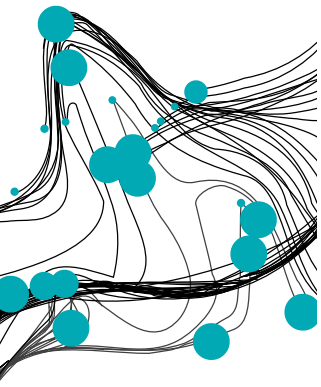
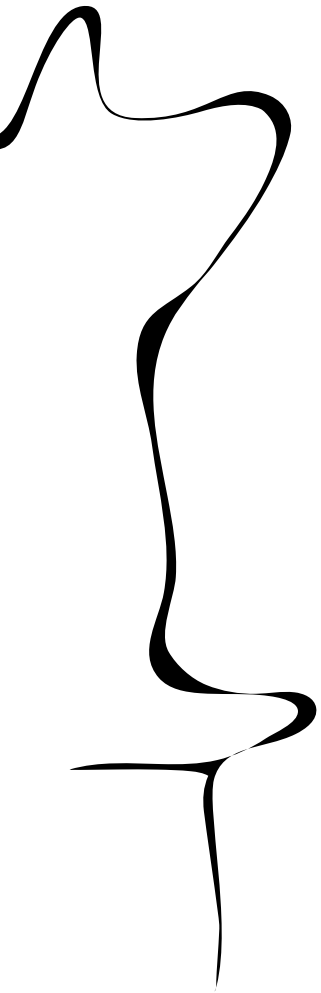


UNIVERSITY OF TWENTE.

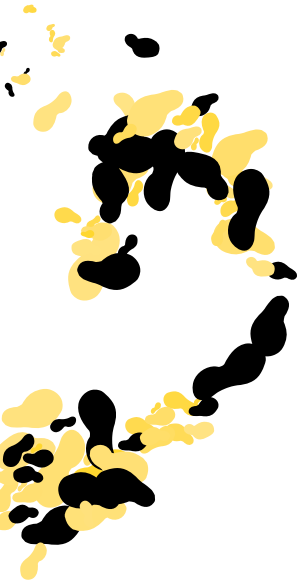
**Faculty of Electrical Engineering,
Mathematics & Computer Science**



An algorithmic approach to a conjecture of Chvátal on toughness and hamiltonicity of graphs

Tim Kemp

M.Sc. Thesis
August 2020



Supervisors:
prof.dr.ir. H.J. Broersma
prof.dr. M.J. Uetz
dr. T. van Dijk

Formal Methods and Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

The main motivation for this project is to find a nonhamiltonian graph with toughness at least $\frac{9}{4}$. This would improve the lower bound on t_0 in the following conjecture by Chvátal: there exists a real number t_0 such that every t_0-tough graph is hamiltonian [Chv73]. Secondly, we aim to research the open question whether the known nonhamiltonian 2-tough graph on 42 vertices is the smallest nonhamiltonian 2-tough graph [BBV00]. A third motivation is to analyse chordal graphs similarly. Every 10-tough chordal graph is hamiltonian [KK17], and there exist nonhamiltonian chordal graphs with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small real $\epsilon > 0$ [BBV00].

In contrast to the purely graph theoretical approach of the referenced papers, our approach is mainly algorithmic. However, it is based on the same construction that is used to construct nonhamiltonian graphs with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small real $\epsilon > 0$ [BBV00], which can also be applied to chordal graphs. We apply this known construction on all possible nonisomorphic graphs up to a certain number of vertices, by designing and implementing an algorithm that can determine the hamiltonicity and toughness of the constructed graph. We also design and implement an evolutionary algorithm, with the same aim to generate suitable graphs to construct nonhamiltonian graphs with a high toughness. Our research aims to optimise the performance of these two algorithms.

We conclude that there is no graph H of order $n \leq 11$ such that this construction results in a nonhamiltonian graph G with toughness $\tau \geq \frac{9}{4}$. Similarly, we conclude that there is no chordal graph H of order $n \leq 13$ such that this construction results in a nonhamiltonian chordal graph G with toughness $\tau \geq \frac{7}{4}$. This construction can neither be used to produce a smaller 2-tough nonhamiltonian graph than the known graph on 42 vertices, nor to produce a smaller nonhamiltonian chordal graph with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, by respectively applying the construction to graphs up to order 11 and 13. We have found that the field of evolutionary algorithms is well-suited to be applied to this problem, as the evolutionary algorithm enabled us to analyse larger graphs than an enumeration algorithm could ever handle on existing hardware. Based on the results of our evolutionary algorithm, we conjecture that there is no graph H of order $n \leq 16$ such that the aforementioned construction results in a nonhamiltonian graph G with toughness $\tau \geq \frac{9}{4}$.

Contents

Chapter 1

Introduction

Graph theory is a branch of mathematics that is focused on modelling and analysing problems that can be described by graphs. Graphs consist of two sets, namely a set of vertices representing certain objects, and a set of edges representing the relationships between pairs of objects. As an example, the graph of a world map could consist of vertices that represent the countries on the map and edges that indicate which of the pairs of countries share a border. But, a graph can also consist of vertices representing different types of objects, for instance jobs and machines, where the edges indicate which jobs can be processed on which machines. These two examples demonstrate the wide applicability of graphs. These applications range from mathematical scheduling problems to social networks to biological networks. Some examples where graphs are used in the field of computer science are search engines, network security and product recommendation.

The terminology in graph theory is inspired by the visual representation of graphs. A graph can easily be depicted by drawing the vertices as little circles and the edges as lines or curves between the circles that represent the corresponding vertices. This graphical representation is what makes graphs easy to understand, and this can make even very difficult problems conceptually simple. An example of a graph is shown in Figure 1.1. Note that the formal definition of a graph is presented in Chapter 2.

To variate on the opening application of a world map, it is also possible to analyse the infrastructure within a country. The vertices would be cities, where two cities are joined by an edge whenever there is a direct road between them. Graphs can be extended by putting weights on the edges of the graph. These would, in this case, denote the distance (i.e., the length of the direct road) between two cities. An important problem defined on such graphs is the travelling salesman problem. It states the following question: “What is the shortest route that visits each city exactly once, and returns to the initial city afterwards?”. If there are no weights on the edges, this optimisation problem reduces to the decision problem whether such a cycle exists. These

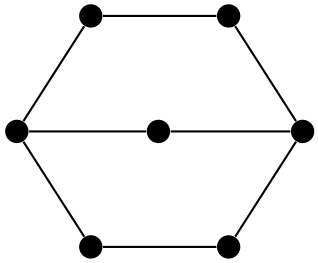


Figure 1.1: A nonhamiltonian graph with a toughness of $\frac{2}{3}$.

cycles are named Hamilton cycles, and hence this problem is known as the Hamilton cycle problem. The graph from Figure 1.1 is an example of a graph that does not contain a Hamilton cycle.

Many necessary and sufficient conditions regarding the existence of Hamilton cycles have been studied. In this research, we focus on those related to a graph's toughness. The relatively new concept of toughness was introduced in the 1970s by Chvátal as a measure to indicate how tightly various pieces of a graph hold together. Toughness is relatable to the connectivity of a graph. A graph G is t-tough if it cannot be split into k different connected components by the removal of fewer than $t \cdot k$ vertices. The toughness $\tau(G)$ of a graph G is the maximum value of t for which G is t-tough. The example graph of Figure 1.1 can be split into three components by the removal of two vertices; thus, the toughness is at most $\frac{2}{3}$. It is rather easy to check that there does not exist a cut which can lower this ratio, thus the toughness is exactly $\frac{2}{3}$.

In the paper where toughness has been introduced, Chvátal formulated Conjecture 1.1. A graph is said to be hamiltonian if it contains a Hamilton cycle. Chvátal proved that t_0 should be larger than $\frac{3}{2}$.

Conjecture 1.1 ([Chv73])**.** There exists a real number t_0 such that every t_0-tough graph is hamiltonian.

The conjecture that every t-tough graph with $t > \frac{3}{2}$ is hamiltonian has been disproven by Thomassen [Ber78]. More research on this conjecture is discussed in Chapter 2, where we also discuss different graph classes for which the conjecture holds. It has been proven that the conjecture can only be true if $t_0 \geq \frac{9}{4}$, by providing a nonhamiltonian graph with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small real $\epsilon > 0$ [BBV00].

In this research, we will try to disprove this conjecture for $t_0 \geq \frac{9}{4}$, using the same method to construct a tough nonhamiltonian graph as used in [BBV00]. Secondly, the similar conjecture on chordal graphs stated in Conjecture 1.2 will be analysed. A graph is chordal if every cycle of length greater than three has a chord. A chord is an edge which is not part of the cycle but joins two vertices of the cycle. Chordal graphs are generalisations

of trees, and many problems that are NP-hard to solve for general graphs are polynomially solvable for chordal graphs.

Conjecture 1.2. There exists a real number t_1 such that every t_1-tough chordal graph is hamiltonian.

It has been shown that there exist nonhamiltonian chordal graphs with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small real $\epsilon > 0$ [BBV00]. It has also been proven that Conjecture 1.2 is true by showing that every 18-tough chordal graph is hamiltonian [Che+98]. This has been improved by proving that every 10-tough chordal graph is hamiltonian [KK17]. There is still a large gap between $t_1 \geq \frac{7}{4}$ and $t_1 \leq 10$. We conjecture that t_1 in Conjecture 1.2 is closer to $\frac{7}{4}$ than to 10, but this research aims to increase the lower bound of $\frac{7}{4}$.

Both nonhamiltonian tough graphs found by Bauer, Broersma and Veldman, the general one and the chordal one, have been found by developing a construction that applies a graph operation on a small graph H in order to produce a new graph G. In order for G to be nonhamiltonian, the smaller graph H should satisfy the following property: "there exist two vertices u and v in H, such that there does not exist a Hamilton path from u to v".

A Hamilton path is a path that starts in u and terminates in v, and visits each vertex exactly once. If this property is satisfied, the toughness of G can be determined based on the choice of u and v. Both checking the existence of a Hamilton path and determining the toughness are NP-hard problems, with the former being NP-complete.

The resulting graph has been found by applying this construction to various small graphs by hand. Ever since, there has been the question whether applying this construction on a different graph could provide a better result, i.e. a nonhamiltonian graph with toughness at least $\frac{9}{4}$. This task to apply the construction on multiple graphs lends itself very well to be automated. In this research, we have designed and implemented algorithms to do so.

1.1 Problem statement and approach

The main motivation for this project is to find a nonhamiltonian graph with toughness at least $\frac{9}{4}$. This would improve the lower bound of t_0 in Conjecture 1.1. Alternatively, we may find different nonhamiltonian graphs with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$ than the one already known. This would perhaps also answer an open question whether the nonhamiltonian 2-tough graph on 42 vertices constructed by Bauer, Broersma and Veldman [BBV00] is the smallest nonhamiltonian 2-tough graph. A second motivation is to analyse chordal graphs in a similar manner. Recall that it is known that $\frac{7}{4} \leq t_1 \leq 10$ for Conjecture 1.2. Our method could increase this lower bound of $\frac{7}{4}$.

As it is not sure whether graphs that satisfy these criteria even exist, our first research objective is to apply the known construction on all possible nonisomorphic graphs up to a certain number of vertices. This will provide information whether more research on this construction would be useful or not. This objective will be approached by designing and implementing an algorithm which can efficiently enumerate over all nonisomorphic graphs of a specific order and determine which nonhamiltonian graph has the highest toughness, after applying the construction used in [BBV00]. The algorithm will be named the enumeration algorithm hereafter.

A second research objective is to design and implement an evolutionary algorithm, that can generate a tough nonhamiltonian graph using the aforementioned construction. An evolutionary algorithm is an optimisation technique that can alter a random initial graph in order to produce a desired graph after many iterations. This objective has been added during the research, in order to analyse larger graphs compared to those that the enumeration algorithm can handle.

The goal is to make both algorithms as fast as possible, in order to analyse more graphs. The enumeration algorithm provides the most reliable information, as it can give certainty whether applying the construction on any graph on n vertices could provide a nonhamiltonian graph with toughness above a fixed value t. By using an evolutionary algorithm, it is possible to test larger graphs for which the set of all nonisomorphic graphs could not even be generated. The set of nonisomorphic graphs of order n grows quickly. There are, for example, 11,117 nonisomorphic connected graphs of order 8, over a billion of order 11, and over fifty trillion of order 13. Generating all these nonisomorphic graphs of order thirteen would take months. Using our enumeration algorithm, enumerating over all nonisomorphic graphs is hardly feasible for graphs containing more than 12 vertices. Both the enumeration algorithm and the evolutionary algorithm can be verified by being able to return the known class of nonhamiltonian graphs with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$.

In Chapter 2, an introduction to graph theory is given, including relevant concepts and terminology. Afterwards, the literature that is relevant to our research is summarised. In Chapter 3, a similar introduction to evolutionary algorithms and its corresponding relevant literature is given. Chapter 4 contains the design and implementation of our algorithms. In Chapter 5, we present the results obtained by these algorithms. Finally, these results are discussed in Chapter 6, together with the possibilities for future research.

Chapter 2

Graph theory

In Chapter 1 a short explanation of graph theory has been given. In this section we shall explain it in a more formal way. This section starts with the definition of a graph, followed by common terminology and concepts. Whenever a new definition is given, it is highlighted in *italics*. In Section 2.1, Hamilton cycles are introduced in combination with some research on sufficient conditions for the existence of Hamilton cycles. In Section 2.2, the concept of toughness is introduced in combination with its relation to hamiltonicity. Finally, in Section 2.3 an important result on the sufficiency of toughness for hamiltonicity is explained. We refer to the textbook by Bondy and Murty [BM08] for an extensive introduction to graph theory.

A *graph* $G = (V, E)$ is an ordered pair of respectively a set of *vertices* and a set of *edges*, together with an *incidence function* that associates two vertices of G with each edge of G. These two vertices are called the *ends* of an edge. The *order* of a graph is the number of vertices in the graph. Example 2.1 shows a graph G corresponding to the graphical representation of Figure 1.1. Note that the vertices are labelled clockwise starting from the left, and v_7 corresponds to the central vertex in Figure 1.1.

Example 2.1. $G = (V, E)$ where

$$\begin{aligned} V =& \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \\ E =& \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\} \end{aligned}$$

and the incidence function ψ_G is defined by:

$$\begin{aligned} &\psi_G(e_1) = \{v_1, v_2\}\ \psi_G(e_2) = \{v_2, v_3\}\ \psi_G(e_3) = \{v_3, v_4\}\ \psi_G(e_4) = \{v_4, v_5\} \\ &\psi_G(e_5) = \{v_5, v_6\}\ \psi_G(e_6) = \{v_1, v_6\}\ \psi_G(e_7) = \{v_1, v_7\}\ \psi_G(e_8) = \{v_4, v_7\}. \end{aligned}$$

A *loop* is an edge with identical ends. A *link* is an edge with distinct ends. If two links have the same pair of ends, these links are *parallel edges*. A *simple* graph is a graph that does not contain loops or parallel edges. If this condition is not satisfied, the graph is a *multigraph*. There are other

variants of graphs, such as directed graphs. In a directed graph the incidence function associates each edge with two vertices, a tail vertex and a head vertex, such that each edge has a direction. Another variant is a weighted graph, where each edge is assigned a weight in the form of a number. This weight could for example represent a cost or a length.

A graph is *finite* if both the vertex set and the edge set are finite. In this thesis it is assumed that all graphs are simple and finite, thus in the remainder of this text, the term graph will refer to a simple finite graph. When restricted to simple graphs, it is possible to simplify the definition of a graph by omitting the incidence function. This is shown in Definition 2.2. The *complete* graph K_n refers to the graph of order n containing all $\frac{n\cdot(n-1)}{2}$ possible edges.

Definition 2.2. A simple graph G is an ordered pair (V, E), where V is a set of vertices and E is a set of two-element subsets of V.

Using this definition, the edge set of Example 2.1 can be represented as

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}, \{v_5, v_6\}, \{v_1, v_6\}, \{v_1, v_7\}, \{v_4, v_7\}\}.$$

Although this is more convenient than the definition used in Example 2.1, it is still not as clear as it could be. As suggested by the name, graphs can be represented graphically. Each vertex is depicted as a dot, and each edge is depicted as a line in between the two dots representing its ends. It is common to refer to a graphical representation of a graph as the graph itself, hence the dots and lines are also called the vertices and the edges. In the upcoming paragraphs, we present common terminology and concepts of graph theory that are used in this thesis.

Degree The ends of an edge are *incident* with the edge itself. Two vertices are *adjacent* if there exists an edge that is incident to both vertices. Distinct vertices are *neighbours* when these are adjacent. The *degree* of a vertex v in G refers to the number of edges of G incident with v. The degree of a vertex v is denoted by $d(v)$, and $\delta(G)$ is the minimum degree taken over all vertices in a graph G. The maximum degree is denoted by $\Delta(G)$.

Isomorphism A graph can have multiple graphical representations, but it is also possible for different graphs to have the same graphical representation. In this case the only difference between these graphs are the labels of the vertices. Such graphs are said to be isomorphic. Two graphs G and H are *isomorphic* if there exists a bijection $f : V(G) \to V(H)$ such that any $u, v \in V(G)$ are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H. Note that a bijection is a mapping that maps each element from one set to exactly one element of another set, and each element of the second set is also paired with exactly one element of the first set. An *automorphism* is an isomorphism of a graph to itself.

Subgraphs Let F and G be two graphs. Then F is a *subgraph* of G if $V(F) \subseteq V(G)$ and $E(F) \subseteq E(G)$. These subgraphs can be obtained by edge deletion and vertex deletion. *Edge deletion* is the removal of a specific edge e from $E(G)$, and the obtained graph is denoted by $G \setminus e$. *Vertex deletion* is the removal of a vertex v and all edges incident with v, and the obtained graph is denoted by $G - v$. F is a *spanning subgraph* of G if it can be obtained from G by edge deletions (possibly none). F is an *induced subgraph* of G if it can be obtained from G by vertex deletions (possibly none). An induced subgraph obtained by removing a set of vertices $X \subseteq V(G)$ is denoted as $G - X$. An induced subgraph obtained by keeping a set of vertices $Y \subseteq V(G)$ and removing all vertices of $V(G) \setminus Y$ from G is denoted by $G[Y]$.

Walks, trails, paths, and cycles A walk W is a sequence $v_0 e_1 v_1 \dots e_k v_k$ whose terms are alternately vertices and edges, such that v_{i-1} and v_i are the ends of e_i for $1 \leq i \leq k$. This walk W is referred to as a (v_0, v_k)-walk. v_0 is the *initial vertex*, v_k is the *terminal vertex*, and $v_1, \dots, v_k$ are the *internal vertices*. The *length* of the walk is the number of edges in the walk (k). Since we only consider simple graphs, a walk can be specified by only the vertices: $v_0 v_1 \dots v_k$. A *trail* is a walk whose edges are distinct. A *path* is a trail in which all the vertices are distinct. A walk is *closed* if the initial vertex and the terminal vertex are the same. A *cycle* is a closed trail that has a positive length and whose initial and internal vertices are all distinct.

Two vertices u and v in G are *connected* if there exists a $(u, v)-$path in G. Note that the existence of an (u, v)-walk guarantees the existence of an (u, v)-path. Connection is an equivalence relation on V, which means that it is reflexive, symmetric, and transitive. A consequence of being an equivalence relation is that there exists a partition of V into nonempty subsets $V_1, V_2, \dots, V_\omega$, such that u and v are connected if and only if both u and v belong to the same subset V_i. The induced subgraphs $G[V_1], G[V_2], \dots, G[V_\omega]$ are the *components* of G. The number of components of a graph G is denoted by $\omega(G)$. If $\omega(G) = 1$, G is *connected*. Otherwise, G is *disconnected*.

Chordal graphs A *chord* of a cycle C in a graph G is an edge in $E(G) \setminus E(C)$ both of whose ends lie on C. It is thus an edge that is not part of the cycle, but joins two vertices of the cycle. A graph is *chordal* if every cycle of length greater than three has a chord. An equivalent definition is that the graph contains no induced cycle of length four or more. An *induced cycle* is an induced subgraph consisting of exactly one cycle. An example of a chordal graph is shown in Figure 2.1.

