for $e(3, 12, n)$ listed in Table 5.16. They are better than those in [81] for all $51 \leq n \leq 68$.

n	$e(3, 13, n) \geq$	comments
40	84t	86 in [81], 87 is exact [4]
41	91	93 in [81], 94 is exact [3]
42	97	100 in [81], 101 in [3]
43	104	107 in [81], 108 in [3]
44	112	114 in [81], 115 in [3]
45	120	122 in [81], 123 in [3]
46	128	130 in [81], 132 in [3]
47	136	139 in [81], 140 in [3]
48	146	148 in [81]
49	157	158 in [81]
50	167	167 in [81], 168 in [3]
51	177	178 in [3]
52	189	
53	200	
54	212	
55	223	
56	234	
57	247	
58	260	
59	275	
60	289	
61	303	
62	319	326 required for $R(3, 14) \leq 76$
63	334	
64	350	345 required for $R(3, 14) \leq 77$, Theorem 5.13
65	365	
66	381	
67	398	402 maximum
68	∞	hence $R(3, 13) \leq 68$, new bound
69	∞	hence $R(3, 13) \leq 69$, old bound

Table 5.17: Lower bounds on $e(3, 13, n)$, for $n \geq 40$.

Lower bounds for $e(3, 14, n)$

The exact values of $e(3, 14, \leq 41)$ are determined by Theorem 5.8. Only lower bounds for $e(3, 14, \geq 66)$ are included in the second column of Table 5.18, since these are relevant for our further analysis of $R(3, 15)$ and $R(3, 16)$. They are based on solving inequality (4) using lower bounds for $e(3, 13, n)$ listed in Table 5.17. They are better than those in [81] for all $66 \leq n \leq 77$.

n	$e(3, 14, n) \geq$	comments
66	321	
67	334	335, proof based on Table 9 [4]
68	350	
69	365	
70	381	
71	398	407 required for $R(3, 15) \leq 86$
72	415	414 required for $R(3, 15) \leq 87$, Theorem 5.13
73	432	
74	449	
75	468	
76	486	494 maximum
77	∞	hence $R(3, 14) \leq 77$, new bound
78	∞	hence $R(3, 14) \leq 78$, old bound

Table 5.18: Lower bounds on $e(3, 14, n)$, for $n \geq 66$.

Lower bounds for $e(3, 15, n)$

The exact values of $e(3, 15, \leq 44)$ are determined by Theorem 5.8. Only lower bounds for $e(3, 15, \geq 81)$ are included in the second column of Table 5.19, since these are relevant for further analysis of $R(3, 16)$. They are based on solving inequality (4) using lower bounds for $e(3, 14, n)$ listed in Table 5.18. They are better than those in [81] for all $81 \leq n \leq 87$.

n	$e(3, 15, n) \geq$	comments
81	497	498, proof based on Table 10 [4]
82	515	518 required for $R(3, 16) \leq 97$
		511 required for $R(3, 16) \leq 98$, Theorem 5.13
83	533	
84	552	
85	572	
86	592	602 maximum
87	∞	hence $R(3, 15) \leq 87$, new bound
88	∞	hence $R(3, 15) \leq 88$, old bound

Table 5.19: Lower bounds on $e(3, 15, n)$, for $n \geq 81$.

5.3.9 Improving lower bounds for $R(3, k)$

Next to improving the upper bounds for $R(3, k)$, a logical step to try to get closer to determining the actual value of $R(3, k)$ would be to improve the lower bounds for $R(3, k)$. Many of the known lower bounds for the classical Ramsey numbers which are listed in Radziszowski's dynamic survey [99] were obtained by performing highly tuned metaheuristics such as tabu search. We refer the reader to [47, 48, 98] for more details about these metaheuristics. In November 2012, Exoo [49] improved the lower bound of $R(3, 11)$ by one to 47 by using such a metaheuristic.

We tried to improve the lower bounds for $R(3, k)$ by using a different method. Let l be the currently known lower bound for $R(3, k)$. We start our algorithm from an initial set of $(3, k; l - 1)$-graphs which is as large as we could get. This set consists of the known $(3, k; l - 1)$-graphs. In case of $k = 10$, this set also contains graphs which were constructed by our gluing extender algorithm from Section 5.3.4. We then repeat Algorithm 5.6 until no new $(3, k; l - 1)$-graphs are found.

Algorithm 5.6 Procedure to construct more $(3, k; l-1)$-graphs.

for every input $(3, k; l-1)$-graph G **do**
  Recursively add edges in all possible ways to G such that no triangles are formed and store the mtf graphs in $\mathcal{M}$.
end for
for every mtf $(3, k; l-1)$-graph G in $\mathcal{M}$ **do**
  Recursively remove edges in all possible ways from G such that the obtained graphs are $(3, k; l-1)$-graphs and output these graphs.
end for

Afterwards we apply Algorithm 5.7 to test if one of the $(3, k; l-1)$-graphs G can be extended to a $(3, k; l)$-graph by adding a vertex and connecting it to an independent set in G.

Algorithm 5.7 Onevertex_extension($(3, k; l-1)$-graph G)

for every *maximal* independent set S where $deg(s) < k-1$ for all $s \in S$ **do**
  add vertex v to G and connect it to all $s \in S$
  if G is a $(3, k; l) - graph$ **then**
    output G
  end if
  remove v from G
end for

An independent set S of G is called *maximal* if no superset of S is an independent set. We only have to apply the expansion to maximal independent sets of G to test if a $(3, k; l-1)$-graph G can be extended to a $(3, k; l)$-graph by adding a vertex and connecting it to edges in G. Since if applying the expansion to an independent set I of G yields a $(3, k; l)$-graph, all graphs obtained by applying the expansion to independent sets which contain I are also $(3, k; l)$-graphs. We do not have to apply the expansion to independent sets which contain vertices of degree $k-1$ since then the expanded graph would contain a vertex of degree k and thus cannot be a $(3, k; l)$-graph. In principle the independent sets to which the expansion is applied must contain at least one vertex from every independent set of order $k-1$ from G, else the expanded graph would contain an independent set of order k. This is similar to the "approximating vertices" optimisation from Section 5.2.4. However we did not implement this optimisation here as Algorithm 5.7 is already more than fast enough for our purposes.

We also construct additional $(3, k; l-1)$-graphs by taking our current set of $(3, k; l-1)$-graphs, dropping every vertex (and its incident edges) once and applying Algorithm 5.7 to these $(3, k; l-2)$-graphs. Afterwards we again apply and repeat Algorithm 5.6 to this new set of $(3, k; l-1)$-graphs until no additional graphs are found.

It is known that $R(3, 10) \geq 40$ (see [47]). We applied this procedure to (amongst others) the set of almost 300 000 known $(3, 10; 39)$-graphs which were found by Exoo [49] and to the $(3, 10; 39)$-graphs which were constructed by our gluing extender algorithm. This allowed us to construct more than $42 \cdot 10^6$ $(3, 10; 39)$-graphs, and very likely there are more of them. Unfortunately none of these graphs is extensible to a $(3, 10; 40)$-graph. The $(3, 10; 39)$-graphs which we constructed have between 161 to 175 edges, hence we have $151 \leq e(3, 9, 39) \leq 161$. We expect that the actual value of $e(3, 9, 39)$ is much closer, if not equal, to 161. Despite many attempts by Exoo, us, and others, no $(3, 10; 40)$-graphs were constructed.

