Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Reducible Configurations for TSP on Subcubic Graphs

Diploma Thesis

2020

Bc. Adrián Kocúrek

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Reducible Configurations for TSP on Subcubic Graphs

Diploma Thesis

Study Programme:	Computer Science
Field of Study:	2508 Informatika
Department:	Department of Computer Science
Supervisor:	RNDr. Róbert Lukoťka, PhD.

Bratislava, 2020
Bc. Adrián Kocúrek

67393877

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Adrián Kocúrek
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Reducible configurations for TSP on subcubic graphs

Annotation: Dvořák, Kráľ, and Mohar conjectured that each connected graph that contains only vertices of degree 2 and 3 contains a closed spanning walk of size at most 1.25 n_3+ 1.75 n_2, where n_i denotes the number of vertices of degree i. They found several reductions with respect of this conjecture. The aim of the thesis is to find other reducible configurations with emphasis on reductions of circuits of length 7. The search for reductions will probably require computer assistance.

Supervisor:	doc. RNDr. Robert Lukoťka, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	25.09.2018
Approved:	04.05.2021

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

....

Student Supervisor

Abstrakt

KOCÚREK, Adrián: Reducibilné konfigurácie pre TSP na subkubických grafoch [Diplomová práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra informatiky. Školiteľ: RNDr. Róbert Lukoťka, PhD. Bratislava FMFI UK, 2021.

Cieľom tejto práce bolo preskúmať existenciu redukcií v zatiaľ nepopísaných inštanciách subkubických grafov. Práca je členená na 5 kapitol. V prvej kapitole definujeme základné konvencie a pojmy, ktoré sa v texte vyskytujú. Druhá kapitola sa zaoberá momentálnym stavom problematiky zrýchlenia riešenia grafického TSP. Zároveň hlbšie popisuje konštrukcie redukcií, ktoré sú neskôr v práci použité na testovanie. Tretia kapitola stručne popisuje to, ako sme si prácu rozvrhli a kroky, ktoré sme pri jej riešení absolvovali. V štvrtej kapitole sa sústredíme na popis nášho riešenia a jeho implementáciu. V piatej, poslednej, kapitole sa zaoberáme popisom experimentov ktoré sme po vyhotovení a otestovaní nášho programu uskutočnili. 45 strán.

Kľúčové slová: TSP, grafy, redukcie, stromy

Abstract

KOCÚREK, Adrián: Reducible Configurations for TSP on Subcubic Graphs [Master Thesis].Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics. Supervisor: RNDr. Róbert Lukoťka, PhD. Bratislava FMFI UK, 2021.

The goal of this thesis was to search for reductions on as of now untapped instances of subcubic graphs. The thesis is divided into 5 chapters. The first chapter serves as an introduction to the terminology used during in the text. The second chapter discusses the current state of the art in the saearch for faster solutions to graphic TSP problem. It also delves into the details of the construction of reductions, which are later used for testing purposes. The third chapter briefly discusses the overall plan for achieving our goals and the steps we set out to complete. The fourth chapter provides an extended look into the implementation of the needed program. The fifth, and final, chapter discusses the various experiments which have been conducted after we finished the implementation and testing of our program. 45 pages.

Keywords: TSP, graphs, reductions, trees

Contents

List of Figures

Introduction

The field of computer science has many famous hard to solve problems. One of the more famous of these is the Travelling Salesman Problem, often abbreviated as simply TSP. This problem is often used to teach many of the basic properties of hard problems, such as their time general time complexities and relationship with approximations. Despite being known for so long and studied so thoroughly, there is still a lot of research being done on it. While the general problem is relatively well established, there are still many interesting and slightly more constrained instances of the problem which are being worked on day and night. One of these is the subcubic version of the problem. Recently there have been numerous advancements in the field of approximating this subset and one of these served as the inspiration for this work.

Dvořák, Kráľ, and Mohar conjectured that each connected graph that contains only vertices of degree two and three contains a closed spanning walk of size at most $1.25n_3 + 1.75n_2$, where n_i denotes the number of vertices of degree i[2]. They found several reductions with respect of this conjecture. The direct aim of our thesis was to find other reducible configurations with emphasis on reductions of circuits of length 7, so called seven-cycles. The search for reductions was this time realised with the direct aid of computing power, as the problem grows in complexity as the cycle length expands and it becomes simply too big to handle one case at a time by rote checking.

The very first chapter of our work introduces some of the basic terminology we are dealing with, both on the theoretical side and the practical side. Not only does it talk about the basics of graph notation, it also introduces the concept of a reduction as it has been used by Dvořák, Kráľ, and Mohar, serving as the backbone for all work done here.