Chordal graphs are generalisations of trees, and many problems that are NP-hard to solve for general graphs are polynomially solvable for chordal graphs, like the vertex colouring problem. Chordal graphs are also generalisations of interval graphs that appear naturally in scheduling applications: each

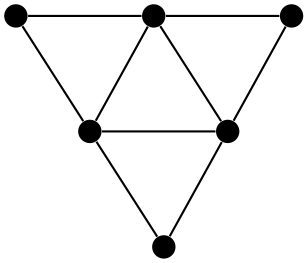


Figure 2.1: An example of a chordal graph.

vertex represents an interval on the real line, and two vertices are adjacent whenever the corresponding intervals intersect.

Like trees and interval graphs, a chordal graph G admits a so-called elimination scheme: if G has at least two vertices, then G contain a vertex v such that

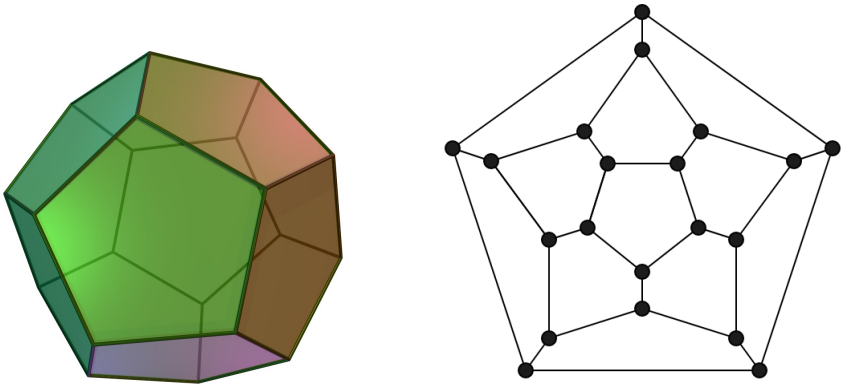
(i) $N(v)$ induces a complete graph in G (i.e. $N(v)$ is a clique in G), and

(ii) $G - v$ is again a chordal graph.

In a tree, v can be chosen as a vertex with degree 1; in an interval graph, v can be chosen as the interval with the leftmost starting point on the real line. The nice thing is that applying this recursively, one obtains a sequence of vertices $v_1, v_2, \ldots, v_n$ such that for every v_i $(i < n)$, the subgraph G_i of G induced by $\{v_{i+1}, \ldots, v_n\}$ is a chordal graph, and the neighbours of v_i in G_i form a clique. As an example, using the reverse ordering, one can obtain an optimal vertex colouring in polynomial time, by choosing the smallest eligible colour for the newly added vertex from the set of colours $\{1, 2, \ldots, k\}$ in each step.

2.1 Hamilton cycles

Hamilton cycles are named after William Hamilton, who first described them in his Icosian game [BM08]. The game is played along the vertices and edges of a dodecahedron graph, i.e. a graph on 20 vertices in which each vertex has degree 3. A dodecahedron is a three-dimensional shape with twelve flat polygonal faces and straight edges. An illustration of a (regular) dodecahedron is shown in Figure 2.2a. Each edge along this shape corresponds to an edge in the dodecahedron graph, which is illustrated in Figure 2.2b. The first player constructs a path along 5 vertices and the other has to extend this path into a Hamilton cycle. A *Hamilton cycle* is a cycle containing every vertex of a graph. A graph G containing a Hamilton cycle is called *hamiltonian*. Similarly, a *Hamilton path* is a path that contains every vertex of a graph, and a graph containing such a path is called *traceable*.



(a) A regular dodecahedron [Wik20]. (b) The dodecahedron graph [Com20].

Figure 2.2: Illustration of a dodecahedron and its corresponding graph.

The decision problem whether a graph contains a Hamilton cycle is NP-complete. Decision problems that are in the set NP have a solution that can be verified in polynomial time. A problem is NP-hard if all decision problems in the set NP can be transformed into it in polynomial time. Finally, a problem is NP-complete if it is both in NP and NP-hard. Hamiltonicity was one of the original 21 NP-complete problems proven to be reducible from the Boolean satisfiability problem, which was the first NP-complete problem [Kar72]. The set of problems that can be solved in polynomial time is called P. It is still an important open problem in computer science whether P and NP are distinct sets. As of now, no one has thus found a polynomial-time algorithm for any of the NP-complete problems and it is reasonable to believe that these do not exist.

Note that the problems whether a graph contains a Hamilton cycle and whether a graph contains a Hamilton path have the same complexity, as there are reductions from one problem to the other and vice versa. Given the decision problem whether there exists a Hamilton path in a graph G, one can construct a graph H by adding a single vertex v to G and by adding edges from v to all vertices in G. The existence of a Hamilton path in G now follows from the existence of a Hamilton cycle in H. Now G is traceable if and only if H is hamiltonian. So if the Hamilton problem is in P, then the traceability problem is also in P. If the second problem is in NP-complete, the first is also. For the other direction, the given problem is whether G is hamiltonian. The graph H is defined by adding a vertex v and making it a copy of a vertex $u \in G$ by adding the edges $\{v, w\}|\{u, w\} \in E(G)$, and by adding two more vertices s, t and joining them to respectively u and v. If G has a Hamilton cycle, then there clearly exists a Hamilton path from s to t in H. G is hamiltonian if and only if H is traceable, and it follows that the two decision problems are both NP-complete.

Hamiltonicity has been the subject of a lot of research, both in developing faster algorithms to check hamiltonicity by a computer and by devising necessary conditions and sufficient conditions. Our research is focused on a sufficient condition for hamiltonicity based on the graph's toughness. Below we will discuss some relevant necessary conditions and sufficient conditions for hamiltonicity. The motivation for doing this is two-fold. Firstly, it helps in placing this research in perspective by showing related work on hamiltonicity. Secondly, some of these theorems could be used in our own research. For this purpose we are particularly interested in conditions that are easy to check by a computer (in terms of algorithmic complexity), such as degree-based conditions.

Two well known classic results are the degree-based sufficient conditions by Dirac (Theorem 2.3) and Ore (Theorem 2.4). The latter is a generalisation of the former and therefore more often applicable, but it is harder to check. All the following results can be found in the dissertation 'On Hamiltonian Connected Graphs' [Wil73]. Both Theorem 2.3 and Theorem 2.4 can be derived from Theorem 2.5, which is an even more general condition based on the degrees of all vertices.

Theorem 2.3 (Dirac)**.** Let G be a graph of order n ($n \geq 3$). If $\delta(G) \geq \frac{n}{2}$, then G is hamiltonian.

Theorem 2.4 (Ore)**.** Let G be a graph of order n ($n \geq 3$). If $d(u)+d(v) \geq n$ for every pair u, v of distinct nonadjacent vertices of G, then G is hamiltonian.

Theorem 2.5 (Posa)**.** Let G be a graph of order n ($n \geq 3$). If for all $1 \leq j < \frac{n}{2}$, the number of vertices of degree not exceeding j is less than j, then G is hamiltonian.

The latter theorem can also be defined on the degree sequence of a graph. A *degree sequence* $d = (d_1, d_2, \ldots, d_n)$ of a graph G is a nondecreasing sequence of the vertex degrees of all vertices in G. Note that other texts may define it as a nonincreasing sequence, which is not applicable on the theorems below. Theorem 2.6 gives an even weaker sufficient condition. The final degree-based sufficient condition presented here is Theorem 2.7, which is the best possible generalisation of the theorems of Dirac, Pósa, and Bondy [Chv72].

Theorem 2.6 (Bondy)**.** Let G be a graph of order n ($n \geq 3$). If the degree sequence of G satisfies: $d_i \leq i$ and $d_j \leq j$, where $i \neq j$, implies $d_i + d_j \geq n$, then G is hamiltonian.

Theorem 2.7 (Chvátal [Chv72])**.** Let G be a graph of order n ($n \geq 3$). If the degree sequence of G satisfies: $d_k \leq k < \frac{n}{2}$ implies $d_{n-k} \geq n - k$, then G is hamiltonian.

Another useful necessary and sufficient condition for a graph to be hamiltonian is based on a graph's closure. The *k-closure* of a graph G is obtained by recursively joining pairs of nonadjacent vertices u and v with $d(u) + d(v) \geq k$ by an edge, until no more edges can be added. Note that the degrees are updated whenever an edge is added, and that the order in which edges are added has no effect on the final result. The *closure* of a graph of order n refers to the n-closure. The theorem shown in Theorem 2.9, known as the Bondy-Chvátal theorem, now shows how the closure of a graph can be used to check the hamiltonicity, for example using Corollary 2.10. The proof of this theorem makes use of the following result by Ore.

Theorem 2.8 (Ore)**.** Let G be a graph with n vertices and u, v be distinct nonadjacent vertices of G with $d(u) + d(v) \geq n$. Then G is hamiltonian if and only if $G + (u, v)$ is hamiltonian.

Theorem 2.9 (Bondy-Chvátal)**.** A graph is hamiltonian if and only if its closure is hamiltonian.

Corollary 2.10. Let G be a graph on $n \geq 3$ vertices. If G has a complete n-closure (meaning the n-closure is K_n), then G is hamiltonian.

Our construction for creating a nonhamiltonian graph G is based on the duplication of a smaller graph H (see Section 2.3). The absence of a Hamilton cycle in the produced graph G relies on the nonexistence of a Hamilton path between two vertices u and v in H. It is also useful for our research to look at graphs where such a path does not exist for any pair of vertices u and v.

A graph is *hamiltonian-connected* if there exists a Hamilton path from u to v for every pair of distinct vertices u, v in G. For most of the discussed sufficient conditions for being hamiltonian, there are similar conditions for the sufficiency of being hamiltonian-connected. Below some of these are listed. We list the easy to compute equivalents of Dirac and Ore's degree-based conditions, and a sufficient condition on the number of edges of the graph.

Theorem 2.11 (Dirac)**.** Let G be a graph of order n ($n \geq 3$). If $\delta(G) \geq \frac{n+1}{2}$, then G is hamiltonian-connected.

Theorem 2.12 (Ore)**.** Let G be a graph of order n ($n \geq 3$). If $d(u) + d(v) \geq n + 1$ for every pair u, v of distinct nonadjacent vertices of G, then G is hamiltonian-connected.

Theorem 2.13 (Ore)**.** Let G be a graph of order n ($n \geq 3$). If G has k edges such that $k \geq \binom{n-1}{2} + 3$, then G is hamiltonian-connected.

The least restrictive sufficient condition is again based on the closure of the graph.

Theorem 2.14. Let G be a graph on n vertices, and let u and v be distinct nonadjacent vertices of G with $d(u) + d(v) \geq n + 1$. Then G is hamiltonian-connected if and only if $G + (u, v)$ is hamiltonian-connected.

Theorem 2.15. Let G be a graph on n vertices. If G has a complete $(n + 1)$-closure, then G is hamiltonian-connected.

Other important necessary conditions and sufficient conditions for hamiltonicity are based on the toughness of a graph. These are shown in the next section, after the definition of toughness has been presented.

2.2 Toughness

The concept of toughness is originally introduced by Chvátal. According to Chvátal, "It measures in a simple way how tightly various pieces of a graph hold together" [Chv73]. It is related to the connectivity of a graph and is defined using the following definition. Recall that $\omega(G)$ denotes the number of components of a graph G.

Definition 2.16 (t-tough)**.** A graph G is t-tough ($t \in \mathbb{R}, t \geq 0$) if $|S| \geq t \cdot \omega(G - S)$ for every subset S of $V(G)$ with $\omega(G - S) > 1$.

So a t-tough graph G cannot be split into k different connected components by the removal of fewer than $t \cdot k$ vertices. By setting k to 2, the more commonly known definition of connectivity is implied. Namely that every t-tough graph is $\lceil 2t \rceil$-connected. The *toughness* of a graph G, denoted by $\tau(G)$, is the maximum value of t for which G is t-tough. Since the above definition cannot be applied to complete graphs, we need a separate definition. The toughness of complete graphs is defined to be infinite: $\tau(K_n) = \infty$ for every n. A graph is disconnected if and only if its toughness is zero. The following useful property is easy to check.

$$\text{If } G \text{ is a spanning subgraph of } H, \text{ then } \tau(G) \leq \tau(H).$$

The paper where toughness was introduced by Chvátal is centred around the importance of toughness for the existence of Hamilton cycles [Chv73]. The observation that a cycle graph C_n (that consists of a single cycle on $n \geq 3$ vertices) is 1-tough in combination with the above property, leads to the following result.

Theorem 2.17. Every hamiltonian graph is 1-tough.

The converse of Theorem 2.17 does not hold, as is shown by the graph in Figure 2.3. It is easy to check that this graph is 1-tough, by checking all cut sets and the number of components that result from their removal. The lack of a Hamilton cycle can be concluded from the common neighbour of the

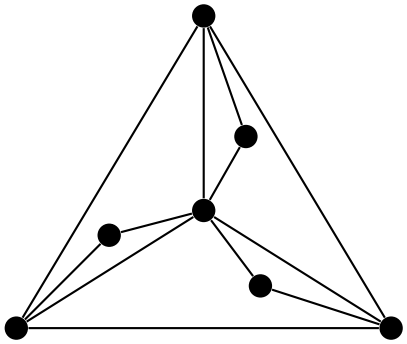


Figure 2.3: A 1-tough nonhamiltonian graph.

three vertices with degree 2: this prevents that these four vertices together end up in a cycle. This raises the question whether a stricter condition on the toughness could guarantee the existence of a Hamilton cycle. In [Chv73], Chvátal put up the following conjecture.

Conjecture 2.18. There exists a real number t_0 such that every t_0-tough graph is hamiltonian.

As he devised a nonhamiltonian graph with toughness $\frac{3}{2}$ himself, he also added the following conjecture.

Conjecture 2.19. Every t-tough graph with $t > \frac{3}{2}$ is hamiltonian.

In 1978 Thomassen proved that there exist infinitely many nonhamiltonian graphs G with $\tau(G) > \frac{3}{2}$ [Ber78]. This led to the new conjecture that every 2-tough graph is hamiltonian. If this conjecture were true, it would imply the following theorem and conjectures [BBV00].

Theorem 2.20 (Fleischner 1974)**.** The square of every 2-connected graph is hamiltonian.

Conjecture 2.21 (Matthews & Sumner 1984)**.** Every 4-connected claw-free graph is hamiltonian.

Conjecture 2.22 (Thomassen 1986)**.** Every 4-connected line graph is hamiltonian.

This turned out not to be the case, as the conjecture that every 2-tough graph is hamiltonian has been disproved by Bauer, Broersma and Veldman [BBV00]. This has been achieved by showing that there exist $(\frac{9}{4} - \epsilon)$-tough graphs for arbitrarily small $\epsilon > 0$ without a Hamilton path. The counterexample to the conjecture that every 2-tough graph is hamiltonian has been created by a construction based on the duplication of a smaller graph. As this construction seems promising to construct other nonhamiltonian graphs with a higher toughness, it is further explained in Section 2.3.

Since Chvátal introduced toughness, it has been subject to a lot of research. Most of this research was focused on several conjectures published by Chvátal, mainly relating toughness conditions to the existence of cycle structures [BBS06]. Conjecture 2.18 is still an open question, which is the motivation for this project. As with hamiltonicity, calculating a graph's toughness is NP-hard. The decision problem whether a graph is t-tough is co-NP-complete for every fixed positive rational t [BHS90]. A recent overview of Chvátal's conjecture and related problems can be found in a survey by Broersma [Bro15]. Many other results regarding toughness have been collected in a survey by Bauer, Broersma and Schmeichel [BBS06]. The survey lists theorems related to toughness and circumference, toughness and factors, the toughness of special graph classes, the computational complexity of toughness, and plenty of other results. As the survey is very extensive, we shall only discuss a few noteworthy results. An interesting result is that Conjecture 2.18 is true for the class of graphs having $\delta(G) \geq \epsilon \cdot n$ for any fixed $\epsilon > 0$. This is a consequence of the following theorem.

Theorem 2.23. Let G be a t-tough graph on $n \geq 3$ vertices with $\delta > \frac{n}{t+1} - 1$. Then G is hamiltonian.

2.2.1 Other graph classes

For many other graph classes, it has also been proven that Conjecture 2.18 holds. For example, every t_0-tough planar graph is hamiltonian if $t_0 > \frac{3}{2}$. A graph is planar if it can be drawn in a plane such that its edges intersect only at their ends. This result for planar graphs is best possible in the sense that there exist nonhamiltonian planar graphs with toughness $\frac{3}{2}$. For claw-free graphs, it is known that the conjecture is true for $t_0 = \frac{7}{2}$. A graph is claw-free if it does not have the complete bipartite graph $K_{1,3}$ as an induced subgraph. Another graph class consists of the chordal graphs, which are those graphs that do not contain an induced cycle of length four or more. For chordal graphs, it is known that Conjecture 2.18 is true, but the best possible result is not yet known. In 1997 it was proven that every 18-tough chordal graph is hamiltonian [Che+98]. This result has been improved recently, by proving that every 10-tough chordal graph is hamiltonian [KK17]. The paper introducing nonhamiltonian graphs with toughness approaching $\frac{9}{4}$ also applied their construction to chordal graphs, as this construction preserves the property of being chordal. They have shown that there exist $(\frac{7}{4} - \epsilon)$-tough chordal graphs for arbitrarily small $\epsilon > 0$ without a Hamilton path [BBV00].

2.3 Constructing tough nonhamiltonian graphs

As the construction method by Bauer, Broersma and Veldman is essential for our results, it is explained here. For the sake of completeness, we also repeat the proof from [BBV00] as it is rather short and very insightful.

Let H be a graph and x, y two vertices of H. The graph $G(H, x, y, \ell, m)$ ($\ell, m \in \mathbb{N}$) is defined as follows. The graph H is copied m times. These copies are called $H_1, \ldots, H_m$, and x_i, y_i refer to the vertices in H_i corresponding to x, y in H ($i = 1, \ldots, m$). F_m is the graph obtained by taking the disjoint union of $H_1, \ldots, H_m$, and by adding an edge between each pair of vertices in the set $\{x_1, \ldots, x_m, y_1, \ldots, y_m\}$. The graph G is the join of F_m and K_ℓ (the complete graph of order ℓ). The following theorem now shows how to use this construction to build a nonhamiltonian graph.

Theorem 2.24. Let H be a graph and x, y two vertices of H that are not connected by a Hamilton path of H. If $m \geq 2\ell + 3$, then $G(H, x, y, \ell, m)$ is nontraceable.

Proof. This theorem is proven using a proof by contradiction. Assume that $G(H, x, y, \ell, m)$ contains a Hamilton path P. The intersection of P and F_m consists of a collection $\mathbb{P}$ of at most $\ell + 1$ disjoint paths, which together contain all vertices in F_m. As $m \geq 2\ell + 3 = 2(\ell + 1) + 1$, there is a subgraph H_{i_0} in F_m such that no end vertex of a path in $\mathbb{P}$ lies in H_{i_0}. As H_{i_0} is a copy of H, there cannot be a Hamilton path in H_{i_0} from x_{i_0} to y_{i_0}. The intersection of P with H_{i_0} covers all vertices of H_{i_0} and must start in x_{i_0} or y_{i_0} and end in the other, as these are the only vertices connected with other copies of H. This contradicts with the statement that there cannot be a Hamilton path in H_{i_0} from x_{i_0} to y_{i_0}. □

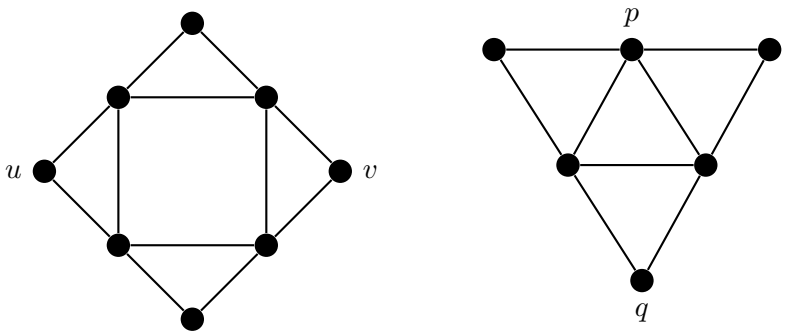
As G is nontraceable, it is clearly also nonhamiltonian. In order to prove that G is nonhamiltonian it is sufficient if $m \geq 2\ell + 1$, as is shown in Theorem 2.25. The proof is very similar to that of Theorem 2.24.