Since there are that many $(3, 10; 39)$-graphs while there is only one critical graph for $R(3, 9)$ (see Theorem 5.9 from Section 5.3.6), one might expect that $(3, 10; 40)$-graphs exist. On the other hand there are 477 142 critical graphs for $R(3, 8)$ (see Section 5.3.3). This again illustrates the irregular and unexpected behaviour of Ramsey numbers. Therefore we think that it is not possible to make a meaningful conjecture about the existence of $(3, 10; 40)$-graphs based on our current knowledge of $(3, 10; 39)$-graphs.

We also applied this procedure to try to improve the lower bounds for $R(3, k)$, for $11 \leq k \leq 16$, but unfortunately this did not lead to any new bounds.

5.3.10 Testing

Correctness

Since most results which are obtained in this chapter (and in this thesis in general) rely on computations, it is very important that the correctness of our programs has been thoroughly verified. In this section, we describe which correctness tests we performed.

- For every $(3, k)$-graph which was output by our programs, we verified that it does not contain an independent set of order k by using an independent program.
- For every $(3, k; n, e(3, k, n))$-graph which was generated by our programs,

we verified that dropping any edge creates an independent set of order k.

- For various $(3, k; n, \leq e)$-graphs we added up to f edges in all possible ways to obtain $(3, k; n, \leq e + f)$-graphs. For the cases where we already had the complete set of $(3, k; n, \leq e + f)$-graphs we verified that no new $(3, k; n, \leq e + f)$-graphs were obtained. We used this, amongst other cases, to verify that no new $(3, 9; 24, \leq 43)$, $(3, 9; 28, \leq 70)$, $(3, 9; 30, \leq 87)$ or $(3, 10; 30, \leq 67)$-graphs were obtained.

- For various $(3, k; n, \leq e+f)$-graphs we dropped one edge in all possible ways and verified that no new $(3, k; n, \leq e+f-1)$-graphs were obtained. We used this technique, amongst other cases, to verify that no new $(3, 9; 24, \leq 42)$, $(3, 9; 28, \leq 69)$, $(3, 9; 33, \leq 119)$, $(3, 9; 34, \leq 130)$, $(3, 10; 30, 66)$ or $(3, 10; 32, 81)$-graphs were obtained.

- For various sets of $(3, k + 1; n, \leq e)$-graphs we took each member G and constructed all G_v's from it. We then verified that this did not yield any new $(3, k; n - deg(v) - 1, \leq e - Z(v))$-graphs for the cases where we have all such graphs. We performed this test, amongst other cases, on the sets of $(3, 9; 28, \leq 70)$- and $(3, 10; 31, \leq 74)$-graphs.

Various sets of graphs can be obtained by more than one extension method from Section 5.3.3. As a test for the correctness, we applied multiple methods for the generation of several sets of graphs. We also compared our results with known results. In each case, the results are in complete agreement. More details are given below:

- The sets of $(3, 9; 24, \leq 43)$ and $(3, 9; 25, \leq 48)$-graphs (and several others) were obtained by both the minimum degree and neighbourhood glueing extension methods. The results were in complete agreement.

- The sets of $(3, 7; 21, \leq 55)$, $(3, 7; 22)$, $(3, 8; 26, \leq 76)$ and $(3, 8; 27, \leq 88)$-graphs were obtained by both the maximal triangle-free method and the neighbourhood gluing extension method. The results were in complete agreement. As these programs are entirely independent and the output sets are large, we think that this provides strong evidence of their correctness.

- The counts of $(3, 7; 16, 20)$, $(3, 7; 17, 25)$, $(3, 7; 18, 30)$, $(3, 7; 19, 37)$, $(3, 7; 20, 44)$, $(3, 7; 21, 51)$ and $(3, 7; 22, e)$-graphs for all $60 \leq e \leq 66$, are confirmed by [100].

- The counts of $(3,7;18,31)$, $(3,7;19,38)$, $(3,7;20,45)$ and $(3,7;21,\leq 53)$-graphs are confirmed by [101].
- The counts of $(3,8;19,25)$, $(3,8;20,30)$, $(3,8;21,35)$ and $(3,9;24,40)$-graphs are confirmed by [102].
- The counts of $(3,7;16,21)$, $(3,7;17,26)$, $(3,8;22,42)$ and $(3,9;25,47)$-graphs are confirmed by [4].

We also performed additional correctness tests for the specialised algorithms described in Section 5.3.7 (which were used to prove $R(3,10) \leq 42$):

- The specialised program of Section 5.3.7 was used to extend $(3,8;26,76)$- to $(3,9;35,140)$-graphs and it produced the unique $(3,9;35,140)$-graph.
- We relaxed the conditions to generate the $(3,9;32,108)$-graphs from Lemma 5.11 (see Section 5.3.7) by dropping the requirement that each vertex of degree 6 has 3 neighbours of degree 7, and enforcing just one vertex of degree 7 with exactly one neighbour of degree 6. This yielded 21 602 graphs. We verified that each of these graphs was indeed already included in the set $\mathcal{X}$ and that $\mathcal{X}$ does not contain any additional such graphs.

Since our results are in complete agreement with previous results and since all of our consistency tests passed, we believe that this is strong evidence for the correctness of our implementations and results.

Computation Time

We implemented the extension algorithms described in Sections 5.3.4 and 5.3.7 in C code. Most computations were performed on a cluster with Intel Xeon L5520 CPU's at 2.27 GHz, on which a computational effort of one CPU year can be usually completed in about 8 elapsed hours. The overall computational effort which was required to improve the upper bounds of $R(3,k)$ is estimated to be about 50 CPU years. This includes the time used by a variety of programs. The most CPU-intensive tasks are listed in the following.

For the proof of $R(3,10) \leq 42$, the first phase of obtaining $(3,9;32,108)$-graphs required about 5.5 CPU years. The bottlenecks of this phase were the computations required for extending all $(3,8;24,\leq 59)$-graphs (which required approximately 3.5 CPU years) and extending the $(3,8;25,\leq 68)$-graphs (which took more than 2 CPU years). The second phase of obtaining the special $(3,9;32,108)$-graphs with $n_6 = 8$, $n_7 = 24$ as in Lemma 5.11 took about 5.8 CPU years. The

specialised program from Section 5.3.7 extended all $(3, 9; 32, 108)$-graphs to 9-regular $(3, 10; 42, 189)$-graphs relatively fast, i.e. in about 0.25 CPU years. The computations to generate all $(3, 10; 39, \leq 150)$-graphs (there are none of these), which were needed for the bound $R(3, 11) \leq 50$, took about 4.8 CPU years.

The CPU time needed to complete the computations of Section 5.3.8 was negligible.

5.3.11 Closing remarks

We think that it is likely that the techniques from this section can also be used to obtain better bounds and possibly even determine the Ramsey number for some of the other graphs of order 10 whose Ramsey number could not be determined by the techniques from Section 5.2. In particular we think that these techniques could be applied to determine the Ramsey number of $K_{10} - e$ since it is known that $37 \leq R(K_3, K_{10} - e) \leq 38$.

However, the computations from Section 5.3.7 which were needed to improve the upper bound of $R(3, 10)$ were barely feasible. Despite many attempts by Exoo, us, and others, no $(3, 10; 40)$-graphs were constructed. Consequently, we anticipate that any further improvement to either of the bounds in $40 \leq R(3, 10) \leq 42$ will be very difficult.

Therefore we conclude that determining Ramsey numbers still remains an extremely difficult problem. Or to use a famous quote from Erdős [60]:

> *Suppose aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find the Ramsey number for red five and blue five. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the Ramsey number for red six and blue six, however, we would have no choice but to launch a preemptive attack.*

Appendix A

Notation

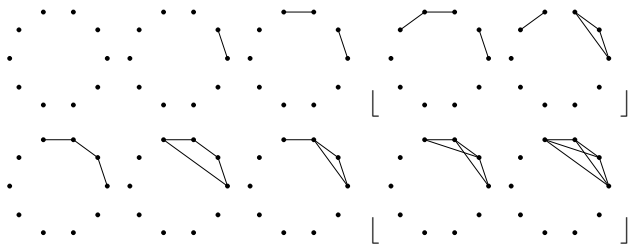
In each notation it is assumed that G and G' are graphs and that v and w are vertices.

$V(G)$	:	The set of vertices of G.
$E(G)$	:	The set of edges of G.
G^c	:	The complement of G.
$G[X]$	:	The subgraph of G induced by $X \subseteq V(G)$.
$N(v)$	:	The neighbourhood of v.
$deg(v)$	:	The degree of v.
$\delta(G)$	:	The minimal degree of G.
$\Delta(G)$	:	The maximal degree of G.
$d(v,w)$	:	The distance between v and w.
$g(G)$	:	The girth of G.
$\lambda_c(G)$	:	The cyclic edge-connectivity of G.
$\chi(G)$	:	The chromatic number of G.
$\chi'(G)$	:	The chromatic index of G.
$\alpha(G)$	:	The independence number of G.
$G \cong G'$	:	G and G' are isomorphic.
$\mathrm{Aut}(G)$	:	The automorphism group of G.
K_n	:	The complete graph with n vertices.
$K_{m,n}$	:	The complete bipartite graph with partitions of size m and n.
P_n	:	The path with n vertices.
$R(m,n)$	:	The Ramsey number $R(K_m, K_n)$.

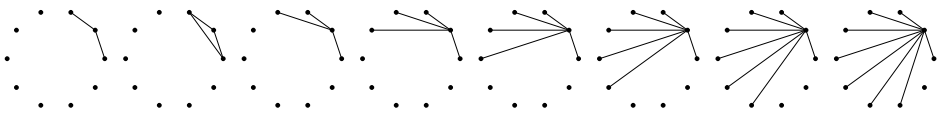
Appendix B

Ramsey numbers of connected graphs of order 10

The following 10 graphs G^c have $R(K_3, G) > 30$, but we were unable to determine their Ramsey number. Graphs which must have the same Ramsey number are grouped by $\lfloor$ and $\rfloor$.

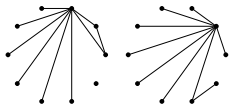


If G^c is one of the graphs:

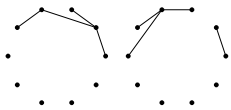


Then $R(K_3, G) = 36$. These are possibly not **all** graphs with $R(K_3, G) = 36$.

If G^c is contained in one of the graphs:

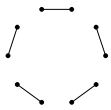


and contains one of the graphs:

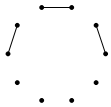


Then $R(K_3, G) = 31$. These are possibly not **all** graphs with $R(K_3, G) = 31$.

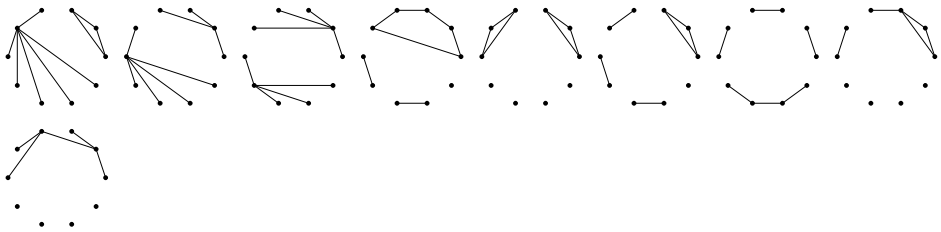
$R(K_3, G) = 30$ if and only if G^c is contained in:



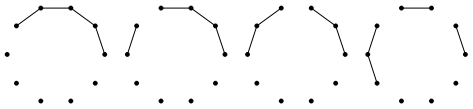
and contains:



$R(K_3, G) = 29$ if and only if G^c is contained in one of the graphs:



and contains one of the graphs:



The graphs with Ramsey number $R(K_3, G) < 29$ can be obtained from [22] or [31].

Appendix C

Number of Ramsey graphs for $R(3,k)$

Tables C.1,C.2, C.3 and C.4 contain all known exact counts of $(3,k;n,e)$-graphs for specified n, for $k = 7, 8, 9$ and 10, respectively. All graph counts were obtained by the algorithms described in Section 5.3.3. Empty entries indicate 0. In all cases, the maximum number of edges is bounded by $\Delta(G)n/2 \leq (k-1)n/2$. All $(3, \leq 9; n, e(3,k,n))$-graphs which were constructed by our programs can be obtained from the *House of Graphs* [15] by searching for the keywords "minimal ramsey graph" or from [58].

edges	number of vertices n						
e	16	17	18	19	20	21	22
20	2						
21	15						
22	201						
23	2965						
24	43331						
25	498927	2					
26	4054993	30					
27	?	642					
28	?	13334					
29	?	234279					
30	?	2883293	1				
31	?	?	15				
32	?	?	382				
33	?	?	8652				
34	?	?	160573				
35	?	?	2216896				
36	?	?	?				
37	?	?	?	11			
38	?	?	?	417			
39	?	?	?	10447			
40	?	?	?	172534			
41	?	?	?	1990118			
42-43	?	?	?	?			
44	?	?	?	?	15		
45	?	?	?	?	479		
46	?	?	?	?	10119		
47	?	?	?	?	132965		
48	?	?	?	?	1090842		
49-50		?	?	?	?		
51		?	?	?	?	4	
52			?	?	?	70	
53			?	?	?	717	
54			?	?	?	5167	
55				?	?	27289	
56				?	?	97249	
57				?	?	219623	
58					?	307464	
59					?	267374	
60					?	142741	1
61						43923	6
62						6484	30
63						331	60
64							59
65							25
66							10

Table C.1: Number of $(3,7;n,e)$-graphs, for $n \geq 16$.