Theorem 2.25. Let H be a graph and x, y two vertices of H that are not connected by a Hamilton path of H. If $m \geq 2\ell + 1$, then $G(H, x, y, \ell, m)$ is nonhamiltonian.

This construction is then applied to the graphs shown in Figure 2.4. Figure 2.4a shows the graph L, such that $\tau(G(L, u, v, \ell, m)) = \frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, if ℓ and $m = 2\ell + 1$ are sufficiently large. This is a consequence of the following theorem (again from [BBV00]), where L refers to the graph from Figure 2.4a, by choosing sufficiently large ℓ and m.

Theorem 2.26. For $\ell \geq 2$ and $m \geq 1$,

$$\tau(G(L, u, v, \ell, m)) = \frac{\ell + 4m}{2m + 1}$$



(a) The general graph L. (b) The chordal graph M.

Figure 2.4: Two graphs used to construct tough nonhamiltonian graphs.

Figure 2.4b shows the chordal graph M. A similar result shows that $\tau(G(M,p,q,\ell,m)) = \frac{\ell+3m}{2m+1}$ if $\ell \geq 2$. As a consequence, $\tau(G(M,p,q,\ell,m)) = \frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, if ℓ and $m = 2\ell + 1$ are sufficiently large.

Chapter 3

Evolutionary algorithms

An evolutionary algorithm is an optimisation technique inspired by biological evolution. An evolutionary algorithm is based around a population of individuals. These individuals can reproduce themselves to generate offspring. This can be achieved by either mutation or crossover and recombination. Thereafter, a selection procedure is used to reduce the size of the population, based on the fitness of an individual. As the process of evolution is abstracted away, it may not do a good job at describing evolution in nature, but the resemblance to it can be seen by the jargon that is used. This group of algorithms could be effective in the domain of graph theory, as the evolutionary operators can be applied directly on the graphs. This direct encoding may make it possible to perform more meaningful operations, as opposed to a grammatical encoding.

Historically, the field of evolutionary algorithms can be divided into three paradigms [Jon06]. The first is evolutionary programming; it is based on a population of a fixed size where each parent produces exactly one offspring. The second paradigm is called evolution strategies, where a popular strategy is the $(1 + \lambda)$ model. This approach uses a single parent that produces λ offspring. Afterwards, the fittest individual among the offspring as well as the single parent is chosen as the new parent for the next generation. The final paradigm is genetic algorithms, which has more focus on application-independent algorithms. Individuals are always represented as a fixed-length binary string, such that all algorithms could use the same type of mutation and crossover. Even though these paradigms have been studied separately, they are instances of the same abstract evolutionary algorithm. This unified view on evolutionary algorithms will be explained below, based on the theory in the textbook by Jong [Jon06].

3.1 Algorithm design

All evolutionary algorithms are based on a population of size m that evolves in each iteration of the algorithm. During an iteration, the current population reproduces itself and produces n offspring. A selection procedure is used to reduce the population from $m + n$ to m individuals. In order to apply an evolutionary algorithm, one should have a method to represent an individual and a fitness measure to quantify how good an individual functions as a solution to the problem being solved. In order to design an algorithm, one also has to decide how to select the parents, how to select the survivors, and how to generate the offspring. For all three aspects, one should also determine the number of individuals that are being selected or generated in each step. We shall go over each element of the evolutionary algorithm and discuss the different options one has in designing an algorithm.

A method for representing individuals. An important choice in designing an evolutionary algorithm is in the encoding of the problem. It is common to use an indirect encoding, simply because the problem's solution does not permit evolutionary operators to be applied to it. This difference between the actual solution and its representation is named the genotype-phenotype distinction, where the former is the representation and the latter the actual solution. A classic genotypic representation is a binary string. An alternative approach is to ignore the distinction between the genotype and phenotype, and to use a direct encoding.

A disadvantage of an indirect encoding is that the variation at the genotype level may not correlate with the variation at the phenotype level, which introduces a bias into the algorithm. This bias is not present when a direct encoding is being used. As mentioned before, the possibility of direct encoding on graphs is what motivated this research on evolutionary algorithms. The disadvantage is that reproductive operators are now problem-specific. Thus there are no general solutions regarding the applicability of an operator and its corresponding parameters.

A parent population of size m. Increasing the population size will improve the possibility for parallel search. Having a larger population size will increase the likelihood that the global optima will be explored, and hence serves as a mechanism for reducing the variance due to convergence to local optima. One downfall of evolution strategies, which generally have small populations, is the risk of getting stuck at a local optimum. As evolutionary algorithms are stochastic, and the result will be different each time, it is hard to determine what population size is needed to have a sufficiently low variance. This partly depends on the number of local optima in the fitness landscape.

An offspring population of size n. An important trade-off in designing an evolutionary algorithm is the balance between exploitation and exploration. If n is relatively high, the current population is quickly replaced, and new regions will be explored quickly. If there are few offspring instead, the parent population will continue its current search for a longer time. Having high exploration could result in a quicker convergence, at the risk of having a population of individuals that are stuck in a local optimum.

Selection methods to decide which parents reproduce and which offspring survives. There are multiple ways to decide which parents reproduce or which offspring survives. At least one of these should be based on a fitness function that evaluates how good an individual is in solving the original problem. Selection methods can be either deterministic or stochastic. The evolutionary programming algorithms have a deterministic approach for parent selection, as each individual produces exactly one offspring. Genetic algorithms have a stochastic selection procedure, where parents are chosen according to a fitness-based probability distribution. Such fitness-based selection methods can take multiple forms. Common selection methods are truncation selection, linear ranking, uniform selection, fitness-proportional selection, and tournament selection.

Truncation selection always chooses the fittest individuals. Linear ranking uses a probability distribution where the more fit individuals have a higher probability of being selected, whilst the uniform selection does not take fitness into account and gives each individual an equal probability of being selected. Tournament selection picks k candidates uniformly and selects the fittest individual among those k as the winner. Finally, fitness-proportional selection is somewhat similar to the linear ranking as more fit individuals have a higher probability of being chosen. However, instead of using a linear distribution it makes use of a dynamic distribution based on the current fitness of the individual compared to the total fitness of all individuals. This distribution thus gets more uniform, as the population converges.

As with the other design decisions in building an evolutionary algorithm, there is a trade-off between exploitation and exploration. Having an elitist selection scheme where only the fittest individuals are selected, the algorithm may quickly converge to a local optimum. It is therefore common that either the parent selection or the survival selection uses a uniform selection method. In deciding which offspring survives, the selection methods can again be categorised in two groups. The first group consists of the non-overlapping generation models, where all parents die after reproduction. The alternative model is the overlapping generation model, where the parents compete with their offspring for survival. Having overlapping models will increase the exploitation of the current search space, resulting in early convergence. On the other hand, the non-overlapping-generation models

Algorithm	parent selection	survival selection
Evolutionary programming	uniform	truncation
Evolution strategies	uniform	truncation
Genetic algorithms	fitness-proportional	uniform

Table 3.1: An overview of the selection methods in evolutionary programming, evolution strategies, and genetic algorithms.

have the disadvantage that it is quite possible to lose some of the fittest individuals from the population, especially using stochastic selection. An overview of the selection methods used in the three discussed paradigms is shown in Table 3.1.

A set of reproductive operators In order to produce offspring, parents can reproduce in multiple ways. One method to reproduce is by mutation, where a single parent clones itself and modifies one or more genes in a stochastic way. When using a fixed-length array with length L of real-valued numbers as genotypic representation, a common mutation method is the Gaussian mutation operator. This operator performs a mutation using a normal distribution with mean 0. It mutates on average 1 gene, according to the given standard deviation. As we use a direct encoding, we have to restrict ourselves to operators that can be applied directly on graphs. Some obvious mutations are the addition of a new edge, the removal of an edge, the addition of a vertex, and the removal of a vertex. An alternative mutation is edge contraction, where an edge e is removed, and its adjacent vertices are merged. Another mutation can be obtained by 'moving' an edge such that it stays incident to one vertex whilst changing the other end of the edge to a new vertex. Other mutations are also possible, as there are more possibilities for the merging and splitting of vertices.

Another method to produce offspring is by recombination, where two parents are partially cloned and then combined to form a new individual. When evolutionary algorithms use a fixed-length array of numbers, which is a standard encoding for optimisation problems, this combination could be done quite easily using 1-point crossover. In this case, the first part of the array is taken from the first parent, and the second part is taken from the second parent using a randomly selected crossover point. This method can be improved by taking multiple crossover points, such that multiple segments are copied alternately from each parent. Even the amount of crossover points can be made stochastic, to avoid the so-called distance bias where the distance between genes in the array influences the probability of them being inherited together. Like the mutation operator, the implementation is very different when using a direct graph encoding. Compared to linking

two partial arrays together, the recombination of different graphs is far from trivial.

In [SPC04], two different crossover techniques are compared. The first one is the Globus crossover, named after the author [Glo+00], which is an operator that divides each parent into disjoint connected subgraphs. Afterwards, two connected subgraphs from two different parents are merged together to create a new graph. The Globus crossover operator is "representative of the best work in this area" of fragmentation and recombination operators [SPC04]. The graph is decomposed by an edge cut, such that the graph is split into two components that together form a spanning subgraph of the original graph. An edge cut of a connected graph is a set of edges whose removal disconnects the graph. The division of a parent into two connected subgraphs is shown in Algorithm 3.1.

Algorithm 3.1 Split a graph into two components

```
1: initialise an empty set S
2: choose a random edge e
3: while S is not an edge cut do
4:     find the shortest path between the vertices incident with e
5:     remove a random edge in this shortest path from the graph
6:     add the removed edge to S
```

In order to merge two components from two different parents, one component is chosen from each parent at random. The combination of this component together with the edge cut that is used to split the original graph is called a fragment. The algorithm to merge these components is shown in Algorithm 3.2. Note that the largest fragment refers to the fragment, whose (randomly selected) component contains the most vertices. The smaller fragment refers to the fragment corresponding to the (randomly selected) component from the other parent.

Algorithm 3.2 Merge two fragments

```
1: for each broken edge e from the largest fragment's edge cut do
2:     if the other fragment has broken edges in its edge cut then
3:         merge e with a random broken edge from the smaller fragment
4:     else
5:         if a random coin flip turns head then
6:             attach e to a random node in the smaller fragment
7:         else
8:             discard the broken edge e
```

Both algorithms are designed in such a way to reduce the bias towards selecting certain vertices or edges. Nevertheless, the Globus operator does have has a bias to split graphs into two components, such that one component

contains more vertices than the other. This is a consequence of the fact that the edges that are incident with the initially chosen edge e have a greater probability of breaking than any other edges in the graph. As the path is disconnected by removing a random edge in the path (line 5 of Algorithm 3.1), the algorithm also tends to destroy the structure of fragments more than necessary in order to split the graph into two components. According to [Glo+00], it can be expected that fit parents reproduce very unfit children, due to the very destructive nature of the crossover operator.

An alternative crossover mechanism is the GraphX crossover. This operator tries to perform the same point crossover technique as used on fixed-length arrays of numbers, by using the adjacency matrix of a graph. Note that this research uses directed graphs and that this operator would not preserve the symmetry of the adjacency matrix that is present in undirected graphs without modifying the algorithm. On graphs of equal order, it uses a 2-point crossover based on two randomly selected points. When applied to graphs of a different order, all-zero columns and rows are added to the smaller matrix in order to obtain symmetric square matrices of the same dimension.

The values of the adjacency matrix in between the two crossover points are then swapped, such that two offspring are produced. After the swapping, the dimension of the adjacency matrix for the smaller graph is restored by removing the same columns and rows that were added (possibly containing nonzero entries).

These two crossover mechanisms are compared in the paper that introduces the GraphX operator [SPC04], but in our opinion, it is hard to draw meaningful conclusions due to the setup of the experiment. The goal is to evolve towards a predefined target graph, but isomorphism is not taken into account. This gives the GraphX operator an advantage, as it performs operations on the representation instead of the graph structure itself. Even though the GraphX operator clearly outperforms the Globus operator on this problem, there is little evidence that it would work well on other problems. There has not been much research on this topic ever since, and the mentioned paper [SPC04] is quite old and has never been actually published. The method thus does not seem very impactful, and relying solely on the mutation operator seems more promising. This approach of reproduction using only the mutation operator is also successfully used in the field of Cartesian genetic programming. As this method partly inspired our approach, it will be explained in the following section.

3.2 Cartesian genetic programming

A related subject in evolutionary algorithms that has been popular recently is the field of Cartesian genetic programming. Genetic programming refers to the set of evolutionary algorithms that are applied to computer programs, such that the programs evolve over time. Classical genetic programming uses trees to represent the program [Koz93]. The mutation operator can simply change a node in the tree, but even an effective crossover operator can be implemented easily by selecting two nodes and swapping them, including their subtrees. An advantage of this crossover technique is that it succeeds well in preserving the sub-structures that are present. A drawback of tree-based genetic programming is the occurrence of bloat. Bloat is the phenomenon where a solution becomes increasingly larger, whilst there is no significant increase in its fitness. A solution is thus bloated when it is unnecessarily complicated.

Cartesian genetic programming is a variant to genetic programming that uses directed acyclic graphs as representation. This enables the reuse of nodes, which significantly reduces the amount of bloat [SL07]. What makes Cartesian genetic programming interesting is the lack of a crossover operation, as it only makes use of a mutation operator. Due to its representation, it is particularly well suited for problems with multiple inputs and outputs and has been applied successfully in a variety of domains ranging from robot controllers, to real-value optimisation problems, digital circuits, and plenty more [MR19]. Cartesian genetic programming has been well studied, and many variants exist.

Multiple crossover techniques have been tried, and even though some of them look promising there is not yet a crossover operation that is widely accepted into Cartesian genetic programming. Whilst the graphs in our research are neither directed nor acyclic, the fact that no such operator exists for these directed acyclic graphs strengthens our belief that the crossover operator will not be very effective on our problem.

Chapter 4

Method

Section 4.1 looks at the generation of nonisomorphic graphs and explains a very compact data format to store graphs. Section 4.2 covers the efficiency of different options to represent a graph on a computer. In Section 4.3 and Section 4.4, we respectively look at the algorithms required to find a Hamilton cycle and to determine the toughness of a graph. Section 4.5 shows the design of our enumeration algorithm by presenting a basic algorithm and discussing various improvements to increase the speed. Section 4.6 presents the final version of the enumeration algorithm, where we discuss the implementation. In Section 4.7, the evolutionary algorithm is presented. Section 4.8 discusses how to verify the results.

4.1 Nonisomorphic graph generation

In order to implement an enumeration algorithm, a set of graphs to enumerate over is required. This set should not include isomorphic graphs, as this would lead to duplicate calculations. Disconnected graphs can also be disregarded, as these have a toughness of zero. The set of graphs should contain all connected nonisomorphic graphs of a certain order n. The number of nonisomorphic connected graphs of order $1 \leq n \leq 14$ is shown in Table 4.1. The table also shows the number of connected chordal graphs, as these will be analysed separately.

Up to now, there is no polynomial algorithm to solve the decision problem that asks whether two graphs are isomorphic. The complexity of this problem is open and one of the most prominent open problems within graph theory and computational complexity. This makes the task of calculating all nonisomorphic graphs of a certain order challenging. An exact and reasonably fast algorithm is presented in [McK98], where it has also been proven that all possible graphs are generated using this method. Note that the presented algorithm does have an exponential time complexity.

n	connected graphs
1	1
2	1
3	2
4	6
5	21
6	112
7	853
8	11,117
9	261,080
10	11,716,571
11	1,006,700,565
12	164,059,830,476
13	50,335,907,869,219
14	29,003,487,462,848,061

n	chordal graphs
1	1
2	1
3	2
4	5
5	15
6	58
7	272
8	1,614
9	11,911
10	109,539
11	1,247,691
12	17,566,431
13	305,310,547
14	6,558,690,953

Table 4.1: Respectively the number of connected graphs [Inc20a] and connected chordal graphs with n vertices [Inc20b].

This algorithm is implemented as part of the Gtools suite included in Nauty [MP14], which is a program that computes automorphism groups of graphs. The program that implements this is named Geng. Geng provides the option to only generate graphs that satisfy specific properties, such as being connected, being chordal, or having a minimum or maximum degree. The output of Geng for some common classes of graphs can be downloaded from the personal website of Brendan McKay [McK20a]. The number of connected graphs from Table 4.1 is available for $n \leq 11$, and the number of connected chordal graphs is available for $n \leq 13$. Alternatively, we could make use of the house of graphs database [Bri+13]. This website lists many classes of graphs and also provides a database of interesting graphs.

4.1.1 Graph6 format

Geng returns a set of graphs that are stored in the graph6 format, which is a very compact format to store graphs [McK20b]. It achieves this compact form by encoding the graph in ASCII characters. It only consists of bytes whose value lie in the range from 63 to 126. In our explanation of this format, it is assumed that a graph has at most 62 vertices. The encoding works as follows.

The first character encodes the order n of the graph. It is encoded as the byte $n + 63$. For the example graph seen in Figure 4.1, this would have the value 68. The remaining characters encode half of the adjacency matrix (see Section 4.2 for a definition) of the graph. The upper triangle of the

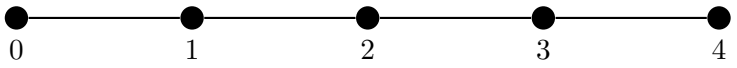


Figure 4.1: A path graph to illustrate the graph6 format.

adjacency matrix of a graph is written as a bit vector. This upper triangle is traversed column by column, as described by the following order.

$$(0,1),(0,2),(1,2),(0,3),(1,3),(2,3),\ldots,(n-1,n)$$

For our example, the bit vector would be 1010010001. This bit vector is padded with zeros on the right if needed, such that its length becomes a multiple of 6. It is then split across groups of 6 bits each. Our example would then be denoted as 101001 000100. Finally, these groups are interpreted as binary numbers and increased by 63. For our example, this would lead to 104 67. Combined with the order of the graph, Figure 4.1 can be denoted by the bytes 68 104 67, corresponding with the ASCII characters "DhC".

4.2 Graph representations

A graph G of order n can be represented in multiple ways. One possibility is to use an edge list, similar to the formal definition of simple graphs (Definition 2.2). This is a single list containing all the edges of a graph. It is straightforward to implement and requires relatively little space, but testing whether an edge exists or finding the neighbours of a vertex is not time efficient. Two common representations to store graphs on a computer are adjacency matrices and adjacency lists.

An adjacency matrix is the $n \times n$ matrix $A_g := (a_{uv})$, where a_{uv} is defined as follows.

$$a_{uv} = \begin{cases} 1 & \text{if } (u,v) \in E(G) \\ 0 & \text{otherwise} \end{cases}.$$

The main benefit of an adjacency matrix is constant time access to test whether two vertices are adjacent. A disadvantage is that the neighbours of a vertex cannot be listed efficiently, but require linear time in the number of vertices of a graph. Note that only half of the adjacency matrix has to be stored for undirected graphs, as the matrix is symmetric.

An alternative is to use an adjacency list. For each vertex v of a graph, the neighbours $N(v)$ can be contained in a list. The list containing all these lists is called the adjacency list: $(N(v) : v \in V)$. As an example, the adjacency list for the graph of Figure 4.2 would be $[[2,3],[1,3],[1,2],[1]]$. It has the main benefit that it can list all neighbours of a graph in a time proportional to the degree of the vertices. It has the disadvantage that testing whether two vertices are adjacent also takes time proportional to the degree of the

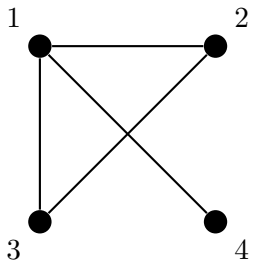


Figure 4.2: An example to explain the adjacency list.

vertices. Our implementation makes use of an adjacency list, the motivation for doing so is presented in Section 4.6.1.

4.3 Hamilton cycle algorithm

Two different algorithms for finding Hamilton cycles or paths will be discussed here. These two approaches are based on backtracking and dynamic programming. We shall focus on the algorithms for finding Hamilton paths instead of cycles, as these are used in our final algorithm. These can easily be modified into similar algorithms for Hamilton cycles (see Section 2.1). There are many heuristics related to finding Hamilton paths in very large graphs, but most available literature is not relevant in our scenario. Since finding a Hamilton path is NP-hard, a polynomial algorithm most likely does not exist. The algorithm with the lowest time complexity has complexity $O(1.657^{|V|})$. This has been proven in [Bjö10], and it provided the first superpolynomial improvement since 1962 to the exponential complexity of the dynamic programming solution developed independently by Bellman [Bel62] and Held and Karp [HK61]. Multiple other algorithms to find Hamilton cycles and variations on the above-mentioned algorithms are described in the thesis 'Finding Hamiltonian cycles: algorithms, graphs and performance' [Van98].