edges	number of vertices n								
e	19	20	21	22	23	24	25	26	27
25	2								
26	37								
27	763								
28	16939								
29	?								
30	?	3							
31	?	60							
32	?	1980							
33	?	58649							
34	?	1594047							
35	?	?	1						
36	?	?	20						
37	?	?	950						
38	?	?	35797						
39	?	?	1079565						
40-41	?	?	?						
42	?	?	?	21					
43	?	?	?	1521					
44	?	?	?	72353					
45	?	?	?	2331462					
46-48	?	?	?	?					
49	?	?	?	?	102				
50	?	?	?	?	8241				
51	?	?	?	?	356041				
52	?	?	?	?	10326716				
53-55	?	?	?	?	?				
56	?	?	?	?	?	51			
57	?	?	?	?	?	3419			
58	?	?	?	?	?	129347			
59	?	?	?	?	?	3288695			
60	?	?	?	?	?	64233886			
61-64	?	?	?	?	?	?			
65	?	?	?	?	?	?	396		
66	?	?	?	?	?	?	21493		
67		?	?	?	?	?	613285		
68		?	?	?	?	?	11946369		
69-72		?	?	?	?	?	?		
73			?	?	?	?	?	62	
74				?	?	?	?	1625	
75				?	?	?	?	23409	
76				?	?	?	?	216151	
77				?	?	?	?	1526296	
78-84					?	?	?	?	
85							?	?	4
86							?	?	92
87							?	?	1374
88								?	11915
89								?	52807
90								?	122419
91								?	151308
92									99332
93									33145
94									4746

Table C.2: Number of $(3, 8; n, e)$-graphs, for $n \geq 19$.

edges	number of vertices n											
e	24	25	26	27	28	29	30	31	32	33	34	35
40	2											
41	32											
42	2089											
43	115588											
44-45	?											
46	?	1										
47	?	39										
48	?	4113										
49	?	306415										
50-51	?	?										
52	?	?	1									
53	?	?	1									
54	?	?	444									
55	?	?	58484									
56-60	?	?	?									
61	?	?	?	700								
62	?	?	?	95164								
63	?	?	?	6498191								
64-67	?	?	?	?								
68	?	?	?	?	126							
69	?	?	?	?	17223							
70	?	?	?	?	1202362							
71-76	?	?	?	?	?							
77	?	?	?	?	?	1342						
78	?	?	?	?	?	156686						
79-85	?	?	?	?	?	?						
86	?	?	?	?	?	?	1800					
87	?	?	?	?	?	?	147335					
88-94	?	?	?	?	?	?	?					
95	?	?	?	?	?	?	?	560				
96	?	?	?	?	?	?	?	35154				
97-103		?	?	?	?	?	?	?				
104			?	?	?	?	?	?	39			
105				?	?	?	?	?	952			
106				?	?	?	?	?	18598			
107				?	?	?	?	?	234681			
108				?	?	?	?	?	2104151			
109-117					?	?	?	?	?			
118							?	?	?	5		
119							?	?	?	69		
120-128							?	?	?	?		
129										?	1	
130										?	4	
131										?	$\geq$ 15	
132										?	$\geq$ 40	
133											$\geq$ 54	
134											$\geq$ 43	
135											$\geq$ 20	
136											$\geq$ 7	
137-139												
140												1

Table C.3: Number of $(3, 9; n, e)$-graphs, for $n \geq 24$.

edges	number of vertices n					
e	29	30	31	32	33	34
58	5					
59	1364					
60–65	?					
66	?	5084				
67	?	1048442				
68–72	?	?				
73	?	?	2657			
74	?	?	580667			
75–80	?	?	?			
81	?	?	?	6592		
82–89	?	?	?	?		
90	?	?	?	?	57099	
91–98	?	?	?	?	?	
99	?	?	?	?	?	≥ 1
≥ 100	?	?	?	?	?	?

Table C.4: Number of $(3, 10; n, e)$-graphs, for $29 \leq n \leq 34$.

Bibliography

[1] E. Albertazzi, C. Domene, P.W. Fowler, T. Heine, G. Seifert, C. Van Alsenoy, and F. Zerbetto. Pentagon adjacency as a determinant of fullerene stability. *Physical Chemistry Chemical Physics*, 1(12):2913–2918, 1999.

[2] R.E.L. Aldred, G Brinkmann, D. Van Dyck, V. Fack, and B.D. McKay. Graph structural properties of non-Yutsis graphs allowing fast recognition. *Discrete Applied Mathematics*, 157(2):377–386, 2009.

[3] J. Backelin. Contributions to a Ramsey Calculus. Unpublished, 2000-2012.

[4] J. Backelin. Personal communication, 2012.

[5] A.T. Balaban. Valence-isomerism of cyclopolyenes. *Revue Roumaine de chimie*, 11(9):1097–1116, 1966.

[6] R. Balakrishnan and K. Ranganathan. *A textbook of graph theory*. Universitext. Springer-Verlag, New York, 2000.

[7] J. Bornhöft, G. Brinkmann, and J. Greinus. Pentagon-hexagon-patches with short boundaries. *European Journal of Combinatorics*, 24(5):517–529, 2003.

[8] S. Brandt, G. Brinkmann, and T. Harmuth. All Ramsey numbers $r(K_3, G)$ for connected graphs of order 9. *Electronic Journal of Combinatorics*, 5, 1998.

[9] S. Brandt, G. Brinkmann, and T. Harmuth. The generation of maximal triangle-free graphs. *Graphs and Combinatorics*, 16(2):149–157, 2000.

[10] G. Brinkmann. *Zur mathematischen Behandlung gestörter periodischer Pflasterungen*. PhD thesis, Universität Bielefeld, 1990.

[11] G. Brinkmann. Fast generation of cubic graphs. *Journal of Graph Theory*, 23(2):139–149, 1996.

[12] G. Brinkmann. Problems and scope of spiral algorithms and spiral codes for polyhedral cages. *Chemical Physics Letters*, 272(3-4):193–198, 1997.

[13] G. Brinkmann. All Ramsey numbers $r(K_3, G)$ for connected graphs of order 7 and 8. *Combinatorics, Probablility and Computing*, 7(2):129–140, 1998.

[14] G. Brinkmann. Isomorphism rejection in structure generation programs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 51:25–38, 2000.

[15] G. Brinkmann, K. Coolsaet, J. Goedgebeur, and H. Mélot. House of Graphs: a database of interesting graphs. *Discrete Applied Mathematics*, 161(1-2):311–314, 2013. Available at `http://hog.grinvin.org/`.

[16] G. Brinkmann, S. Crevals, H. Mélot, Rylands L., and Steffen E. *alpha*-Labelings and the Structure of Trees with Nonzero α-Deficit. *Discrete Mathematics and Theoretical Computer Science*, 14(1):159–174, 2012.

[17] G. Brinkmann, O. Delgado Friedrichs, S. Lisken, A. Peeters, and N. Van Cleemput. CaGe - a Virtual Environment for Studying Some Special Classes of Plane Graphs - an Update. *MATCH Commun. Math. Comput. Chem.*, 63(3):533–552, 2010. Available at `http://caagt.ugent.be/CaGe`.

[18] G. Brinkmann, O. Delgado Friedrichs, and U. von Nathusius. Numbers of faces and boundary encodings of patches. In *Graphs and Discovery*, volume 69 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Sciences*, pages 27–38, 2005.

[19] G. Brinkmann and A.W.M. Dress. A constructive enumeration of fullerenes. *Journal of Algorithms*, 23:345–358, 1997.

[20] G. Brinkmann, D. Franceus, P.W. Fowler, and J.E. Graver. Growing fullerenes from seed: Growth transformations of fullerene polyhedra. *Chemical Physics Letters*, 428:386–393, 2006.