Backtracking The backtracking algorithm for finding a Hamilton path is shown as pseudocode in Algorithm 4.1. Note that the path variable is initially an empty vector. The backtracking algorithm has complexity $O(|V|!)$. Ideally, the graph is stored by an adjacency list such that the for loop can be performed efficiently. Possible improvements in the speed of the algorithm can be achieved by changing the order of the adjacent vertices that are analysed (line 6), for example, based on the degree. Both the above-mentioned thesis [Van98] and the thesis 'A Fast Algorithm For Finding Hamilton Cycles' [Cha08], suggest many heuristic-based improvements to the backtracking algorithm. We believe that these heuristics would not help

Algorithm 4.1 Backtracking Hamilton path

```
1:  function HAMILTON_PATH(graph, v_start, v_end, path)
2:      path.push_back(v_start)                    ▷ add to the current path
3:      if v_start = v_end and length(path) = order(graph) then
4:          return true
5:      else
6:          for all vertices w adjacent to v_start do
7:              if w is not in the current path then
8:                  if HAMILTON_PATH(graph, w, v_end, path) then
9:                      return true
10:     path.pop_back()          ▷ remove the last element from the path
11:     return false
```

much on the very small graphs we analyse, but an extensive comparison of performance has not been made.

Dynamic programming Alternatively, one could use an algorithm based on dynamic programming. This has time complexity $O(|V|^2 \cdot 2^{|V|})$. The advantage is that it enables the reuse of computations. This is especially beneficial if the algorithm is called multiple times on the same graph (between different vertices), which is exactly what we are doing in our enumeration algorithm (see Section 4.5). It works as follows.

Consider the problem of finding a Hamilton path in G from v_{start} towards v_{end} (both vertices in G). In order to use dynamic programming on this problem, we need to have a property that can be defined recursively. This is the following property: for a given set $S \subseteq V(G)$ and a vertex $v \in S$, there exists a path from v_{start} which goes through all vertices of S and ends at v. We call this property p and define it recursively as follows.

Definition 4.1. $p[S][v] = \exists u \in S : p[S - \{v\}][u] \wedge (u, v) \in E(G)$.

So a path exists for a combination of S and v if and only if v has an adjacent vertex u such that a path exists through $S - \{v\}$ ending at u. There is a Hamilton path from v_{start} towards v_{end} if $p[V(G)][v_{\text{end}}]$ is true.

Comparison The dynamic programming solution has the benefit that it can reuse computations and that it does not have a factorial time complexity. The backtracking algorithm has the benefit that it has a very good best-case performance. If it is, for example, applied to a complete graph, it will terminate on the first path being tried. The dynamic programming solution does not have this benefit. If that algorithm is applied to a complete graph K_n, it has to calculate $p[S][v]$ for all possible $S \subseteq V(K_n)$ before terminating. This difference in performance is valid to a lesser extent for other dense

graphs. Both algorithms have been implemented, and timing results can be found in Section 5.4. Our implementations showed that for small graphs, the backtracking algorithm performed better than the alternative. This backtracking algorithm is used both in our enumeration algorithm and the evolutionary algorithm.

4.4 Toughness algorithm

In Section 2.3 it has been explained how to construct a tough nonhamiltonian graph G, based on a smaller graph H, in which u and v are chosen such that there is no Hamilton path in H between u and v. Let H_i denote the copies of H in G for $i = 1, 2, \ldots, 2\ell + 1$, and let K denote the complete graph on ℓ vertices in G, so with $V(K) = V(G) \setminus (V(H_1) \cup V(H_2) \cup \ldots \cup V(H_{2\ell+1}))$. We assume that ℓ is large. Let S be a cut set of G for which $\tau(G) = \frac{|S|}{\omega(G-S)}$.

In order to determine the toughness of this constructed graph G, based on only the small graph H, it would be desirable that the cut set S can be chosen such that the vertices of S in each copy H_i all correspond to the same set of vertices in H. To show that this is indeed possible, we claim that the following holds.

Theorem 4.2. We can choose S in such a way that $V(H_i) \cap S$ corresponds to the same subset $T \subset V(H)$ for all $i = 1, 2, \ldots, 2\ell + 1$.

Proof. First of all, it is obvious that $V(K) \subset S$, since S is a cut set and all vertices of K are adjacent to all other vertices of G, so they have to be part of the cut set. Since $G - V(K)$ consists of one component, S contains a cut set of $G - V(K)$. Let $S_i = V(H_i) \cap S$, and let u_i and v_i denote the vertices of H_i that correspond to u and v of H. Then H_i contributes $\omega(H_i - S_i)$ components to $\omega(G - S)$ if both u_i and v_i belong to S_i, and $\omega(H_i - S_i) - 1$ components to $\omega(G - S)$ if at least one of u_i and v_i does not belong to S_i. We call these contributions of $H_i - S_i$ to $\omega(G - S)$ the private components of $H_i - S_i$, and denote their number by ω_i. All non-private components, if any, end up in one component of $G - S$, because all remaining vertices u_i and v_i in $G - S$ are mutually adjacent by the construction of G. Since every cut set of G gives an upper bound on $\tau(G)$, and the minimum of these upper bounds is reached by taking the cut set S, we have that

$$\tau(G) = \min \frac{\ell + \sum_{i=1}^{2\ell+1} |S_i|}{(1+) \sum_{i=1}^{2\ell+1} \omega_i},$$

where the minimum is taken over all possible combinations of cut sets S_i of H_i, and in which the additional $+1$ only applies if at least one of the S_i does not contain both u_i and v_i.

Let s denote the average of $|S_i|$ taken over all cut sets S_i, so

$$s = \frac{\sum_{i=1}^{2\ell+1} |S_i|}{2\ell + 1}.$$

Similarly, let ω denote the average of ω_i taken over all ω_i, so

$$\omega = \frac{\sum_{i=1}^{2\ell+1} \omega_i}{2\ell+1}.$$

With this notation, from the above it follows that

$$\tau(G) = \min \frac{\ell + (2\ell+1)s}{(1+)(2\ell+1)\omega} = \min \frac{1 + (2+\frac{1}{\ell})s}{(\frac{1}{\ell}+)(2+\frac{1}{\ell})\omega}.$$

If we take ℓ large enough, we can get this expression arbitrarily close (from below) to

$$\frac{1+2s}{2\omega}.$$

If there is a cut set $S_i \subset V(H_i)$ with $|S_i| = s$ and $\omega_i = \omega$ for some $i \in \{1, 2, \ldots, 2\ell+1\}$, then we can clearly choose all S_i corresponding to the same cut set $T \subset V(H)$, and there is nothing to prove.

Next assume that this is not the case. Consider all S_i and ω_i. By definition,

$$|S_1| + |S_2| + \ldots + |S_{2\ell+1}| = (2\ell+1)s$$

and

$$\omega_1 + \omega_2 + \ldots + \omega_{2\ell+1} = (2\ell+1)\omega.$$

We also have that

$$\frac{1+2s}{2\omega} < \frac{1+2s_i}{2\omega_i},$$

since otherwise choosing S_i for every H_i would contradict the assumption on the choice of S. Using the latter inequality, we get

$$\omega_i(1+2s) < \omega(1+2s_i),$$

and summing over all $i = 1, 2, \ldots, 2\ell+1$, this yields

$$(1+2s)(2\ell+1)\omega < \omega(2\ell+1)(1+2s),$$

a clear contradiction. This completes the proof of this theorem. □

It follows from the above proof that we can assume that there is a cut set $S_i \subset V(H_i)$ such that $|S_i| = s$ and $\omega_i = \omega$ for some $i \in \{1, 2, \ldots, 2\ell+1\}$. It has also been shown that the toughness can be approached from below by

$$\frac{1+2s}{2\omega}.$$

Let S_H be the vertices in H corresponding to the vertices in S_i and let $\omega'(H - S_H)$ be the number of components in $H - S_H$ not containing u or v. The toughness of G is now equal to

$$\frac{1+2|S_H|}{2\omega'(H-S_H)} - \epsilon$$

for arbitrarily small $\epsilon > 0$, by taking ℓ sufficiently large. Our algorithm approximates it as follows.

$$\tau(G) \approx \frac{1 + 2|S_H|}{2\omega'(H - S_H)} \tag{4.1}$$

Our goal is to write a reasonably fast exact algorithm that can calculate the toughness using Equation (4.1) for all possible nonisormorphic graphs of a certain order. A straightforward program to do so without any optimisations is described by the pseudocode in Algorithm 4.2. Note that the graphs G and H do not have the same meaning as above, but that in the below pseudocode G is the (small) input graph.

Algorithm 4.2 Toughness calculation

Require: vertices are labelled as $1, 2, \ldots, |V(G)|$ and $u, v \in V(G)$

1: **function** GET_TOUGHNESS(G, u, v)
2: $P \leftarrow$ GET_SUBSETS($|V(G)|$)
3: **for all** $S \in P$ **do**
4: $H \leftarrow$ SUBGRAPH(G, S) ▷ induced subgraph $G[S]$
5: $n_c \leftarrow$ CONNECTED_COMPONENTS(H, u, v)
6: ▷ count components excluding those containing u or v
7: **if** $n_c > 0$ **then**
8:

$$T \leftarrow \min(T, \frac{1 + 2 \cdot (|V(G)| - |V(H)|)}{2 \cdot n_c})$$

9: **return** T

The functions that are being used in this algorithm are listed below.

- `subgraph`: It should take a graph and a set of vertices in this graph as parameters and return the subgraph induced by this set of vertices.
- `get_subsets`: This should take a parameter n and return the powerset of $\{1, 2, \ldots, n\}$. The powerset is the set containing all $2^{|V|}$ possible subsets of a set V. It is therefore required that the vertices are labelled as $1, 2, \ldots, |V(G)|$.
- `connected_components`. It should take a graph and two vertices u, v in the graph as input, and return the number of components excluding those containing u or v. This has complexity $O(|V| + |E|)$.

4.5 Algorithm design

In order to implement an algorithm that can apply the construction from Section 2.3, the algorithm for the Hamilton path and the algorithm that determines the toughness need to be combined into a single algorithm. A simple algorithm that does this is shown in Algorithm 4.3. It returns whether a graph H can be used to construct a nonhamiltonian graph G with toughness above 2. A second algorithm should then loop over all nonisomorphic graphs of a certain order generated by Geng (see Section 4.1) and apply `basic_algorithm` to each graph. As Algorithm 4.3 is not very efficient, we will discuss many possible improvements to this algorithm in the following subsections.

Algorithm 4.3 Basic algorithm

```
1: function BASIC_ALGORITHM(G)
2:     for all u, v ∈ V(G) : u ≠ v do
3:         h ← HAMILTON_PATH(G, u, v)
4:         if not h then
5:             t ← GET_TOUGHNESS(G, u, v)
6:             if t ≥ 2 then
7:                 return true
8:     return false
```

Before looking at possible optimisations, the worst-case time complexity of Algorithm 4.3 is analysed. The for loop (line 2) iterates over $\frac{|V|\cdot(|V|-1)}{2}$ possible pairs: $O(|V|^2)$. The toughness calculation (Algorithm 4.2) iterates over at most $2^{|V|}$ subsets. For each subset, calculating the number of components has the highest time complexity, namely $O(|V| + |E|)$. The complexity of Algorithm 4.2 is thus $O(2^{|V|} \cdot (|V| + |E|))$. Checking the existence of a Hamilton path using backtracking has complexity $O(|V|!)$. A factorial function quickly grows larger than the exponential function with base 2 (for $n \geq 4, n! > 2^n$), and the constant time factor $|V| + |E|$ does not change this. So the Hamilton path algorithm has the dominating complexity, making $O(|V|^2 \cdot |V|!)$ the worst-case time complexity of Algorithm 4.3.

We have analysed the efficiency of the following optimisations in multiple ways. For some optimisations a theoretical argument suffices, because they do not add any new computations and reduce the existing amount of required computations. For other optimisations, this is less trivial. One approach to analyse this is by benchmarking, either by solving all graphs of a specific order or by analysing a smaller set of larger graphs. Another option is to use a profiler that can provide dynamic program analysis by measuring performance whilst running the code. This could create a call graph, showing how many times each subroutine has been called, or keep track how much time is spent in each subroutine.

4.5.1 Toughness algorithm termination

The first improvement is based on the fact that we are not interested in the actual toughness, but only whether it is higher than a fixed value t_0. For Algorithm 4.3 this value is set to two ($t_0 = 2$). It may be more efficient to set this value to $\frac{9}{4}$, but we are also interested in the structure of other graphs with toughness $\tau \geq 2$. By altering the toughness calculation, the algorithm is a lot more efficient on graphs with low toughness. After line 8 in Algorithm 4.2, we can insert the following code to terminate as soon as possible.

if $T < 2$ **then**
    return T

Note that this enables the algorithm to return an incorrect toughness if the toughness is below two, but for our usage in Algorithm 4.3 this would not matter.

4.5.2 Cut size in the toughness algorithm

The second improvement is also based on the fact that it only matters whether the toughness is higher than a fixed value t_0. For the remaining part of this thesis, the term cut refers to a set $S \subseteq V(G)$ used to calculate the toughness. The term cut size refers to the number of vertices of S. Recall from Section 4.4 that the toughness can be approximated by

$$\tau(G) \approx \frac{1 + 2|S_H|}{2\omega'(H - S_H)}.$$

Recall that this approximation is approached from below, thus in order to test whenever $\tau(G) < t_0$, a cut $S \subseteq V(G)$ should satisfy the property:

$$\frac{1 + 2 \cdot |S|}{2 \cdot \omega'} < t_0.$$

This yields a formula to calculate what size of S can produce an upper bound on the toughness that is below t_0. As Algorithm 4.3 uses $t_0 = 2$, this value will be used from now on for the sake of simplicity. The previous formula is equivalent to:

$$|S| < t_0 \cdot \omega' - \frac{1}{2}. \tag{4.2}$$

Since $|S|$ and ω' are both integer values, the possible cut size $|S|$ required to check whether $\tau(G) < 2$ satisfies

$$|S| \leq 2 \cdot \omega' - 1. \tag{4.3}$$

So only sets S of odd size have to be considered, as is shown by Theorem 4.3.

graph order	$\lvert S\rvert$ to analyse
4 or less	1
5,6, or 7	1 and 3
8,9, or 10	1,3, and 5
11,12, or 13	1,3,5, and 7

Table 4.2: Cut sizes required to test whether the toughness is below 2.

Theorem 4.3. Here we assume the above notation. $\tau(G) < 2$ if and only if $\min_{S \subseteq V(H): |S| = 2k+1 (k \in \mathbb{N})} (\frac{1 + 2 \cdot |S|}{2 \cdot \omega'}) < 2$.

Proof. ($\Longrightarrow$) Assume $\tau(G) < 2$. Then there exists a set $S \subseteq V(H)$ such that $\frac{1+2 \cdot |S|}{2 \cdot \omega'} < 2$. If $|S| = 2k + 1$ ($k \in \mathbb{N}$), the result follows. If $|S| = 2k$ ($k \in \mathbb{N}$), then any set T such that $|T| = |S|+1$ and $T \supseteq S$ satisfies $\frac{1+2 \cdot |S|}{2 \cdot \omega'} < 2$, as a consequence of Equation (4.3).

($\Longleftarrow$) This follows directly from the definition of toughness. □

Certain cut sizes $|S|$ cannot produce a toughness below two, simply because there are not enough vertices in $G - S$ remaining to have the required number of components not containing u or v. By combining this observation, with Theorem 4.3, the cut sizes that have to be iterated over in our algorithm are shown in Table 4.2.

Note that Equation (4.2) allows a more generalised theorem than Theorem 4.3. The only cuts $S \subseteq V(G)$ that need to be considered in order to test whether $\tau < t_0$ should satisfy that $|S| = \lim_{\epsilon \to 0^+} \lfloor t_0 \cdot k - \frac{1}{2} - \epsilon \rfloor$ for $k \in \mathbb{N}$. This generalisation can, for example, be used to apply this optimisation to chordal graphs.

4.5.3 Cut ordering in the toughness algorithm

Another improvement to Algorithm 4.2 can be made by iterating over the powerset in a different order. Using empirical evidence, we have concluded that it is most efficient to start with the largest cuts. For the remaining cut sizes, the algorithm iterates over them in decreasing order. It has also been tested whether all cuts of a certain size could be iterated over in a more efficient order. A random order did not perform better than a lexicographical order. Minor improvements have been obtained by applying different orders, but in order to avoid overfitting, we have decided not to use this modification.

4.5.4 Even fewer cuts in the toughness algorithm

This improvement is an extension to Section 4.5.2, but it only works for graphs of a certain order. Assume a graph of order $n = 3k - 1$ ($k \in \mathbb{N}$). As a consequence of the previous improvement, the cuts of size $2k - 1$ are the

first to be analysed. Using a cut S of size $2k-1$, one has to find k distinct components that do not contain u and v. Since there are only k vertices left, u and v must be included in the cut S. This reduces the number of possible cuts of this size from $\binom{3k-1}{k}$ to $\binom{3k-3}{k}$.

As an example: for a graph of order 8 one has to find 3 disconnected vertices with a cut of size 5. This reduces the possibilities for this remaining set from $\binom{8}{3} = 56$ to $\binom{6}{3} = 20$. For graphs of order 11 one has to find 4 disconnected vertices with a cut of size 7, this leads to a reduction from $\binom{11}{4} = 330$ to $\binom{9}{4} = 126$.

More generally, the relative reduction in the amount of sets is given by the following formula:

$$\frac{\binom{3k-3}{k}}{\binom{3k-1}{k}} = \frac{(2k-1)(2k-2)}{(3k-1)(3k-2)}$$

This reduction in the number of cuts can be implemented by passing u and v as an argument to the `get_subsets` function. Note that this cut size is tried first, because it can often be concluded that the toughness is below 2, by using cuts of this size.

4.5.5 Reuse of cuts within a graph

All calculations are repeated for each possible pair of vertices u, v in a graph. When a cut S contains both u and v and yields a toughness below 2, this cut also results in a toughness below 2 for other pairs of vertices $u', v' \in S$. When either u or v is not in the cut, the set may still be reusable if the graph has a relatively low toughness. The algorithm can thus be improved by keeping track of a set containing successful cuts. These cuts should then be tried before all other possible cuts returned by `get_subsets`.

4.5.6 Reuse of cuts across different graphs

The set of nonisomorphic graphs of a certain order that serve as input to Algorithm 4.3 are generated by Geng (see Section 4.1). Geng creates all nonisomorphic graphs of a certain order by iteratively adding edges. Due to the particular generation method being used, successively generated graphs are relatively similar. Inspired by the previous optimisation, it is also possible to reuse cuts of the previous graph. This could be implemented by creating a hashset that uses a pair of vertices as key and stores the final cut of the previous graph as value. The final cut is the cut that yields a toughness below 2 and terminates the algorithm (by the optimisation from Section 4.5.1). These last two optimisations on the reuse of cuts are relatively simple, but are very useful optimisations due to the fact that they are often applicable.

4.5.7 Reuse of the Hamilton path

Inspired by the reuse of cuts discussed in the previous optimisation, it is also useful to reuse Hamilton paths that have been found for the previous graph. This could be implemented similarly by creating a hashset. It uses a pair of vertices as key and stores a Hamilton path between this pair of vertices as value, if such a path has been found. For the next graph, it is checked whether this Hamilton path is still a valid path in the graph (i.e. each edge of the path is actually in the graph).

4.5.8 Preprocessing

If a graph is dense enough, there may exist a Hamilton path from u to v for every pair of distinct vertices u, v in G. As explained in Section 2.1, such a graph is called hamiltonian-connected. If a graph is hamiltonian-connected, it does not have to be analysed by our algorithm as it clearly cannot contain a solution. The densest graphs require most of the time in running our algorithm, so a significant improvement can be made if these can be skipped by an easy to calculate sufficient condition for hamiltonian-connectedness. Multiple conditions are presented in Section 2.1, the least restrictive one being the following.

Theorem 4.4. Let G be a graph on n vertices. If G has a complete $(n+1)$-closure, then G is hamiltonian-connected.

It would be possible to use an even easier to compute degree-based sufficiency condition before calculating the closure. For the small graphs, this could only result in a minor improvement, as the calculation of the closure is already quite fast. An obvious algorithm to compute the k-closure takes $O(n^4)$ time, by cycling through a list of nonexistent edges to see whether the degree sum is at least k. A more efficient algorithm calculates the closure of a graph in a time proportional to the output. We have implemented this algorithm, which is presented in [Spi04]. It works as follows.

Let G be a graph of order n and let G_c be a graph that will eventually be the k-closure of G. Let Q be the set of all nonexistent edges (x, y) of G, such that $d(x) + d(y) \geq k$. For each possible vertex degree i (from 0 to $n-1$), a list is maintained containing all vertices of degree i in G_c. The set Q represents the edges that will be added to G_c. While Q is nonempty, the following steps are repeated.

1. Let x, y be a pair of vertices from Q.
2. Let d_x be the degree of x, and let d_y be the degree of y.
3. Remove x, y from Q and add it as an edge to G_c.
4. Move x from list d_x to list $d_x + 1$.
5. If $d_x + 1 \leq k$, then add (x, z) to Q if (x, z) is not an edge of G for each vertex z in list $k - d_x - 1$.

6. Move y from list d_y to list $d_y + 1$.
7. If $d_y + 1 \leq k$, then add (y, z) to Q if (y, z) is not an edge of G for each vertex z in list $k - d_y - 1$.

4.5.9 Order hamiltonicity and toughness

In Algorithm 4.3 the existence of a Hamilton path between u and v is determined before the toughness of the graph is calculated. This order could also be reversed. This would clearly be beneficial if deciding whether $\tau < 2$ is faster than determining the nonexistence of a Hamilton path. A third option would be to place the hamiltonicity check half-way the toughness check, e.g. after checking the toughness using the maximal cut size (successful most of the time) and before switching to smaller cuts.

These options have been tested in combination with all other optimisations, and we have found that it is ideal to start with testing the successful cuts described in Section 4.5.5 and Section 4.5.6, followed by checking the nonexistence of a Hamilton path (also using the reuse from Section 4.5.7), and finally calculating the toughness.

This has been deduced empirically, and it is clearly not ideal for every graph. As mentioned in Section 4.5.6, the input graphs are analysed in a certain order. Generally, the first graphs being generated are rather sparse and the final graphs are quite dense. By using all previous optimisations, the sparse graphs are analysed very quickly due to the low toughness of those graphs and the reuse of cuts. If a graph is very dense, it can be skipped by making use of our preprocessing algorithm from Section 4.5.8. The toughness algorithm requires more time for dense graphs, as these have a higher toughness, and it is harder to show that a toughness is below two. If the toughness is at least two, the algorithm only terminates after all possible cuts. The empirical observation on the ideal order could be explained by the fact that most of the time is spent on relatively dense graphs. For these graphs, the toughness algorithm is very time-consuming.

4.6 Enumeration algorithm

Our final algorithm is thus a variation on Algorithm 4.3, incorporating the optimisations presented in Sections 4.5.5 to 4.5.9. In order to calculate the toughness, it makes use of a variation on Algorithm 4.2, such that the optimisations presented in Sections 4.5.1 to 4.5.4 are integrated. In order to do so, Algorithm 4.2 is modified such that the collection of possible cuts P, is passed as a parameter to the function `get_toughness`. In order to analyse chordal graphs, the algorithm can be modified to test whether the toughness is at least $\frac{7}{4}$. The optimisations that depend on testing whether the toughness is below two also require some modifications. A link to our

implementation can be found in Appendix A, where we shortly explain the structure of our code.

4.6.1 Implementation

Our algorithm has first been implemented in Python. Python is suitable for fast prototyping, both by the nature of the language (resulting in relatively small code size) and the availability of feature-rich graph libraries. These libraries include Networkx [HSS08], Graph-tool [Pei14], and Igraph [CN06]. After comparing these, we have chosen to use the Networkx library, as it had all the features required for implementing the algorithm and can read a graph described by the graph6 format.

For our final algorithm, another programming language has been used to optimise the performance. The following algorithms need to be optimised.

- Counting the number of components.
- Checking the presence of a Hamilton path between two vertices.
- Creating an induced subgraph, such that the algorithm to count the components can be executed on this induced subgraph.

Given these requirements, we have chosen to use the C++ Boost Graph Library [SLL02] (available at [Boo20]) to implement our algorithm. The Boost Graph Library (BGL) is a generic library, which means (among others) that the provided algorithms are data-structure independent. This has the benefit that we can choose a data structure that is optimal for the algorithms being used. Secondly, C++ is known to be among the fastest general-purpose programming languages, so the language is not a restrictive factor.

For both counting the number of components and checking the presence of a Hamilton path by Algorithm 4.1, it is required to iterate over the neighbours of a vertex. The graphs are therefore stored using adjacency lists. The adjacency list in BGL consists of a `VertexList` and an `OutEdgeList`. As BGL is data-structure independent, it is possible to change the data-structure of these two lists depending on which functionality should perform optimal. Possible data structure are lists, vectors, or sets (all from the C++ Standard Template Library). For iterating over the neighbours of a vertex, the data type of `OutEdgeList` is important. This operation is constant time for all types, but there is a significant constant factor time difference between the various types [Boo20]. The speed of this operation in order from fastest to slowest is vector, linked list, and set. For this reason, the `OutEdgeList` is stored as a vector. The type of the `VertexList` does not matter much, as no vertices are added or removed during our algorithm. It is implemented as a vector, to reduce space overhead [Boo20].

Components By profiling our implementation, it can be observed that most of the time is spent on counting the number of components, in order to determine the toughness. The time complexity for counting the number of components is $O(V+E)$. The Boost Graph Library provides an algorithm to count the number of components, based on performing a depth-first search. Recall that the goal is to count the components not containing two vertices u and v. The algorithm in BGL that counts the number of components stores the component of each vertex, such that the components containing u and/or v can be subtracted afterwards. This has been implemented by checking whether u and/or v are in the cut S, and whether u and v are part of the same component.

Hamilton path The Boost Graph Library does not contain an algorithm that can find a Hamilton path. As mentioned in Section 4.3, both a backtracking algorithm and a dynamic programming algorithm have been implemented. The backtracking algorithm (Algorithm 4.1) can be implemented efficiently by storing the graph as an adjacency list. For the dynamic programming algorithm, it is more efficient to store the graph as an adjacency matrix. In order to implement the dynamic programming solution from Section 4.3, we iterate over all subsets in bitwise order. The recursive property $p[S][v]$ is stored in a hashmap (C++ `unordered_map`). The hash function takes a pair of two numbers as key, where the first number is a bitwise representation of the set S and the second number is the vertex v. The backtracking algorithm performed better than the alternative, so this implementation is used in our final algorithm (see Section 5.4).

Induced subgraph The Boost Graph Library supports a so-called filtered graph, which is exactly what is needed to perform algorithms on an induced subgraph. A predicate function determines which vertices and edges should show up in the filtered graph. For an induced subgraph, only the vertices are restricted. Thus a predicate function should return `true` if a vertex is in the filtered graph and `false` if it is not. By using a vector of Booleans containing whether each vertex is in the induced subgraph, the predicate function can apply the filtered graph in constant time. The filtered graph does not create a copy of the original graph, but instead uses a reference to the original graph.

4.6.2 Parallelisation

The code has been run on the DSI computing lab on the University of Twente [Twe20]. The optimisations from Section 4.5.6 and Section 4.5.7 have the disadvantage that they make parallelisation a bit more complex, as one cannot simply distribute all graphs separately as isolated jobs over all cores. This can be solved by distributing chunks of graphs (e.g. jobs consisting of

500.000 graphs). As there is no data exchange between the different chunks, our task can easily be distributed across different machines. This computing cluster makes use of the workload manager Slurm [YJG03].

4.7 Evolutionary algorithm

The approach of iterating over all possible nonisomorphic graphs is only feasible for relatively small graphs, as the amount of graphs increases rapidly, as shown in Table 4.1. It is unlikely that we can improve the algorithm to make it multiple orders of magnitude faster, which is required to analyse all graphs of order 12. An alternative approach for finding tough nonhamiltonian graphs is by analysing random graphs instead. Merely generating graphs randomly and analysing those would be a lot less efficient than iterating over nonisomorphic graphs, due to the chance of generating duplicate or isomorphic graphs. On the other hand, it would make it possible to test larger graphs for which the set of all nonisomorphic graphs could not even be generated. The downside is that it would be hard to draw a meaningful conclusion if the algorithm does not find a sufficiently tough nonhamiltonian graph. As explained in Chapter 3, a good alternative would be to use an evolutionary algorithm instead.

The first choice in the design process of an evolutionary algorithm is in the definition of the fitness function. A natural choice for the fitness function is the graph's toughness, after applying the construction from Section 2.3. The fitness function should also incorporate the existence of a Hamilton path between two vertices u and v. The existence of such a path should result in a low fitness, as it is not a solution to the problem. Note that the vertices u and v are fixed, contrary to the enumeration algorithm. This leads to the fitness function f described by Equation (4.4).

$$f(G) = \begin{cases} \tau(G), & \text{if there exists no Hamilton path from } u \text{ to } v \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

The toughness function τ is the same as in the enumeration algorithm. A drawback of this fitness function is that it is computationally expensive. It has an exponential time complexity, and some optimisations of the toughness algorithm explained in Section 4.5 are not applicable anymore. On the other hand, Section 4.5.6 explains an effective optimisation based on the analysis of similar graphs, that could be even more effective in a mutation-based evolutionary algorithm. This would be the case if every mutated graph is very similar to the previous graph, and our mutations are chosen in such a way that this is the case. This effective calculation of the toughness is part of the motivation for using a mutation-based evolutionary algorithm in the first place.

The second design choice is on the reproductive operators. The crossover operator on two individuals has not yet proven itself in evolutionary algorithms with direct graph encodings, nor in Cartesian genetic programming (see Chapter 3). We have thus chosen not to use crossover, and rely solely on the mutation operator. Multiple mutation operators have been listed in Chapter 3, but we have decided only to use the following two. One of these is selected using a uniform distribution.

- Add a random edge
- Remove a random edge

We have chosen not to include mutations that change the order of the graph. This choice is motivated by three reasons. Firstly, having a constant number of vertices provides multiple benefits regarding implementation efficiency. One major advantage is that it enables the reuse of a cut as described in Section 4.5.6. Secondly, allowing the addition of new vertices could easily lead towards a graph for which the fitness function would be unfeasible to calculate. Finally, applying the evolutionary algorithm on graphs of a fixed order enables us to compare the results to our enumeration algorithm easily. This comparison can be made in terms of the results, as well as in the running time. It would be possible to add other mutation operators, that do not change the order of the graph. There is, however, no clear benefit to do so. An advantage of this small set of mutation operators is that for half of the mutations, there is no need to check whether there exists a Hamilton path in the mutated graph. If there is no Hamilton path from u to v in the original graph, it is clear that this still holds after removing an edge.

Without a crossover operator, there is not much need for a large population and a single-survivor algorithm is a natural choice [APS18]. Single survivor algorithms mostly fall in the evolutionary strategy paradigm. In this paradigm, there is a distinction between the $(1 + \lambda)$ model and the $(1, \lambda)$ model. The former is an overlapping generation model, where the parent competes against its offspring, and the fittest individual survives. The latter is a non-overlapping generation model, where the parent is disregarded, and the fittest offspring survives. We have opted to use the $(1 + \lambda)$ evolutionary strategy algorithm, in order to guarantee fitness monotonicity. This increases the selection pressure, by making the algorithm more exploitative, such that the convergence is more rapid. In case of a tie between the parent and its fittest offspring, we have chosen to select the offspring. Given the discrete fitness landscape of graph toughness in combination with our limited amount of mutation operators, multiple mutations have to be performed in order to increase the toughness of a graph. Note that the fitness landscape consists of all evaluations of a fitness function for all possible solutions. Favouring the offspring is also a common choice when using graphs as indirect encodings,

as this is known to have a positive influence on the performance of Cartesian genetic programming [APS18].

In a well-designed evolutionary algorithm, there should be a balance between exploitation and exploration. Commonly, the selection method is mostly responsible for the exploitation, whilst the primary source of exploration is the variation in reproduction [Jon06]. Based on our choice of a non-overlapping generational model and a single-survivor population with an elitist selection method, we have designed a rather exploitative algorithm. This is partly compensated by the fact that our reproduction method is relatively explorative, due to the lack of a crossover operator. Convergence towards a local optimum is usually the biggest downside of having an exploitative algorithm. Another disadvantage of an exploitative algorithm is that convergence may take very long, as the mutations only explore solutions that are nearby in the fitness landscape. Both of these disadvantages will be analysed and discussed in Section 5.2.

The $(1 + \lambda)$ algorithm is also used by Cartesian genetic programming [MR19], and our overall approach is similar to this research [APS18], which applies an evolutionary algorithm to graph programming.

A disadvantage of using a single-survivor algorithm is the lack of parallel search, strongly increasing the variance between different runs of the algorithm. This does not seem to be a problem, as we can still run the algorithm in parallel. Instead of doing the parallelism within the algorithm, this can give us some more information on the fitness landscape. It enables us to analyse how often the algorithm converges prematurely towards a local optimum, and at what fitness values these local optima exist.

The final choice is in deciding how many offspring to generate, i.e. the value of λ in the $(1 + \lambda)$ model. In the reviewed literature, we found that four is a common choice, and we have therefore decided to implement a $(1 + 4)$-ES model. It is also a standard model in the (graph-based) field of Cartesian Genetic Programming [MR19].

4.7.1 Implementation

The evolutionary algorithm is also implemented in the Boost Graph Library, partly reusing the functionality from the enumeration algorithm. Our algorithm can be described by the pseudocode shown in Algorithm 4.4. It makes use of the following functions.

- `exact_toughness`: this can be implemented using Algorithm 4.2. It is similar to the function used in the enumeration algorithm but without some optimisations. The difference is that in the evolutionary algorithm, it is not sufficient to decide whether the toughness is below two.

- `random_graph` takes a parameter n and returns a random graph of order n. It makes use of the $G(n, p)$ model to study random graphs

Algorithm 4.4 (1+4)-ES model

```
1: function EVOLVE(n, m)
2:     G ← RANDOM_GRAPH(n)                          ▷ generate graph of order n
3:     u ← 0
4:     v ← n − 1
5:     t ← EXACT_TOUGHNESS(G, u, v)
6:     for i ← 1 to m do                                          ▷ m generations
7:         for j ← 1 to 4 do                                       ▷ four offspring
8:             H_j ← MUTATE(G)
9:             if not HAMILTON_PATH(G, u, v) then
10:                t_j ← EXACT_TOUGHNESS(G, u, v)
11:            else
12:                t_j ← 0
13:        if max({t_1, t_2, t_3, t_4}) ≥ t then
14:            G ← H_k (k ∈ {1, 2, 3, 4}) such that t_k = max({t_1, t_2, t_3, t_4})
15:            t ← τ(G)
```

[Gil59]. It creates a graph of order n, where each possible edge is chosen to be included in the graph with probability p. We have chosen to use $p = \frac{1}{2}$, such that all possible spanning subgraph G of K_n are equally probable to be generated. The function should also mark two vertices of G as u and v.

- `Mutate` takes a graph G as a parameter and returns a mutated graph H. This mutated graph is created by either adding an edge or removing an edge. Both mutation operators have an equal probability of being selected. If an edge is added, each disjoint pair of vertices u, v has an equal probability of being selected, and the edge $\{u, v\}$ will be added to the graph. This has been implemented by selecting random pairs of vertices, until a nonadjacent pair has been found. If an edge is removed, each edge $e \in E(G)$ has en equal probability of being removed. This has also been implemented by selecting random pairs of vertices, but this time until an adjacent pair has been found.

- `Hamilton path`: this algorithm is exactly the same as for the enumeration algorithm, and described in Algorithm 4.1.

In order to optimise the implementation of Algorithm 4.4, we have made use of the following ideas.

Termination of the toughness The toughness t_0 of the current individual can be passed as an argument to `exact_toughness`. The function `exact_toughness` can terminate as soon as it finds a toughness lower than t_0, as our algorithm will never select an offspring that has a lower toughness than the parent.

Initial population The function `random_graph` may very well generate a graph G having a fitness value of zero. This could be because the graph is disconnected, or because there exists a Hamilton path between the two labelled vertices u and v. Due to the definition of the fitness function, it may take many generations before the individual has a nonzero fitness. A disadvantage of the fitness function is that it does not provide feedback on whether a mutation is useful in order to reach a positive fitness eventually. Until a positive fitness is reached, each mutation is equally fit as the original graph by having a fitness of zero. This disadvantage can be avoided by generating random graphs until a random graph is generated that has a nonzero fitness. This individual is used as the initial population.

Reuse of a cut S The toughness function `exact_toughness` has to calculate the toughness of very similar graphs. A similar optimisation as described in Section 4.5.6 can be applied to the evolutionary algorithm. Recall that the toughness is calculated by the following formula, which is an approximation of the actual toughness neglecting the small real $\epsilon > 0$.

$$\tau(G) = \min_{S \subseteq V(G)} \left(\frac{1 + 2 \cdot |S|}{2 \cdot \omega'}\right)$$

During the calculation of the toughness of a graph G, one should remember a set $S \subseteq V(G)$ such that

$$\tau(G) = \frac{1 + 2 \cdot |S|}{2 \cdot \omega'}$$

In order to calculate the toughness of a mutation H, this set S can then be tried first. If this yields an upper bound on the toughness of H that is lower than the toughness of G, the algorithm can terminate.

Existence of a Hamilton path Whenever a graph G reproduces itself, it cannot contain a Hamilton path between the labelled vertices u and v (by the aforementioned optimisation on the initial population). If a mutated graph H is created by removing an edge e from G, it is clear that there cannot be a Hamilton path from u to v in the mutated graph H. The existence of a

Hamilton path should thus only be checked if an edge is added, not if one is deleted. Note that if such a Hamilton path exists in H, it must contain this newly added edge. This latter observation has not been used to improve the implementation, due to the relatively small improvement it could provide.

Biased algorithm Finally, we present a possible improvement that is not included in our final algorithm, but it will be discussed in the results (in Section 5.2). It is possible to determine the next possible fitness value that is required to obtain a strictly higher fitness than the current individual has. For a graph G with two vertices u and v in G, the set of possible toughness values is restricted by the possible number of components not containing either u or v. Instead of calculating all these possible fractions, we analysed successful runs of our algorithm for graphs of order 8, and looked which fitness values those runs obtained during the run. It is common for our algorithm to go from $\tau \approx \frac{3}{2}$, towards $\tau \approx \frac{7}{4}$, towards $\tau \approx \frac{11}{6}$, and finally towards $\tau \approx \frac{9}{4}$. We can apply the optimisations of the enumeration algorithm presented in Section 4.5.1 and Section 4.5.2, using the next toughness value in this sequence as the required minimal toughness value that offspring should satisfy. A consequence of these optimisations is that the algorithm does not compute the actual toughness anymore. Even though fitness monotonicity is guaranteed by our (1+4)-ES model, it is not guaranteed that the toughness actually increases.

The second disadvantage of this method is that it introduces a bias into our algorithm to favour certain graphs. The mutations that lead to graphs satisfying this predefined sequence of fitness values are favoured. This biased algorithm is very successful in finding the global optimum for graphs of order 8; hence it is included in our report. This can be explained by the fact that there is no toughness monotonicity, as the exact toughness is not calculated. It is thus possible to escape a local optimum in the fitness landscape. We do think that this bias is undesired if the goal is to find nonhamiltonian graphs with toughness $\geq \frac{9}{4}$, and that our obtained results are stronger, if this bias is not present in the algorithm. For larger graphs, the optimisation is also not as efficient due to the possible decrease in toughness.