[21] G. Brinkmann and J. Goedgebeur. Homepage of triangleramsey: `http://caagt.ugent.be/triangleramsey/`.

[22] G. Brinkmann and J. Goedgebeur. House of Graphs: Ramsey numbers: `http://hog.grinvin.org/Ramsey`.

[23] G. Brinkmann and J. Goedgebeur. Downloadable lists of snarks: `http://hog.grinvin.org/Snarks`.

[24] G. Brinkmann, J. Goedgebeur, J. Hägglund, and K. Markström. Generation and properties of snarks. To appear in Journal of Combinatorial Theory, Series B.

[25] G. Brinkmann, J. Goedgebeur, and B.D. McKay. Homepage of buckygen: `http://caagt.ugent.be/buckygen/`.

[26] G. Brinkmann, J. Goedgebeur, and B.D. McKay. Downloadable lists of fullerenes: `http://hog.grinvin.org/Fullerenes`.

[27] G. Brinkmann, J. Goedgebeur, and B.D. McKay. Homepage of snarkhunter: `http://caagt.ugent.be/cubic/`.

[28] G. Brinkmann, J. Goedgebeur, and B.D. McKay. Generation of cubic graphs. *Discrete Mathematics and Theoretical Computer Science*, 13(2):69–80, 2011.

[29] G. Brinkmann, J. Goedgebeur, and B.D. McKay. The generation of fullerenes. *Journal of Chemical Information and Modeling*, 52(11):2910–2918, 2012.

[30] G. Brinkmann, J. Goedgebeur, and B.D. McKay. The smallest fullerene without a spiral. *Chemical Physics Letters*, 522(2):54–55, 2012.

[31] G. Brinkmann, J. Goedgebeur, and J.C. Schlage-Puchta. Ramsey Numbers $R(K_3, G)$ for Graphs of Order 10. *Electronic Journal of Combinatorics*, 19(4), 2012.

[32] G. Brinkmann, J.E. Graver, and C. Justus. Numbers of faces in disordered patches. *Journal of Mathematical Chemistry*, 45(2):263–278, 2009.

[33] G. Brinkmann and B.D. McKay. Fast generation of planar graphs. *MATCH Commun. Math. Comput. Chem.*, 58(2):323–357, 2007.

[34] G. Brinkmann, B.D. McKay, and U. von Nathusius. Backtrack search and look-ahead for the construction of planar cubic graphs with restricted face sizes. *MATCH Commun. Math. Comput. Chem.*, 48:163–177, 2003.

[35] G. Brinkmann and E. Steffen. Snarks and reducibility. *Ars Combinatoria*, 50:292–296, 1998.

[36] G. Brinkmann, U. von Nathusius, and A.H.R. Palser. A constructive enumeration of nanotube caps. *Discrete Applied Mathematics*, 116(1-2):55–71, 2002.

[37] F.C. Bussemaker, S. Čobeljić, D.M. Cvetković, and J.J. Seidel. Cubic graphs on $\leq$ 14 vertices. *Journal of Combinatorial Theory, Series B*, 23:234–235, 1977.

[38] F.C. Bussemaker and J.J. Seidel. Cubical graphs of order $2n \leq 10$. *T.H. Eindhoven*, 1968.

[39] A. Cavicchioli, M. Meschiari, B. Ruini, and F. Spaggiari. A survey on snarks and new results: products, reducibility and a computer search. *Journal of Graph Theory*, 28:57–86, 1998.

[40] U.A. Celmins. *On cubic graphs that do not have an edge 3-coloring.* PhD thesis, University of Waterloo, 1984.

[41] A.G. Chetwynd and R.J. Wilson. Snarks and supersnarks. In Y. Alavi et al., editor, *The Theory and Applications of Graphs*, pages 215–241. John Wiley & Sons, New York, 1981.

[42] J. de Vries. Over vlakke configuraties waarin elk punt met twee lijnen incident is. *Mededeelingen der Koninklijke Akademie voor Wetenschappen, Afdeeling Natuurkunde*, 3(6):382–407, 1889.

[43] J. de Vries. Sur les configurations planes dont chaque point supporte deux droites. *Rendiconti Circolo Matematico Palermo*, 5:221–226, 1891.

[44] R. Diestel. *Graph theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, fourth edition, 2010.

[45] M.S. Dresselhaus, G. Dresselhaus, and P. Avouris. *Carbon Nanotubes: Synthesis, Structure, Properties, and Applications.* Topics in applied physics. Springer-Verlag, 2001.

[46] P. Erdős. Some new problems and results in graph theory and other branches of combinatorial mathematics. In *Combinatorics and graph theory (Calcutta, 1980)*, volume 885 of *Lecture Notes in Mathematics*, pages 9–17. Springer, Berlin, 1981.

[47] G. Exoo. On two classical Ramsey numbers of the form $R(3, n)$. *SIAM Journal on Discrete Mathematics*, 2(4):488–490, 1989.

[48] G. Exoo. On the Ramsey number $R(4, 6)$. *Electronic Journal of Combinatorics*, 19, 2012.

[49] G. Exoo. Personal communication, 2012.

[50] I.A. Faradžev. Constructive enumeration of combinatorial objects. *Colloques internationaux C.N.R.S. No260 - Problèmes Combinatoires et Théorie des Graphes, Orsay*, pages 131–135, 1976.

[51] R.J. Faudree, C.C. Rousseau, and R.H. Schelp. All triangle-graph Ramsey numbers for connected graphs of order six. *Journal of Graph Theory*, 4(3):293–300, 1980.

[52] P.W. Fowler, M. Jooyandeh, and G. Brinkmann. Face-spiral codes in cubic polyhedral graphs with face sizes no larger than 6. *Journal of Mathematical Chemistry*, 50:2272–2280, 2012.

[53] P.W. Fowler and D.E. Manolopoulos. *An atlas of fullerenes*. International series of monographs on chemistry. Clarendon Press, 1995.

[54] M. Gardner. Mathematical games: Snarks, Boojums and other conjectures related to the four-color-map theorem. *Scientific American*, 234:126–130, 1976.

[55] W. Gasarch. Applications of Ramsey theory to computer science: `http://www.cs.umd.edu/~gasarch/ramsey/ramsey.html`.

[56] L.A. Goddyn. *Cycle covers of graphs*. PhD thesis, University of Waterloo, 1989.

[57] E.W. Godly and R. Taylor. Nomenclature and terminology of fullerenes: a preliminary survey. *Pure and applied chemistry*, 69(7):1411–1434, 1997.

[58] J. Goedgebeur. House of Graphs: Minimal Ramsey Graphs: `http://hog.grinvin.org/MinRamsey`.

[59] J. Goedgebeur and S.P. Radziszowski. New computational upper bounds for Ramsey numbers $R(3, k)$. *Electronic Journal of Combinatorics*, 20(1), 2013.

[60] R. Graham and J. Spencer. Ramsey Theory. *Scientific American*, 262(7):112–117, 1990.

[61] Jack E. Graver and James Yackel. Some graph theoretic results associated with Ramsey's theorem. *Journal of Combinatorial Theory*, 4:125–175, 1968.

[62] R. E. Greenwood and A. M. Gleason. Combinatorial relations and chromatic graphs. *Canadian Journal of Mathematics*, 7:1–7, 1955.