4.8 Verification

Whilst we cannot be sure that our implementation is correct, some effort has been made to verify it. Firstly, the graph used to construct a nonhamiltonian graph having toughness $\frac{9}{4} - \epsilon$ (Figure 2.4a from Section 2.3) is an expected result when running the code on graphs of order 8. Secondly, the chordal graph leading to a nonhamiltonian chordal graph having toughness $\frac{7}{4} - \epsilon$ (Figure 2.4b) is an expected result when running the code on chordal graphs of order 6. Thirdly, Algorithm 4.3 has been implemented both in Python using the Networkx library and in C++ using the Boost Graph Library. This makes it possible to compare the results and check if they coincide. Fourthly, checking the existence of Hamilton paths is implemented by two separate algorithms in C++. The results of these implementations have been compared. Finally, it is possible to compare the results from our enumeration algorithm to those of our evolutionary algorithm. These results from both our algorithms, presented in Chapter 5, strengthen our belief that the implementation is correct. Note that if a result has been found by one of our algorithms, both the toughness and hamiltonicity can be proven by hand.

Chapter 5

Results

In Section 5.1 and Section 5.2, we present the results of respectively our enumeration algorithm and our evolutionary algorithm. In Section 5.3, we shortly discuss the efficiency of the preprocessing optimisation presented in Section 4.5.8. Section 5.4 contains a comparison between the two different implementations of the Hamilton path algorithm.

Note that some results presented here are not the direct output of our C++ code, as they have been processed afterwards. In order to visualise the output of our algorithms, there are multiple options. Our algorithms return the solutions in the graph6 format. An online tool to visualise this format can be found at this website [TOT20]. Alternatively, both the Boost Graph Library in C++ and the Networkx framework in Python provide the functionality to export a graph to the `dot` format.

5.1 Enumeration algorithm

All 1,006,700,565 nonisomorphic connected graphs up to order 11 have been analysed by our enumeration algorithm. Recall that our algorithm only checks whether the toughness is above two. In order to provide more information, the output of the algorithm is analysed by a Python script. This script calculates the exact toughness and provides a cut S that leads to this toughness. An overview of the output of our enumeration algorithm can be seen in Table 5.1. It shows the number of nonisomorphic graphs H that can be used to construct a nonhamiltonian graph G having toughness $\tau(G) > 2$. Unfortunately, the enumeration algorithm did not provide a tougher nonhamiltonian graph than the one already known. Neither did it find a graph that could provide a smaller nonhamiltonian 2-tough graph, than the one already known on 42 vertices. Given that our implementation is correct, it does show that the graph shown in Figure 2.4a is the smallest graph H leading to a nonhamiltonian graph G with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, by using the construction described in Section 2.3.

order n of H	$\tau(G) \approx \frac{9}{4}$	$\tau(G) \approx \frac{13}{6}$	runtime
< 7	0	0	0.1 s
8	2	0	0.7 s
9	2	2	45s
10	6	16	2h 32m
11	18	158	unknown[1]

Table 5.1: Results of the enumeration algorithm: the number of nonisomorphic graphs H that can be used to construct a nonhamiltonian graph G having toughness $\tau(G) > 2$.

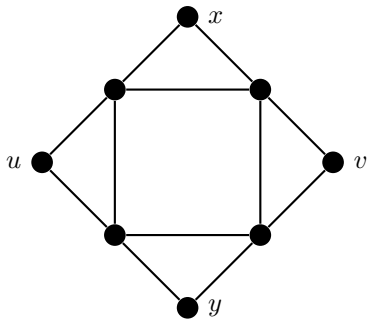
Secondly, it shows that finding a small input graph H of order $n \leq 11$ that can be used to construct a nonhamiltonian graph with toughness at least $\frac{9}{4}$ is not possible, and that the research to find such a graph by hand does not have to be continued. The above experimental results might indicate that the following conjecture is true.

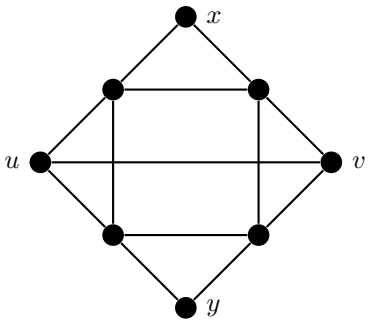
Conjecture 5.1. Every 2-tough graph of order $n \leq 41$ is hamiltonian.

Note that the algorithm can return the same graph multiple times, if there are multiple pairs of vertices u and v for which the construction from Section 2.3 works. Due to the symmetry of the graph presented in Figure 2.4a, there are two possible pairs of vertices to which the construction can be applied. These duplicate graphs are counted only once in Table 5.1. The runtime is an estimate, that is obtained by running the algorithm on Linux with an Intel Core i7-8750H processor. It may be feasible to check all nonisomorphic graphs of order 12 (roughly 164 billion) given enough time and resources. Note that the set of nonisomorphic input graphs, which can be generated using Geng, would require around 2 TB of space. Motivated by the lack of solutions returned by the algorithm for $n \leq 11$, we have chosen to spend our time and resources in developing and improving the evolutionary algorithm instead.

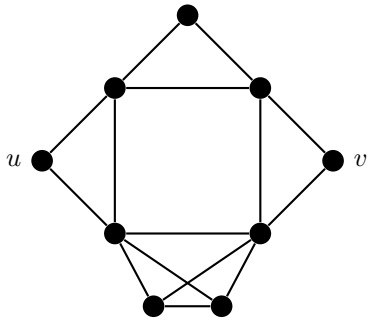
The other graphs that have been found by the enumeration algorithm that can be used to construct a nonhamiltonian graph having $\tau \approx \frac{9}{4}$ are all variations of the two known graphs shown in Figure 5.1a. These two graphs can be modified by replacing either one or both of the vertices x and y by a complete graph; all vertices of which have the same set of additional neighbours $N(x)$ or $N(y)$, respectively. This is illustrated by the graphs shown on the left in Figure 5.1b and Figure 5.1c.

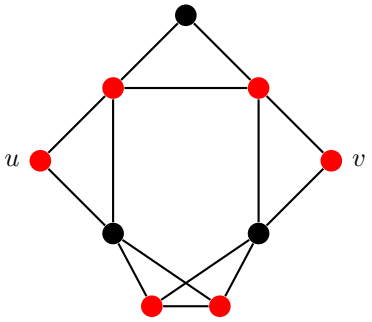
[1]This has not been run with the final version of our algorithm, and it has been run in parallel on a different machine.



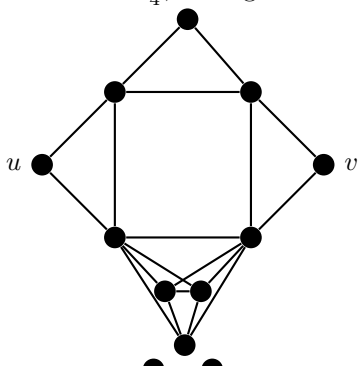


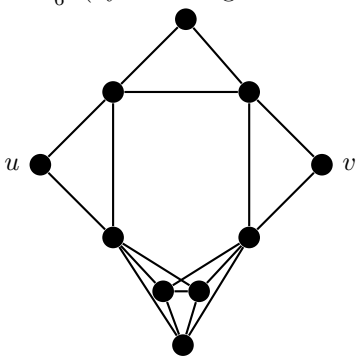
(a) The only two graphs of order 8 returned by the enumeration algorithm.

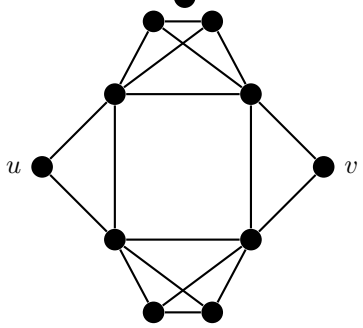




(b) Two graphs of order 9 returned by the enumeration algorithm. The left one leads to $\tau \approx \frac{9}{4}$, the right one leads to $\tau \approx \frac{13}{6}$ (by removing the red vertices).







(c) Three out of the six graphs of order 10 leading to $\tau \approx \frac{9}{4}$.

Figure 5.1: The output of our enumeration algorithm.

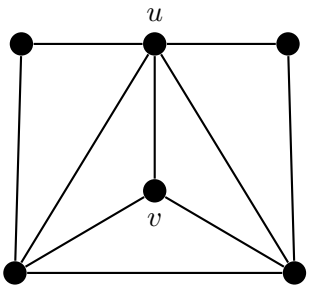


Figure 5.2: Chordal graph leading to $\tau \approx \frac{7}{4}$.

Some of the edges in H could be removed, depending on the order of the complete graphs replacing x or y. This results in many variants on both graphs shown in Figure 5.1a, that can be used to construct a nonhamiltonian graph having $\tau \approx \frac{9}{4}$. By removing more edges, it is possible to construct a nonhamiltonian graph having $\tau \approx \frac{13}{6}$. Half of the output for graphs of order 9 is shown in Figure 5.1b; the other half consists of similar graphs, where u and v are joined by an edge. In the graph on the right in Figure 5.1b, it can be seen that the removal of the edge between u and v results in $\tau \approx \frac{13}{6}$, as the graph can be split into three vertices not containing u or v, by removing the six red vertices. For the graphs of order 10 and 11, only those resulting in $\tau \approx \frac{9}{4}$ are presented here. Again, only those where u and v are not joined by an edge are shown. The graphs of order 10 are shown in Figure 5.1c and those of order 11 are appended in Appendix B.

5.1.1 Chordal graphs

By analysing chordal graphs, we found that there is no chordal graph H of order $n \leq 13$, such that the construction described in Section 2.3 results in a nonhamiltonian chordal graph G with toughness at least $\frac{7}{4}$. Secondly, the graph shown in Figure 2.4b is the smallest chordal graph H leading to a nonhamiltonian chordal graph G with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, by using the construction described in Section 2.3. We did find the expected graph of order 6 that is similar to the graph shown in Figure 2.4b, obtained by joining the two vertices u and v such that there does not exist a Hamilton path from u to v.

The output of our code applied to chordal graphs is shown in Table 5.2. It shows the number of nonisomorphic graphs that can be used to construct a nonhamiltonian graph with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$ for graphs up to order $n = 13$. There exist infinitely many chordal graphs that can be used to construct a nonhamiltonian chordal graph G with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, by using the aforementioned method for general graphs that replaces a single vertex by a complete subgraph.

order n of H	< 5	6	7	8	9	10	11	12	13
$\tau(G) \approx \frac{7}{4}$	0	2	7	25	118	597	671	32,035	68,350

Table 5.2: Results of the enumeration algorithm applied to chordal graphs.

Given that there exists a real number t_0 such that every t_0-tough chordal graph is hamiltonian (see Section 2.2), our research might indicate that Conjecture 5.2 is true.

Conjecture 5.2. Every $\frac{7}{4}$-tough chordal graph is hamiltonian.

5.2 Evolutionary algorithm

Section 5.2.1 contains an overview of our experiments. In Section 5.2.2, we discuss the convergence of our algorithm. In Section 5.2.3, we show that it is possible for our algorithm to get stuck in a local optimum. Finally, in Section 5.2.4, we shortly discuss the modified algorithm that has a bias to avoid these local optima.

For the sake of illustration, we have included a visualisation of an example run of our evolutionary algorithm in Appendix C. Note that this run is rather successful by converging to the global optimum in only 61 iterations. From these 61 iterations, we have removed those iterations where the parent is selected as the survivor for the next generation. The remaining 36 iterations are shown, where the addition of an edge is shown in green, and the removal of an edge is shown in red as a dotted line.

5.2.1 Experiments

If a graph H of order n exists that will lead to a nonhamiltonian graph of toughness $\tau \geq \frac{9}{4}$, there will also be a graph H' of order $m > n$ that has this property. If there is a toughness determining cut set S of H that splits off a single vertex as a component of $H - S$, this graph H' can be constructed by replacing this single vertex $v \in H$ by a complete graph K_l as described in Section 5.1. If there is not a single vertex that can be split off, a more general construction can be used. Consider the smallest component C of $H - S$, then H' can be constructed by adding a complete graph K, and all edges between each vertex of K and each vertex of C as well as all the neighbours of C in S.

This has the useful consequence that we do not have to run our algorithm for graphs of every order, but that we can run it for graphs that are so large that it is only just feasible to terminate the algorithm. The output of some sample runs of our evolutionary algorithm is shown in Table 5.3. We included the runs of order 8 and 11 in this report, to compare the results

n		$\tau \approx \frac{3}{2}$	$\tau \approx \frac{7}{4}$	$\tau \approx \frac{13}{8}$	$\tau \approx \frac{11}{6}$	$\tau \approx \frac{13}{6}$	$\tau \approx \frac{9}{4}$
8	#graphs	0	10,322	0	0	0	8349
	nonisomorphic	0	86	0	0	0	2
11	#graphs	0	903	2	1	39	666
	nonisomorphic	0	894	2	1	12	17
16	#graphs	24	46	0	0	1	13
	nonisomorphic	24	46	0	0	1	13

Table 5.3: Results of the evolutionary algorithm.

to those of the enumeration algorithm. Graphs of order 16 are close to the limit of what our evolutionary algorithm can handle. A second motivation to select these graphs, is that 16 is a power of 2. This might enable the graph to have nice symmetry, inspired by the existent solution of order 8. The algorithm has also run for graphs of respectively order 20 and 24, but these runs have not terminated. They both ran for over seven days, on all cores of respectively the `m610` and `r930` cluster of the DSI computing lab on the University of Twente [Twe20].

Note that the sample runs presented here do not aim to provide an extensive analysis of the performance of our algorithm. As an important motivation for this research has been to find a nonhamiltonian graph with toughness at least $\frac{9}{4}$, our resources have mostly been spent in trying to achieve this goal instead. For example, by trying to run the algorithm for (unfeasibly) large graphs for an extended amount of time. The results in Table 5.3 have been obtained by running the algorithm on the `r930` partition of the DSI computing lab on the University of Twente [Twe20], where the algorithm can run 192 instances in parallel. It is named *caserta* on the web page that is cited above; this page contains an overview of the hardware. Other executions have mostly run on the older `m610` partition, that has 160 cores. The algorithm ran 15:47:27 hours for order 8, 18:27:49 hours for order 11, and 72:45:48 hours for order 16. Note that the algorithm found 17 out of the possible 18 graphs of order 11 that have been found by our enumeration algorithm.

All these results have been obtained by running the evolutionary algorithm for 100,000 iterations. For graphs of order $n \leq 11$, the algorithm can terminate once $\tau \approx \frac{9}{4}$, as a result of our enumeration algorithm. This is not desirable for larger graphs, as the algorithm should be able to return graphs having a toughness above this bound. For graphs of order $n \leq 11$, a consequence of this termination is that graphs with a higher toughness are returned faster than the other graphs. To a lesser extent, this is also true for larger graphs. For the graphs of order 16 shown in Table 5.3, the first two graphs returned by the algorithm were returned after roughly 6 hours. These two graphs terminate with $\tau \approx \frac{9}{4}$, and this toughness is obtained after

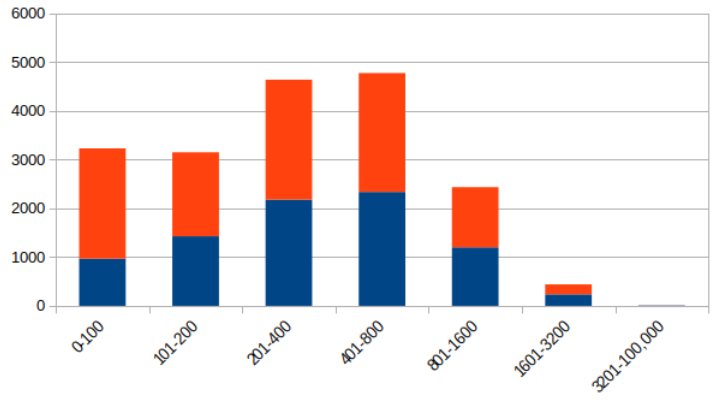
respectively 6940 and 5457 iterations.

For the remaining iterations, the algorithm loops over the same set of possible mutations, which happens relatively fast. It is hard to determine when to terminate an evolutionary algorithm, as it may be possible that either a local or a global optimum has been reached early on, but this is hard to check. An approach to improve the algorithm by detecting local optima is given in the discussion. Note that for small graphs, the algorithm performs a lot better by performing less than 100,000 iterations, as will be explained in the following section.

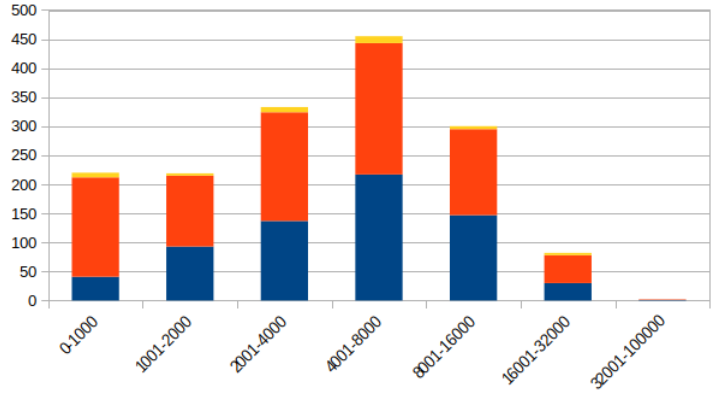
5.2.2 The convergence of the algorithm

Our algorithm can return the iteration where the final toughness value has been reached. For the runs described in Table 5.3, we have analysed these iterations. The results are shown in Figure 5.3. The runs are categorised by the final toughness value, each represented by a different colour. Note that the horizontal axis differs between the three figures.

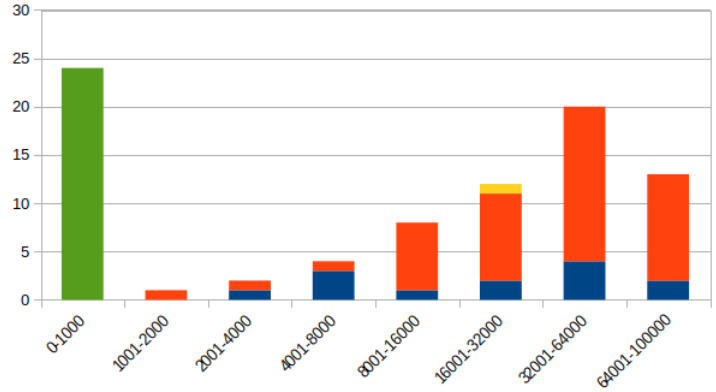
It can be seen that the algorithm nearly always terminates within 3200 runs for graph of order 8 (Figure 5.3a). It is highly probable that for all of these runs, the algorithm reached a local optimum. An example of such a local optimum is shown in Section 5.2.3. The algorithm would clearly be more efficient for order 8 if it is modified to make fewer than 100,000 iterations. A similar result is obtained for graphs of order 11 in Figure 5.3b, where the number of runs of each bar is increased by a factor ten. Here it would also be more efficient to terminate the algorithm earlier. For graphs of order 16, it can be seen in Figure 5.3c that a significant fraction of the runs never obtains a toughness higher than $\frac{3}{2}$. This is probably a consequence of the huge number of possible mutations, and not because the algorithm is stuck in a local optimum. For many cases, the initial graph has $\tau \approx \frac{3}{2}$ and no improvement has been made. The data shown in Figure 5.3c is probably not of sufficient high quantity to make more observations on the fitness landscape.



(a) Graphs of order 8.



(b) Graphs of order 11.



(c) Graphs of order 16.

Figure 5.3: Convergence of the evolutionary algorithm. The horizontal axis shows the number of iterations required to converge. The vertical axis shows the number of runs. Blue represents the number of runs terminating with $\tau \approx \frac{9}{4}$, red represents the runs terminating with $\tau \approx \frac{7}{4}$, green represents the runs terminating with $\tau \approx \frac{3}{2}$, and yellow represents the remaining runs.

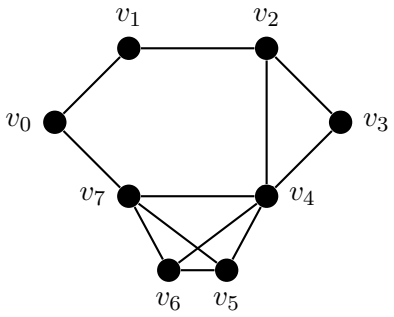


Figure 5.4: A local optimum in the fitness landscape.

5.2.3 Local optima

Let G be the nonhamiltonian graph constructed by the graph H illustrated in Figure 5.4. The graph H is a local optimum in the fitness landscape and $\tau(G) \approx \frac{7}{4}$. In order to reach a global optimum, it would be required to first remove an edge $e \in E(H)$, such that the resulting graph G' has $\tau(G') \approx \frac{3}{2}$. This is impossible, as our evolutionary algorithm guarantees fitness monotonicity. Note that the vertices usually labelled as u and v, such that no Hamilton path between u and v exists, are now labelled as v_0 and v_3 in Figure 5.4. In the following paragraphs, we demonstrate that this is indeed a local optimum.