[63] C. Grinstead and S. Roberts. On the Ramsey numbers $R(3, 8)$ and $R(3, 9)$. *Journal of Combinatorial Theory, Series B*, 33(1):27–51, 1982.

[64] X. Guo, P. Hansen, and M. Zheng. Boundary uniqueness of fusenes. *Discrete Applied Mathematics*, 118(3):209–222, 2002.

[65] M. Hasheminezhad, H. Fleischner, and B.D. McKay. A universal set of growth operations for fullerenes. *Chemical Physics Letters*, 464:118–121, 2008.

[66] D.A. Holton and J. Sheehan. *The Petersen Graph*. Cambridge University Press, 1993. Australian Mathematical Society Lecture Series 7.

[67] Ian Holyer. The NP-completeness of edge-coloring. *SIAM J. Comput.*, 10(4):718–720, 1981.

[68] A. Huck. Reducible configurations for the cycle double cover conjecture. In *Proceedings of the 5th Twente Workshop on Graphs and Combinatorial Optimization (Enschede, 1997)*, volume 99, pages 71–90, 2000.

[69] S. Iijma. Helical microtubules of graphitic carbon. *Nature*, 354(6348):56–58, 1991.

[70] W. Imrich. Zehnpunktige kubische graphen. *Aequationes Mathematicae*, 6:6–10, 1971.

[71] R. Isaacs. Infinite families of nontrivial trivalent graphs which are not Tait colorable. *American Mathematical Monthly*, 82(3):221–239, 1975.

[72] B. Jackson. On circuit covers, circuit decompositions and Euler tours of graphs. In *Surveys in combinatorics, 1993 (Keele)*, volume 187 of *London Math. Soc. Lecture Note Ser.*, pages 191–210. Cambridge Univ. Press, Cambridge, 1993.

[73] F. Jaeger. A survey of the cycle double cover conjecture. In *Cycles in graphs (Burnaby, B.C., 1982)*, volume 115 of *North-Holland Math. Stud.*, pages 1–12. North-Holland, Amsterdam, 1985.

[74] C. Justus. Transformationen zwischen Fullerenen und die Flächenzahl von Patches mit gleichem Rand. Master's thesis, Universität Bielefeld, 2003. (Advisor: G. Brinkmann).

[75] J.G. Kalbfleisch. *Chromatic Graphs and Ramsey's Theorem.* PhD thesis, University of Waterloo, 1966.

[76] F. Kardoš and R. Škrekovski. Cyclic edge-cuts in fullerene graphs. *Journal of Mathematical Chemistry*, 44(1):121–132, 2008.

[77] G. Kéry. On a theorem of Ramsey. *Matematikai Lapok. Bolyai János Matematikai Társulat*, 15:204–224, 1964.

[78] J.H. Kim. The Ramsey number $R(3,t)$ has order of magnitude $t^2/\log t$. *Random Structures & Algorithms*, 7(3):173–207, 1995.

[79] M. Kochol. Snarks without small cycles. *Journal of Combinatorial Theory, Series B*, 67(1):34–47, 1996.

[80] H.W. Kroto, J.R. Heath, S.C. O'Brien, R.F. Curl, and R.E. Smalley. C_{60}: Buckminsterfullerene. *Nature*, 318(6042):162–163, 1985.

[81] A. Lesser. Theoretical and Computational Aspects of Ramsey Theory. Master's thesis, Matematiska Institutionen, Stockholms Universitet, 2001. (Advisor: J. Backelin).

[82] R.K. Lindsay, B.G. Buchanan, E.A. Feigenbaum, and J. Lederberg. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill, New York, 1980.

[83] X. Liu, D.J. Klein, T.G. Schmalz, and W.A. Seitz. Generation of carbon cage polyhedra. *Journal of Computational Chemistry*, 12(10):1252–1259, 1991.

[84] D.E. Manolopoulos and P.W. Fowler. Molecular graphs, point groups, and fullerenes. *Journal of Chemical Physics*, 96(10):7603–7614, 1992.

[85] D.E. Manolopoulos and P.W. Fowler. A fullerene without a spiral. *Chemical Physics Letters*, 204(1-2):1–7, 1993.

[86] D.E. Manolopoulos and J.C. May. Theoretical studies of the fullerenes: C_{34} to C_{70}. *Chemical Physics Letters*, 181:105–111, 1991.

[87] B.D. McKay. Combinatorial Data (Ramsey graphs): `http://cs.anu.edu.au/~bdm/data/ramsey.html`.

[88] B.D. McKay. Practical graph isomorphism. *10th. Manitoba Conference on Numerical Mathematics and Computing (Winnipeg, 1980)*, 30:45–87, 1981.

[89] B.D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998.

[90] B.D. McKay, W. Myrvold, and J. Nadon. Fast backtracking principles applied to find new cages. *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 188–191, 1998.

[91] B.D. McKay and S.P. Radziszowski. R(4,5)=25. *Journal of Graph Theory*, 19(3):309–322, 1995.

[92] B.D. McKay and G.F. Royle. Constructing the cubic graphs on up to 20 vertices. *Ars Combinatoria*, 21A:129–140, 1986.

[93] B.D. McKay and N.C. Wormald. Automorphisms of random graphs with specified degrees. *Combinatorica*, 4:325–338, 1984.

[94] B.D. McKay and K.M. Zhang. The value of the Ramsey number $R(3,8)$. *Journal of Graph Theory*, 16(1):99–105, 1992.

[95] M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30(2):137–146, 1999.

[96] R. Nedela and M. Skoviera. Decompositions and reductions of snarks. *Journal of Graph Theory*, 22:253–279, 1996.

[97] A.N. Petrenjuk and L.P. Petrenjuk. On constructive enumeration of 12 vertex cubic graphs (russian). *Combinatorial Analysis, Moscow*, 3, 1974.

[98] K. Piwakowski. Applying tabu search to determine new Ramsey graphs. *Electronic Journal of Combinatorics*, 3, 1996.

[99] S.P. Radziszowski. Small Ramsey Numbers. *Electronic Journal of Combinatorics*, 1994-2011. Dynamic Survey 1, revision 13, `http://www.combinatorics.org/`.

[100] S.P. Radziszowski and D.L. Kreher. On $(3,k)$ Ramsey graphs: theoretical and computational results. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 4:37–52, 1988.

[101] S.P. Radziszowski and D.L. Kreher. Upper bounds for some Ramsey numbers $R(3,k)$. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 4:207–212, 1988.

[102] S.P. Radziszowski and D.L. Kreher. Minimum triangle-free graphs. *Ars Combinatoria*, 31:65–92, 1991.

[103] F.P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

[104] R.C. Read. *Some enumeration problems in graph theory*. PhD thesis, London University, 1958.

[105] R.C. Read. Every one a winner. *Annals of Discrete Mathematics*, 2:107–120, 1978.

[106] R.W. Robinson and N.C. Wormald. Numbers of cubic graphs. *Journal of Graph Theory*, 7:463–467, 1983.

[107] R.W. Robinson and N.C. Wormald. Almost all cubic graphs are hamiltonian. *Random Structures & Algorithms*, 3(2):117–125, 1992.

[108] V. Rosta. Ramsey Theory Applications. *Electronic Journal of Combinatorics*, 2004. Dynamic Survey 13, `http://www.combinatorics.org/`.