The first observation is that the only edges that can still be added to H by our algorithm are $\{v_0, v_2\}, \{v_0, v_3\}, \{v_0, v_4\}, \{v_0, v_5\}$, and $\{v_0, v_6\}$. By careful analysis, it can be seen that the addition of any other edge would lead to the existence of a Hamilton path from v_0 to v_3.

The second observation is that the removal of the edges listed below would lead to a toughness $\tau < \frac{7}{4}$. We also list the cut $S \subseteq V(H)$ that can be used to obtain this toughness. In the below text ω' refers to the number of components in $H - S$ not containing v_0 or v_3.

- For $H - \{v_0, v_1\}$, the cut $S = \{v_2\}$ leads to $\omega' = 1$.
- For $H - \{v_1, v_2\}$, the cut $S = \{v_0\}$ leads to $\omega' = 1$.
- For $H - \{v_0, v_7\}$, the cut $S = \{v_4\}$ leads to $\omega' = 1$.
- For $H - \{v_2, v_3\}$, the cut $S = \{v_0, v_4\}$ leads to $\omega' = 2$.
- For $H - \{v_3, v_4\}$, the cut $S = \{v_0, v_2\}$ leads to $\omega' = 2$.
- For $H - \{v_2, v_4\}$, the cut $S = \{v_0, v_3\}$ leads to $\omega' = 2$.
- For $H - \{v_5, v_6\}$, the cut $S = \{v_0, v_2, v_4, v_7\}$ leads to $\omega' = 3$.
- For $H - \{v_5, v_7\}$, the cut $S = \{v_0, v_2, v_4, v_6\}$ leads to $\omega' = 3$.
- For $H - \{v_6, v_7\}$, the cut $S = \{v_0, v_2, v_4, v_5\}$ leads to $\omega' = 3$.

Only one edge can be removed by our evolutionary algorithm, and it must be in the set of edges: $\{\{v_{4,5}\}, \{v_{4,6}\}, \{v_{4,7}\}\}$. If a second edge is removed, the toughness would be below $\frac{7}{4}$: for $H - \{\{v_4, v_5\}, \{v_4, v_6\}\}$, the cut $S = \{v_0, v_2, v_3, v_7\}$ leads to $\omega' = 3$. The choice for these two edges can be made without loss of generality. The third observation is that by the removal of either of these three edges, there are still no more new edges that can be added without introducing a Hamilton path from v_0 to v_3.

If v_0 is joined to every other vertex, all the above results still hold, except that it would be possible to remove the edge $\{v_0, v_7\}$. In order to remove $\{v_0, v_7\}$, v_0 should be joined to either v_5 or v_6. Both graphs obtained by replacing the edge $\{v_0, v_7\}$ in H, by either $\{v_0, v_5\}$ or $\{v_0, v_6\}$ are isomorphic to H, thus for these graphs all above observations would still hold.

The algorithm will therefore continue with adding and removing edges from $\{\{v_0, v_2\}, \{v_0, v_3\}, \{v_0, v_4\}, \{v_0, v_5\}, \{v_0, v_6\}, \{v_0, v_7\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_7\}\}$, without removing two edges from $\{\{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_7\}\}$ and without removing all edges from $\{\{v_0, v_5\}, \{v_0, v_6\}, \{v_0, v_7\}\}$.

5.2.4 Biased algorithm

As explained in Section 4.7.1, we have also developed a modification based on some observations on graphs of order 8. This modification introduces a bias to mutate towards certain graphs. For graphs of order 8, this modification makes the algorithm a lot more efficient. For large graphs, however, this bias is undesired. The motivation for developing the algorithm is to construct nonhamiltonian graphs with toughness at least $\frac{9}{4}$. If the algorithm has a bias towards the already known graphs of $\tau \approx \frac{9}{4}$, it may always converge to these optima and never find a higher global optimum if it would exist. We have therefore decided not to include this modification in our final algorithm, but to illustrate the improvement, we show the results for graphs of order 8 in Table 5.4. These results are produced in only 21 minutes, using the same hardware as in the experiments from Table 5.3.

This improvement in speed can at least partly be explained by the fact that our algorithm terminates only after 100,000 iterations, or when $\tau \approx \frac{9}{4}$. If it can always converge towards this global optimum, it will be a lot faster. This convergence towards the global optimum has been achieved by the fact that the modified algorithm can escape local optima, as explained in Section 4.7.1. In order to achieve similar improvements, without introducing

n=8	$\tau \approx \frac{7}{4}$	$\tau \approx \frac{9}{4}$
total graphs	3	39,211
nonisomorphic graphs	3	2

Table 5.4: Results of the evolutionary algorithm with bias.

bias and by still guaranteeing toughness monotonicity, we think it would be better to develop a method to detect local optima such that the algorithm can terminate. Suggestions to do so will be presented in the discussion.

5.3 Complete closures

The efficiency of the preprocessing optimisation presented in Section 4.5.8, has also been analysed by counting the number of graphs that have a complete $(n+1)$-closure. The results are shown in Table 5.5. The algorithm takes around 150 minutes to analyse all nonisomorphic graphs of order 11, on Linux with an Intel Core i7-8750H processor. For these graphs, it can exclude over 43 per cent of the graphs. Recall that our enumeration algorithm spends more time on relatively dense graphs, thus the improvement is (a lot) more than 43 per cent of the time. To illustrate how these number compare to the sufficiency for being hamiltonian (Corollary 2.10), the number of graphs having a complete n-closure are shown in Table 5.6.

order n	complete $(n+1)$-closure	incomplete $(n+1)$-closure
5	3	18
6	11	101
7	104	749
8	1706	9411
9	58,373	202,707
10	3,778,688	7,937,883
11	437,315,425	569,385,140

Table 5.5: The number of connected nonisomorphic graphs having respectively a complete, and an incomplete $(n+1)$-closure.

order n	complete (n)-closure	incomplete (n)-closure
5	7	14
6	45	67
7	352	501
8	5540	5577
9	157,016	104,064
10	8,298,805	3,417,766
11	802,944,311	203,756,254

Table 5.6: The number of connected nonisomorphic graphs having respectively a complete, and an incomplete (n)-closure.

Order n	Number of pairs	Backtracking	Dynamic programming
7	9370	0.07s	0.28s
8	136,275	1.4s	10s
9	3,039,927	3m26	8m52

Table 5.7: A timing analysis of two implementations of the Hamiltonian path algorithm. The algorithms return the number of pairs of distinct vertices u and v, such that there does not exist a path from u to v.

5.4 Hamilton path algorithm

A comparison of the efficiency between the two different Hamilton path algorithms presented in Section 4.3 can be found in Table 5.7. The runtime has been obtained using the same hardware as in the previous section. It has been tested on all nonisomorphic graphs of order 7, 8, and 9. For each graph G, the existence of a Hamilton path is checked between every two distinct vertices u and v in G. The algorithm counts all these different pairs. In our enumeration algorithm, this is not the case due to the optimisations presented in Section 4.5. In our enumeration algorithm the reuse of solutions would be less applicable than it would be in the comparison of Table 5.7, favouring the backtracking algorithm even more. It has also been tested on a subset of the nonisomorphic graphs of order 11, for which the backtracking algorithm was still faster. Given the time complexity of both algorithms, it is to be expected that the dynamic programming algorithm is faster on larger graphs, but we conclude that it is not useful in our approach. Note that the implementation of the dynamic programming solution makes use of an adjacency matrix, as this is more efficient to test the existence of a specific edge.

Chapter 6

Conclusions and discussion

We have designed and implemented both an enumeration algorithm and an evolutionary algorithm, with the aim to find suitable graphs to construct nonhamiltonian graphs with a high toughness. Both algorithms have been verified by being able to return the known class of nonhamiltonian graphs with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$. The enumeration algorithm has also been verified by being able to return the known class of nonhamiltonian chordal graphs with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$. Our belief that the implementation is correct has been motivated in Section 4.8. The main argument is that we have two completely different algorithms, of which one has two different implementations; all three return similar results. These results coincide with the two aforementioned classes of graphs.

The motivation for this project has been threefold. Firstly, to find a nonhamiltonian graph with toughness at least $\frac{9}{4}$. Secondly, to find a nonhamiltonian chordal graph with toughness at least $\frac{7}{4}$. The final motivation is to find a nonhamiltonian 2-tough graph on less than 42 vertices. Unfortunately, we have not found a graph satisfying any of these criteria. Given that our implementation is correct and contrary to our earlier beliefs, we can conclude the following results.

- There is no graph H of order $n \leq 11$ such that the construction described in Section 2.3 results in a nonhamiltonian graph G with toughness $\tau \geq \frac{9}{4}$.
- There is no chordal graph H of order $n \leq 13$ such that the construction described in Section 2.3 results in a nonhamiltonian chordal graph G with toughness $\tau \geq \frac{7}{4}$.
- The graph found by Bauer, Broersma and Veldman [BBV00], as shown in Figure 2.4a is the smallest graph H leading to a 2-tough nonhamiltonian graph G, by using the construction described in Section 2.3. The only other graph H' of order 8 satisfying this property can be obtained by joining u and v by an edge.

- Similarly, the graph shown in Figure 2.4b is the smallest chordal graph H leading to a nonhamiltonian chordal graph G with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, by using the construction described in Section 2.3. The only other graph of order 6 satisfying this property can be obtained by joining u and v by an edge.

Based on the results of our evolutionary algorithm, we conjecture that there is no graph H of order $n \leq 16$ such that the aforementioned construction results in a nonhamiltonian graph G with toughness $\tau \geq \frac{9}{4}$.

We have also shown that there exist infinitely many graphs that can be used to construct a nonhamiltonian graph G with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$. Similarly, there exist infinitely many chordal graphs that can be used to construct a nonhamiltonian chordal graph G with toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$.

6.1 Enumeration algorithm

Our current enumeration algorithm contains many optimisations that make the algorithm multiple orders of magnitude faster than a more straightforward implementation. It is reasonable to believe that the algorithm can still be improved. Given sufficient resources, it may already be possible to analyse graphs of order twelve. For larger graphs, the rapid increase in the number of nonisomorphic graphs would make this approach unfeasible. Instead of improving the algorithm, it may thus be more useful to reduce the input set.

This could be done by incorporating the preprocessing algorithm from Section 4.5.8 into the non-isomorphic graph generation tool Geng. In order to generate all graphs, Geng traverses through a forest of objects. Given this preprocessing algorithm, it may be possible to prune whilst generating the graphs and skip certain branches of graphs in this forest completely. This can be combined with an easier to compute degree-based sufficient condition for hamiltonian-connectedness. This approach would reduce the space required to store all graphs, as well as save time by writing and reading fewer graphs. If it is indeed possible to skip several branches of graphs in the generation process, it will save significantly more time.

Even though our input consists of nonisomorphic graphs, it is possible that our algorithm performs duplicate calculations. Our algorithm loops over all pairs of vertices; thus, if a graph contains a nontrivial automorphism, the same calculation is performed twice. Note that an automorphism is nontrivial if it is not the identity mapping. These automorphisms are not easy to obtain, but perhaps this could be done more efficiently during the generation of the nonisomorphic graphs. In order to generate these graphs, each graph is described by a canonical form such that isomorphic graphs have the same canonical form. This canonical form may also be useful to determine automorphisms. The problem of duplicate calculations could

be avoided, by generating only those graphs that are nonisomorphic with labelled vertices u and v. This has the disadvantage that more graphs will be created.

In order to reduce this set of generated graphs, it is possible to impose the restriction that only those graphs are generated where there does not exist a Hamilton path from u to v. This could probably be done similarly as in [GMZ20], which is explained in more detail in the thesis [Mee18]. Their research explains the design and implementation of a tool to generate k-hamiltonian graphs (graphs with exactly k Hamilton cycles), inspired by the isomorphism checking algorithm used in Nauty (from Section 4.1) that is based on the canonical construction path method by McKay [McK98].

We expect that at least some of the suggestions mentioned above would improve our algorithm. However, it is not sure whether it will provide enough improvement to analyse larger graphs than our current enumeration algorithm can handle. As we believe that an evolutionary algorithm could be more efficient, the above suggestions have not been researched.

6.2 Evolutionary algorithm

We have found that the field of evolutionary algorithms is well-suited to be applied to our problem. The evolutionary algorithm enabled us to analyse larger graphs than an enumeration algorithm could ever handle on existing hardware. It could analyse graphs of order 16, for which there are over 63×10^{21} different nonisomorphic connected graphs. For the generated graphs up to order 16, all returned graphs that can lead to a constructed nonhamiltonian graph with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$ are variants of the known graph of order 8 (Figure 2.4a). As our algorithm has been designed to have as little bias as possible, we conjecture that this toughness value is the best possible result for graphs up to order 16.

We have made multiple design choices that influence our obtained results. Firstly, we have chosen to use a direct graph encoding. This enables us to perform meaningful mutation operations, which greatly improves the performance based on the possible optimisations presented in Algorithm 4.4. Secondly, we have defined a fitness function f. A disadvantage of our fitness function is that it returns a value of zero for many graphs. Therefore, it does not provide feedback on whether a mutation is useful in order to eventually reach a positive fitness value. This has been resolved by generating graphs until a graph has a positive fitness value. An alternative would be to modify the fitness function for disconnected graphs and for graphs containing a Hamilton path between the two labelled vertices u and v. A third design choice has been made by choosing only two mutation-based reproductive operators. Alternatively, one could include operators that enable the addition and removal of vertices. In our work, these are not included, both to improve

efficiency and to avoid a fitness function that is unfeasible to calculate. These two extra mutation operators could be added to our model, by limiting the number of vertices a graph can have. This modification could enable the bottom-up building of larger graphs. A second alternative to the set of reproductive operators would be to include a crossover operation. The operators presented in Chapter 3 do not seem suitable, but a crossover operator might prove itself in the future. One option to devise a meaningful crossover operation is to utilise historical markings, where the historical origin of each gene is marked by an innovation number [SM02]. This could enable meaningful crossover without expensive topological analysis. It might be beneficial to combine this with the bottom-up building of larger graphs, by modifying the reproductive operators.

We have used a (1+4)-ES model. Our original motivation for doing so was to guarantee fitness monotonicity, but it also enabled us to apply some optimisations in the toughness calculation. If it is to be replaced by a non-overlapping generational $(1, 4)$-ES model, the exact toughness values of all offspring have to be calculated. This is not the case for the (1+4)-ES model, as the toughness calculation of an individual can terminate when it is lower than the toughness of its parent. One restrictive property of our algorithm is that it can get stuck in a local optimum, as a consequence of the fitness monotonicity. This has been illustrated by the local optimum having toughness $\frac{7}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$ presented in Section 5.2.3. The (1,4)-ES model does have the advantage that it cannot get stuck in a local optimum, due to the lack of fitness monotonicity. We do believe, however, that the $(1 + 4)$-ES model has been the right choice, as fitness monotonicity is very useful for convergence in larger graphs.

Finally, it could be useful to analyse the $(1 + \lambda)$ model for different values of λ. Clearly, this single survivor model should also be modified if the set of reproductive operations would be changed to include crossover.

6.3 Future work

In the previous two sections, we have discussed possible modifications to our approach. This already leads to possibilities for future research. In this section, we will discuss what directions for future research seem most promising.

As explained in Section 6.1, is it possible to improve the enumeration approach by incorporating the preprocessing algorithm into Geng. If it is possible to skip branches of graphs during the generation of nonisomorphic graphs, this could provide a significant time improvement. Alternatively, it may be possible to obtain improvements by developing a tool that generates all nonisomorphic graphs with labelled vertices u and v, such that there does not exist a Hamilton path from u to v. We expect, however, that it would

not provide many new results, as there is still a very rapid increase in the number of nonisomorphic graphs of order n (see Table 4.1).

We believe that an improved version of the enumeration algorithm would not be able to analyse graphs of the same order as the evolutionary algorithm. It could, however, give more certainty than the evolutionary algorithm, as there is no stochastic method involved. Nevertheless, we expect that research on the evolutionary algorithm is most promising. If the goal is to find all graphs of order n in the class of nonhamiltonian graphs with toughness $\frac{9}{4} - \epsilon$ for arbitrarily small $\epsilon > 0$, the evolutionary algorithm could easily be modified to do so. Instead of returning one graph having toughness $\frac{9}{4} - \epsilon$, it could keep mutating and return similar graphs.

One option for future research is to extend the evolutionary algorithm for chordal graphs. This could be done by implementing an algorithm to check whether a graph is chordal, such as the linear time algorithm presented in [TY84]. Alternatively, for chordal graphs it may be possible to build a solution bottom-up, by changing the set of reproductive operators, and using the elimination scheme of chordal graphs explained in Chapter 2.

Another possibility for future research would be to improve the evolutionary algorithm. This could be done by revising some design choices discussed in Section 6.2. By developing a method to benchmark different algorithms, it would be possible to compare these different design choices.

Another improvement to our algorithm can be made by detecting local optima. This would enable the algorithm to terminate earlier, without the risk of terminating too soon. We have considered multiple options to detect a local optimum, but could not yet decide the ideal implementation. Firstly, one could compare whether an individual is isomorphic to a graph of which it has been proven that it is a local optimum, such as the one presented in Section 5.2.3. This has the disadvantage that there may be many local optima that need to be compared. Secondly, one may be able to devise an algorithm that can detect whether a graph is a local optimum based on the possible set of mutations. This could be implemented by doing a similar analysis, as is shown in Section 5.2.3. Either of these options could then be executed after, for example, every thousand iterations of the algorithm. Finally, it is possible to analyse the recent mutations. If a clear pattern of repetition appears, it may be assumed that an optimum has been reached. This has the disadvantage of providing false positives, where a run is terminated whilst it is not actually in a local optimum.

Alternatively, the evolutionary algorithm could be enhanced by improving the current implementation of certain functions. Some improvements can be made to our Hamilton path algorithm. For example, by using the observation that a Hamilton path should include the edge that is added by the previous mutation, as explained in Section 4.7. The algorithm may also be improved by incorporating certain heuristics to modify the backtracking, as explained in Section 4.3. Alternatively, one could devise an algorithm based on the

dynamic programming solution for finding Hamilton paths (from Section 4.3), that modifies only those sets that could have been changed by the addition or removal of a specific edge. As the same edges are added and removed quite often, it may then be fruitful to cache the results of this dynamic programming function.

Recall that most time is spent in calculating the toughness, thus improving the current implementation of the toughness function would be more useful than improving the efficiency of the Hamilton path algorithm. An interesting optimisation would be to examine how certain calculations of the parent graph can be reused to calculate the toughness of the offspring. If this approach is taken, it may also be useful to analyse the biased algorithm from Section 5.2.4 in more detail.

Finally, it would be useful to obtain more theoretical results on the construction of nonhamiltonian tough graphs. In particular, to devise other constructions to build a tough nonhamiltonian graph G based on a smaller graph H, similar to the one explained in Section 2.3. If such a construction could be devised, it may be the most promising approach for future research. All research provided in this thesis together with the aforementioned suggested research could then be applied to different constructions. The reuse of our work would be very straightforward if formulas to calculate the toughness similar to those presented in Section 4.4 could be devised for alternative constructions.

Bibliography

[Gil59] E. N. Gilbert. 'Random Graphs'. In: *Ann. Math. Statist.* 30.4 (Dec. 1959), pp. 1141–1144. DOI: 10.1214/aoms/1177706098.

[HK61] Michael Held and Richard M. Karp. 'A Dynamic Programming Approach to Sequencing Problems'. In: *Proceedings of the 1961 16th ACM National Meeting*. ACM '61. New York, NY, USA: Association for Computing Machinery, 1961, pp. 71.201–71.204. ISBN: 9781450373883. DOI: 10.1145/800029.808532.

[Bel62] Richard Bellman. 'Dynamic Programming Treatment of the Travelling Salesman Problem'. In: *J. ACM* 9.1 (1962), pp. 61–63. DOI: 10.1145/321105.321111.

[Chv72] V Chvátal. 'On Hamilton's ideals'. In: *Journal of Combinatorial Theory, Series B* 12.2 (1972), pp. 163–168. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(72)90020-2.