[109] C.H. Sah. Combinatorial construction of fullerene structures. *Croatica Chemica Acta*, 66:1–12, 1993.

[110] S. Sanjmyatav. Algorithms for generation of cubic graphs. Master's thesis, Department of Computer Science, Australian National University, 2000. (Advisor: B. D. McKay).

[111] A. Schelten and I. Schiermeyer. Ramsey numbers $r(K_3, G)$ for connected graphs G of order seven. *Discrete Applied Mathematics*, 79(1-3):189–200, 1997.

[112] A. Schelten and I. Schiermeyer. Ramsey numbers $r(K_3, G)$ for $G \cong K_7 - 2P_2$ and $G \cong K_7 - 3P_2$. *Discrete Mathematics*, 191(1-3):191–196, 1998.

[113] T.G. Schmalz, W.A. Seitz, D.J. Klein, and G.E. Hite. Elemental carbon cages. *Journal of the American Chemical Society*, 110(4):1113–1127, 1988.

[114] P. D. Seymour. Sums of circuits. In *Graph theory and related topics (Proc. Conf., Univ. Waterloo, Waterloo, Ont., 1977)*, pages 341–355. Academic Press, New York, 1979.

[115] J. Spencer. Eighty years of Ramsey $R(3, k)\ldots$ and counting! In *Ramsey theory*, volume 285 of *Progress in Mathematics*, pages 27–39. Birkhäuser/Springer, New York, 2011.

[116] G. Szekeres. Polyhedral decompositions of cubic graphs. *Bulletin of the Australian Mathematical Society*, 8:367–387, 1973.

[117] S.J. Tans, M.H. Devoret, H.J. Dai, A. Thess, R.E. Smalley, L.J. Geerligs, and C. Dekker. Individual single-wall carbon nanotubes as quantum wires. *Nature*, 386(6624):474–477, APR 3 1997.

[118] W.P. Thurston. Shapes of polyhedra and triangulations of the sphere. In *The Epstein birthday schrift*, volume 1 of *Geometry & Topology Monographs*, pages 511–549. Geometry & Topology Publications, Coventry, 1998.

[119] W. T. Tutte. A contribution to the theory of chromatic polynomials. *Canadian Journal of Mathematics*, 6:80–91, 1954.

[120] D.B. West. *Introduction to graph theory.* Prentice Hall Inc., Upper Saddle River, NJ, 1996.

[121] H. Whitney. 2-Isomorphic Graphs. *American Journal of Mathematics*, 55(1-4):245–254, 1933.

[122] N.C. Wormald. Models of random regular graphs. *Surveys in Combinatorics*, pages 239–298, 1999. (LMS Lecture Note Series 267, Eds J.D. Lamb and D.A. Preece).

[123] J. Xia. Ramsey numbers involving a triangle: theory and algorithms. Master's thesis, Rochester Institue of Technology, 1993. (Advisor: S. P. Radziszowski).

[124] M. Yoshida and P. W. Fowler. Dihedral fullerenes of threefold symmetry with and without face spirals. *Journal of the Chemical Society, Faraday Transactions*, 93:3289–3294, 1997.

[125] M. Yoshida and E. Osawa. Formalized drawing of fullerene nets. 1. algorithm and exhaustive generation of isomeric structures. *Bulletin of the Chemical Society of Japan*, 68:2073–2081, 1995.

[126] C.Q. Zhang. *Integer flows and cycle covers of graphs*, volume 205 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 1997.

Nederlandstalige samenvatting

In deze thesis ontwikkelen we efficiënte algoritmes voor de generatie van structuren die gebruikt kunnen worden om chemische en wiskundige problemen op te lossen. Hiervoor ontwerpen en implementeren we algoritmes om wiskundige structuren (meer bepaald: *grafen*) te construeren.

In het **eerste hoofdstuk** geven we een inleiding over structuurgeneratie en introduceren we de nodige grafentheoretische begrippen die gebruikt worden in deze thesis. Een graaf is een structuur die bestaat uit een verzameling van toppen en een verzameling van bogen die verbindingen tussen deze toppen voorstellen. Een graaf kan onder andere dienen als model voor een wegennetwerk waar de toppen kruispunten voorstellen en de bogen straten. Dit model kan dan bijvoorbeeld gebruikt worden om de snelste route tussen twee kruispunten te bepalen. Een graaf kan ook een molecule modelleren. Hier stellen de toppen atomen voor en de bogen bindingen tussen de atomen. Een graaf kan ook gebruikt worden om een toekenning van lesgevers aan vakken te bepalen. In dit geval stellen de toppen lesgevers en vakken voor en de bogen geven aan welke vakken een bepaalde lesgever kan geven.

De algoritmes voor de generatie van structuren worden ontworpen om structuren in een specifieke klasse van grafen op een heel efficiënte manier te genereren, bijvoorbeeld de klasse van grafen waar alle toppen exact 3 buren hebben. In deze thesis onderzoeken en ontwerpen we *exhaustieve* generatiealgoritmes. Dit zijn algoritmes die ervoor zorgen dat *alle* grafen in de klasse van grafen die we wensen te genereren, opgesomd worden. Onze algoritmes zijn ook isomorfvrij, dit wil zeggen dat ze garanderen dat er geen *isomorfe* kopieën uitgevoerd worden (dat zijn grafen die dezelfde structuur hebben).

Structuurgeneratie heeft vooral toepassingen in de wiskunde en in de chemie.

In de wiskunde worden complete lijsten van structuren met specifieke eigenschappen gebruikt om wiskundige vermoedens te testen. In de theoretische chemie worden dergelijke lijsten gebruikt om de structuur van moleculen te bepalen of te voorspellen. In deze thesis ontwerpen we algoritmes voor de generatie van complete lijsten van grafen die belangrijke toepassingen hebben in de wiskunde alsook generatiealgoritmes die belangrijk zijn voor de chemie.

In het **tweede hoofdstuk** bespreken we de generatie van 3-reguliere grafen. Dit zijn grafen waar elke top exact 3 buren heeft. De eerste volledige lijst van 3-reguliere grafen werd reeds in 1889 opgesomd door de Nederlander J. de Vries: hij construeerde met de hand alle 3-reguliere grafen met 10 toppen [42, 43]. Sindsdien hebben nog veel andere wetenschappers onderzoek gedaan naar het opsommen van complete lijsten van 3-reguliere grafen. Vandaar dat dit als een benchmark probleem binnen de structuurgeneratie beschouwd kan worden.

Ons generatiealgoritme voor 3-reguliere grafen bestaat uit twee delen: eerst genereren we primitieve grafen (dit is een bijzondere deelklasse van de 3-reguliere grafen) en daarna vertrekken we van deze grafen om alle 3-reguliere grafen te construeren. Onze implementatie van dit algoritme is ruim 4 keer sneller dan de snelste bestaande generatieprogramma's voor 3-reguliere grafen. Tenslotte breiden we ons algoritme uit zodat ook 3-reguliere grafen zonder drie- of vierhoeken op een efficiënte manier opgesomd kunnen worden.

In het **derde hoofdstuk** passen we ons generatiealgoritme voor 3-reguliere grafen aan om *snarks* op een efficiënte manier te genereren. De snarks vormen een belangrijke deelklasse van 3-reguliere grafen. Ze bevatten geen driehoeken en hebben bijzondere kleurbaarheidseigenschappen. Het is gekend dat er voor 3-reguliere grafen altijd 3 of 4 kleuren nodig zijn om de bogen van zo'n graaf te kleuren zodanig dat de 3 bogen van elke top elk een verschillende kleur hebben. Bij snarks kunnen de bogen niet gekleurd worden met 3 kleuren. Deze klasse van grafen is bijzonder interessant omdat voor veel wiskundige vermoedens bewezen kan worden dat de vermoedens juist zijn als en slechts als ze voor snarks juist zijn. Vaak is het zelfs zo dat als er een tegenvoorbeeld zou zijn, het kleinste tegenvoorbeeld een snark zou zijn.

Onze implementatie van dit gespecialiseerde generatiealgoritme voor snarks is ruim 14 keer sneller dan het snelste bestaande programma om snarks te genereren. Dit liet ons toe om alle snarks tot en met 34 toppen op te sommen en alle snarks zonder vierhoeken tot en met 36 toppen. Voordien waren enkel de volledige lijsten van snarks met ten hoogste 32 toppen gekend. Gebruikmakend van deze nieuwe lijsten testen we 22 (open) wiskundige vermoedens. Onze lijsten leveren

verschillende tegenvoorbeelden op voor 8 van deze vermoedens. Dit toont dus aan dat (complete) lijsten van snarks niet louter theoretisch, maar ook in de praktijk een goede bron voor tegenvoorbeelden zijn.

In het **vierde hoofdstuk** bespreken we een belangrijke toepassing van structuurgeneratie in de chemie. Meer bepaald bespreken we de generatie van (modelleringen van) *fullerenen*. Dit zijn 3-reguliere grafen die op een bol getekend kunnen worden en waar elk vlak een vijf- of een zeshoek is. Hier stellen de toppen van de graaf koolstofatomen voor. Het eerste (chemische) fullereen werd in 1985 door H.W. Kroto en zijn collega's ontdekt [80]. Ze hebben een fullereen met de vorm van een voetbal ontdekt, namelijk de fameuze C_{60}-buckyball. Kroto en zijn collega's hebben later ook de Nobelprijs voor de chemie voor deze ontdekking gekregen. De koolstof nanotubes (dit zijn langwerpige fullerenen) vormen een belangrijke deelklasse van de fullerenen. Deze zijn veelbelovend voor toekomstige toepassingen [45]. Zo proberen onderzoekers nanotransistors te bouwen met behulp van nanotubes.

Er werd reeds veel onderzoek naar de generatie van fullerenen gedaan, maar tot nu toe was er nog maar één succesvol algoritme hiervoor gekend. Dit algoritme werd ontworpen door G. Brinkmann en A.W.M. Dress [19]. Zij hebben ook een efficiënt programma – genaamd *fullgen* – gebaseerd op hun algoritme geïmplementeerd. In dit hoofdstuk bespreken we een nieuw en volledig onafhankelijk generatiealgoritme voor fullerenen en tonen aan hoe dit op een heel efficiënte manier geïmplementeerd kan worden. Onze implementatie van dit algoritme is ruim 3.5 keer zo snel als *fullgen*. Onze generator leverde tegenstrijdige resultaten met *fullgen* op. Dit heeft geleid tot de detectie van een programmeerfout in *fullgen*. Door deze fout werden er startend vanaf 136 toppen te weinig fullerenen gegenereerd. Ondertussen is deze fout opgelost en komen de resultaten van *fullgen* volledig overeen met die van ons programma. Gebruikmakend van ons programma zijn we erin geslaagd om alle fullerenen tot en met 400 toppen te genereren. Dit liet ons toe om te bewijzen dat het kleinste tegenvoorbeeld voor het *spiraalvermoeden* 380 toppen heeft [30] (wat een open probleem was sinds 1991).

We ontwerpen ook een gespecialiseerd algoritme voor de generatie van IPR fullerenen. Dit zijn fullerenen die geen aanliggende vijfhoeken bevatten. Dergelijke fullerenen vormen een belangrijke deelklasse van de fullerenen omdat ze de neiging hebben om chemisch stabieler te zijn en het dus waarschijnlijker is dat deze fullerenen ook in de natuur voorkomen. Ook hier is onze implementatie van dit gespecialiseerde algoritme voor de generatie van IPR fullerenen duidelijk

sneller dan andere generatieprogramma's voor IPR fullerenen.

In het **laatste hoofdstuk** bespreken we nog een andere toepassing van structuurgeneratie in de wiskunde, namelijk de generatie van Ramsey grafen voor het bepalen van driehoek Ramsey getallen $R(K_3, G)$. Een driehoek Ramsey graaf is een graaf die geen driehoeken bevat en waarvan de complementaire graaf een gegeven graaf G niet als deelgraaf bevat. Het berekenen van Ramsey getallen is een moeilijk computationeel probleem waar reeds veel onderzoek naar gedaan is en kan als een benchmark probleem beschouwd worden.

In het eerste deel van dit hoofdstuk beschrijven we een geoptimaliseerde versie van een bestaand algoritme voor de generatie van *maximale* driehoeksvrije grafen [9]. Dit zijn grafen die geen driehoeken bevatten en waar het toevoegen van om het even welke boog een driehoek vormt. Daarna tonen we aan hoe dit algoritme uitgebreid kan worden om driehoek Ramsey grafen op een efficiënte manier op te sommen. Gebruikmakend van dit algoritme zijn we erin geslaagd om alle driehoek Ramsey getallen tot en met 30 te bepalen voor grafen met 10 toppen. Door onze computationele resultaten te combineren met nieuwe theoretische resultaten konden we het driehoek Ramsey getal van bijna alle 12 005 168 grafen van orde 10 bepalen, behalve voor 10 van de moeilijkste gevallen. Door de snelle groei van Ramsey getallen zal de lijst van driehoek Ramsey getallen voor grafen van orde 10 hoogstwaarschijnlijk voor een heel lange tijd de laatste lijst zijn die volledig bepaald zou kunnen worden.

In het tweede deel van dit hoofdstuk ontwikkelen we volledig andere gespecialiseerde algoritmes om de bovengrenzen voor *klassieke* driehoek Ramsey getallen te verbeteren. Dit zijn driehoek Ramsey getallen $R(K_3, G)$ waar de graaf G een complete graaf is. Gebruikmakend van deze algoritmes zijn we erin geslaagd om betere bovengrenzen te bepalen voor verschillende klassieke driehoek Ramsey getallen. Meer bepaald hebben we bewezen dat $R(K_3, K_{10}) \leq 42$, $R(K_3, K_{11}) \leq 50$, $R(K_3, K_{13}) \leq 68$, $R(K_3, K_{14}) \leq 77$, $R(K_3, K_{15}) \leq 87$ en $R(K_3, K_{16}) \leq 98$. Al deze nieuwe bovengrenzen verbeteren de oude grenzen met één. We bepalen ook alle *kritische* Ramsey grafen voor K_8 en bewijzen we dat de gekende kritische Ramsey graaf voor K_9 uniek is. Een Ramsey graaf voor G is *kritisch* als hij $R(K_3, G) - 1$ toppen heeft.

List of Figures

List of Tables

Index