[Kar72] Richard M. Karp. 'Reducibility Among Combinatorial Problems'. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[Chv73] Vasek Chvátal. 'Tough graphs and hamiltonian circuits'. In: *Discrete Mathematics* 5.3 (1973), pp. 215–228. DOI: 10.1016/0012-365X(73)90138-6.

[Wil73] James E. Williamson. 'On Hamiltonian Connected Graphs'. 1973. URL: https://scholarworks.wmich.edu/cgi/viewcontent.cgi?article=3965&context=dissertations.

[Ber78] J.C. Bermond. 'Hamiltonian Graphs'. In: *Beineke, L., Wilson, R.J. (eds.), Selected Topics in Graph Theory*. Academic Press, London and New York, 1978, pp. 127–167.

[TY84] Robert Endre Tarjan and Mihalis Yannakakis. 'Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs'. In: *SIAM J. Comput.* 13.3 (1984), pp. 566–579. DOI: 10.1137/0213035.

[BHS90] Douglas Bauer, S. Louis Hakimi and Edward F. Schmeichel. 'Recognizing tough graphs is NP-hard'. In: *Discrete Applied Mathematics* 28.3 (1990), pp. 191–195. DOI: 10.1016/0166-218X(90)90001-S.

[Koz93] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN: 978-0-262-11170-6.

[Che+98] Guantao Chen et al. 'Tough enough chordal graphs are Hamiltonian'. In: *Networks* 31.1 (1998), pp. 29–38. DOI: 10.1002/(SICI)1097-0037(199801)31:1<29::AID-NET4>3.0.CO;2-M.

[McK98] Brendan D. McKay. 'Isomorph-Free Exhaustive Generation'. In: *J. Algorithms* 26.2 (1998), pp. 306–324. DOI: 10.1006/jagm.1997.0898.

[Van98] Basil Vandegriend. 'Finding Hamiltonian cycles: algorithms, graphs and performance'. 1998. URL: https://webdocs.cs.ualberta.ca/~joe/Theses/vandegriend.html.

[BBV00] Douglas Bauer, Hajo Broersma and Henk Jan Veldman. 'Not Every 2-tough Graph Is Hamiltonian'. In: *Discrete Applied Mathematics* 99.1-3 (2000), pp. 317–321. DOI: 10.1016/S0166-218X(99)00141-9.

[Glo+00] A. Globus et al. 'JavaGenes: Evolving Graphs with Crossover'. NAS Technical Report NAS-00-018. 2000.

[SLL02] Jeremy G. Siek, Lie-Quan Lee and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002. ISBN: 978-0-201-72914-6.

[SM02] K. O. Stanley and R. Miikkulainen. 'Evolving Neural Networks through Augmenting Topologies'. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127.

[YJG03] Andy B. Yoo, Morris A. Jette and Mark Grondona. 'SLURM: Simple Linux Utility for Resource Management'. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[Spi04] Jeremy P. Spinrad. 'A note on computing graph closures'. In: *Discret. Math.* 276.1-3 (2004), pp. 327–329. DOI: 10.1016/S0012-365X(03)00316-9.

[SPC04] S. Stone, B. Pillmore and W. Cyre. 'Crossover and mutation in genetic algorithms using graph-encoded chromosomes'. GECCO conference. 2004.

[BBS06] Douglas Bauer, Hajo Broersma and Edward F. Schmeichel. 'Toughness in Graphs - A Survey'. In: *Graphs and Combinatorics* 22.1 (2006), pp. 1–35. DOI: 10.1007/s00373-006-0649-0.

[CN06] Gabor Csardi and Tamas Nepusz. 'The igraph software package for complex network research'. In: *InterJournal* Complex Systems (2006), p. 1695. URL: http://igraph.org.

[Jon06] Kenneth A. De Jong. *Evolutionary computation - a unified approach*. MIT Press, 2006. ISBN: 978-0-262-04194-2.

[SL07] Michael D. Schmidt and Hod Lipson. 'Comparison of tree and graph encodings as function of problem complexity'. In: *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*. 2007, pp. 1674–1679. DOI: 10.1145/1276958.1277288.

[BM08] J. Adrian Bondy and Uppaluri S. R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer, 2008. ISBN: 978-1-84628-970-5.

[Cha08] Andrew Chalaturnyk. 'A Fast Algorithm For Finding Hamilton Cycles'. 2008. URL: http://www.combinatorialmath.ca/G&G/chalaturnykthesis.pdf.

[HSS08] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart. 'Exploring Network Structure, Dynamics, and Function using NetworkX'. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[Bjö10] Andreas Björklund. 'Determinant Sums for Undirected Hamiltonicity'. In: *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. 2010, pp. 173–182. DOI: 10.1109/FOCS.2010.24.

[Bri+13] Gunnar Brinkmann et al. 'House of Graphs: A database of interesting graphs'. In: *Discrete Applied Mathematics* 161.1-2 (2013), pp. 311–314. DOI: 10.1016/j.dam.2012.07.018.

[MP14] Brendan D. McKay and Adolfo Piperno. 'Practical graph isomorphism, {II}'. In: *Journal of Symbolic Computation* 60.0 (2014), pp. 94–112. ISSN: 0747-7171. DOI: http://dx.doi.org/10.1016/j.jsc.2013.09.003.

[Pei14] Tiago P. Peixoto. 'The graph-tool python library'. In: *figshare* (2014). DOI: 10.6084/m9.figshare.1164194.

[Bro15] Hajo Broersma. 'How tough is toughness?' In: *Bulletin of the EATCS* 117 (2015). URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/362.

[KK17] Adam Kabela and Tomás Kaiser. '10-tough chordal graphs are Hamiltonian'. In: *J. Comb. Theory, Ser. B* 122 (2017), pp. 417–427. DOI: 10.1016/j.jctb.2016.07.002.

[APS18] Timothy Atkinson, Detlef Plump and Susan Stepney. 'Evolving Graphs by Graph Programming'. In: *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*. 2018, pp. 35–51. DOI: 10.1007/978-3-319-77553-1_3.

[Mee18] Barbara Meersman. 'Grafen met weinig hamiltoniaanse cykels'. 2018. URL: https://lib.ugent.be/fulltxt/RUG01/002/479/656/RUG01-002479656_2018_0001_AC.pdf.

[MR19] Abdul Manazir and Khalid Raza. 'Recent Developments in Cartesian Genetic Programming and its Variants'. In: *ACM Comput. Surv.* 51.6 (2019), 122:1–122:29. DOI: 10.1145/3275518.

[Boo20] Boost. *The Boost Graph Library*. June 2020. URL: https://www.boost.org/doc/libs/1_73_0/libs/graph/doc/index.html.

[Com20] Wikimedia Commons. *Graph of 20-fullerene w-nodes*. July 2020. URL: https://commons.wikimedia.org/wiki/File:Graph_of_20-fullerene_w-nodes.svg.

[GMZ20] Jan Goedgebeur, Barbara Meersman and Carol T. Zamfirescu. 'Graphs with few hamiltonian cycles'. In: *Math. Comput.* 89.322 (2020), pp. 965–991. DOI: 10.1090/mcom/3465.

[Inc20a] The OEIS Foundation Inc. *The On-Line Encyclopedia of Integer Sequences*. June 2020. URL: https://oeis.org/A001349.

[Inc20b] The OEIS Foundation Inc. *The On-Line Encyclopedia of Integer Sequences*. June 2020. URL: https://oeis.org/A048192.

[McK20a] Brendan D. McKay. *Combinatorial data - graphs*. June 2020. URL: http://users.cecs.anu.edu.au/~bdm/data/graphs.html.

[McK20b] Brendan D. McKay. *Description of graph6, sparse6 and digraph6 encodings*. June 2020. URL: http://users.cecs.anu.edu.au/~bdm/data/formats.txt.

[TOT20] TOTO. *An open database for tree compositions*. July 2020. URL: http://treedecompositions.com/.

[Twe20] University of Twente. *University of Twente — DSI computing lab*. June 2020. URL: https://fmt.ewi.utwente.nl/redmine/projects/ctit_user/wiki.

[Wik20] Wikipedia. *Regular dodecahedron — Wikipedia, The Free Encyclopedia"*. July 2020. URL: https://en.wikipedia.org/wiki/Regular_dodecahedron#/media/File:Dodecahedron.jpg.

Appendix A

Code

Our C++ implementation is open source and available on GitHub.

`https://github.com/Timkemp1/tough-nonhamiltonian-graphs`

The `Graph` class contains functionality used by both the enumeration and the evolutionary algorithm. Both the `EnumerationGraph` class and the `EvolutionGraph` class inherit from the `Graph` class. The enumeration algorithm can be run by the static `EnumerationAlgorithm::read_file` function. The input data required to run the code can be downloaded from the web page in [McK20a]. The evolutionary algorithm can be run by instantiating the `EvolutionaryAlgorithm` class. Examples to illustrate how to run our code are shown in `main.cpp`. The code is also available on Zenodo.

`https://doi.org/10.5281/zenodo.3974274`

Appendix B

Output enumeration algorithm

The output of the enumeration algorithm for graphs of order 11 is shown in Figure B.1 and Figure B.2. Note that only half of the graphs are presented here, the other half consists of similar graphs where the vertices u and v are joined by an edge.

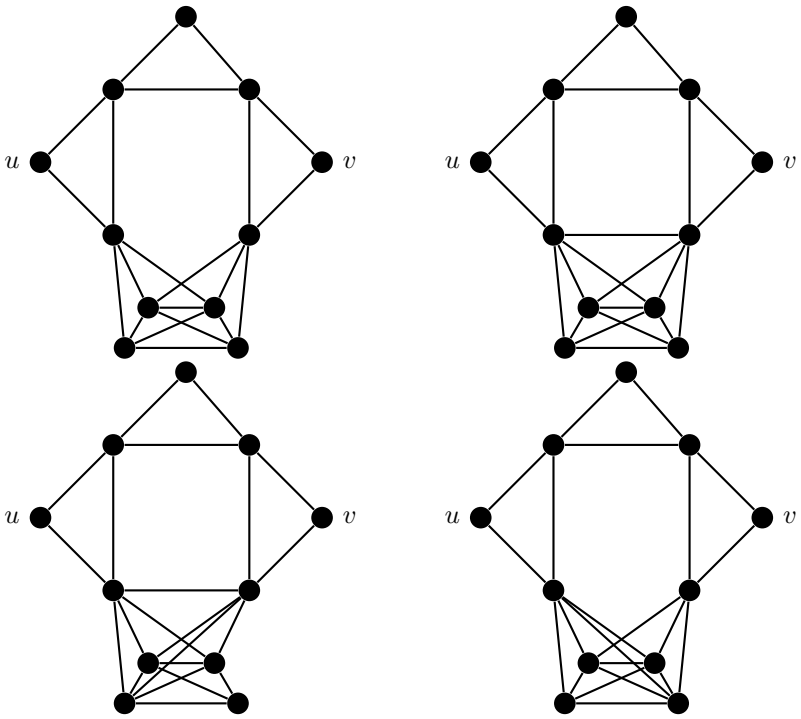


Figure B.1: Four out of the eighteen graphs of order 11 leading to $\tau \approx \frac{9}{4}$.

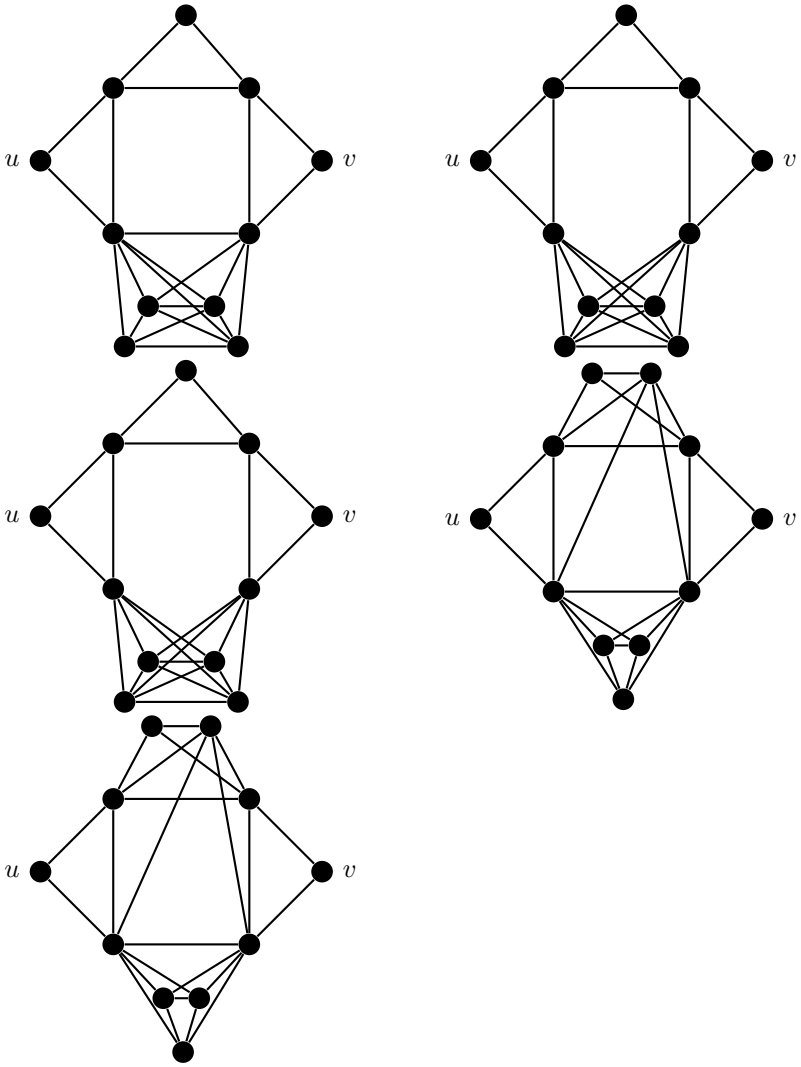
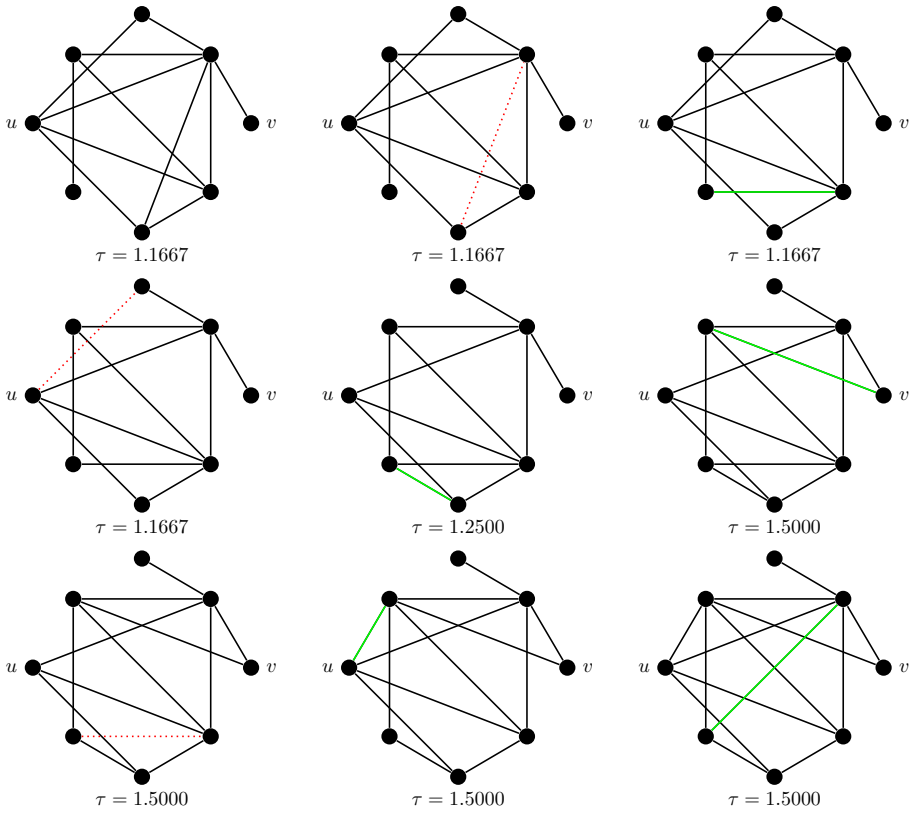


Figure B.2: Five out of the eighteen graphs of order 11 leading to $\tau \approx \frac{9}{4}$.

Appendix C

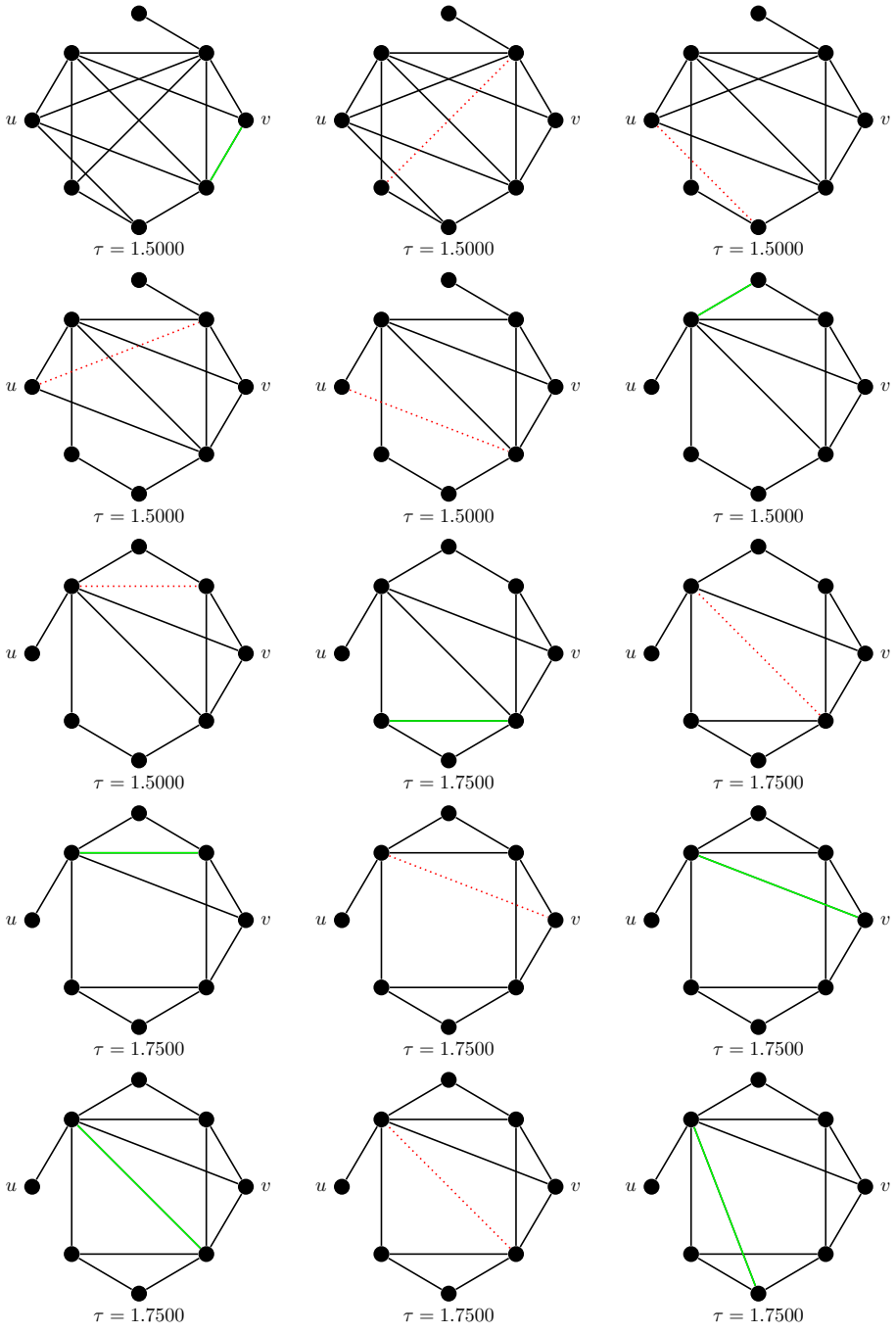
Example run

Table C.1: An example run of the evolutionary algorithm. A newly added edge is shown in green, and a removed edge is shown in red as a dotted line. Iterations where the parent is selected as the survivor are omitted.



Continued on next page

Table C.1 – *Continued from previous page*



Continued on next page

Table C.1 – *Continued from previous page*