The second chapter then tries to provide a good overview of the state of the problem at large. It establishes the current state of the art approximation bounds presented by people working in the field. It also delves deeper into reductions, providing an extended look into the state in which we found ourselves at the start as far as subcubic graph reductions of degree six were concerned. These not only serve as a guideline for what to expect of the problem at hand, but also serve as useful test subjects to prove the validity of our own program.

The third chapter sets up the general outline of the plan we have came up with to divide the implementation and experiments into smaller tasks which could be tackled one by one and build on top of each other to provide a framework that could both be used and relied upon to provide believable results.

The fourth chapter then delves into the process of implementation. Detailing each component of the resulting software both to explain how it has been built and also to serve as a useful reference for anyone who would want to verify the results by themselves or expand upon the developed algorithms and experiments further.

The final chapter then talks about the various experiments we have conducted. It separately deals with various instances of the problem of graphs containing six to ten vertice long cycles. Accompanying them are illustrations which help visualise the problems tackled. These results are largely proofs of non-existence, even if some reductions are eventually found and documented.

Chapter 1

Terminology

1.1 Notation

Our work concerns simple graphs, that is ones with no parallel edges. For a given G we denote its vertex set as $V(G)$, its edge set as $E(G)$. Furthermore, the number of vertices of the graph is denoted by $n(G)$ and the number of vertices of specifically degree 2 by $n_2(G)$. We signal the removal of a vertex w by noting the resulting graph as $G - w$. This graph contains neither the vertex w nor any of the edges incident with it. Continuing the pattern, if we remove a subset of vertices W, then $G - W$ marks the resulting graph. If we choose to remove edges but retain all the vertices in the graph as-is, we mark this as $G \backslash F$ where F is the set of edges to be excluded.

Subcubic graphs are such that have no vertex of degree higher than 3. We consider graph G k-connected if it is connected and should we remove at most $k - 1$ vertices from it, it will stay that way. If G is connected by not 2-connected, then any vertex c, the removal of which leads to the resulting $G - v$ graph not being connected, is called a cut-vertex. By analogy we call an edge which, if removed turns the graph to lose its connected property to be a cut-edge. Note that in subcubic graphs with more than 2 vertices, being 2-connected and being 2-edge-connected are equivalent terms.

We say that a multigraph G is Eulerian if all its vertices are of even degree. If the graph is Eulerian and connected, it has what's called an Eulerian tour. A subgraph G' of graph G is spanning if its vertex set matches that of the original graph.

1.2 Reductions

The idea of reductions rests upon the following conjecture:

Conjecture 1. *Every 2-connected, subcubic, n-vertex graph with n_2 vertices of degree 2 has a TSP walk of length at most: $\frac{5}{4}n + \frac{1}{4}n_2 - 1$.*

Adopting the nomenclature of previous works concerning this problem, we will define a reduction as follows: For any two subcubic graphs G and G', let

$$\delta(G, G') = (n(G) + n_2(G)) - (n(G') + n_2(G'))$$

We will call the 2-connected subcubic graph G' a reducation of the 2-connected subcubic graph G if $n(G') < n(G), \delta(G, G') \geq 0$, and there exists a linear-time algorith which creates a spanning Eulerian subgraph F on G from a given Eulerian subgraph F' on G' while satisfying

$$exc(F) \leq exc(F') + \frac{\delta(G, G')}{4}$$

The importance of this construct is that it gives us a formally defined relation with which we can reduce 2-connected subcubic graphs such that their spanning Eulerian subgraphs conatin fewer edges.

The majority of the work concerns itself with 'general' reductions. That is ones which can be applied without the need for additional assumptions about the rest of the graph. When we make an excursion into 'partial' reductions (that is non-general reductions), we will make an explicit note of it.

Chapter 2

Recent Achievements and The Open Questions

This thesis concerns itself with the practical and theoretical aspects of solving the travelling salesman problem with increased efficiency. Since the problem in question is NP-complete, most approaches attempt to do this using some form of approximation. These algorithms when applied to the graph version of the TSP problem, reduce the number of edges which need to be considered when looking for the optimal solution. Once they compute reach a solution, they retrace these steps to build an approximate of the optimum for the original problem.

2.1 Comparison of Approximation Methods

2.1.1 Algorithmic Complexity

The general TSP problem has been shown to be NP-complete and the best approximation available is still the 3/2 factor achieved by Christofides[1]. This problem has also been shown to be inapproximable to the factor of 123/122[5]. The graphic TSP is a special case of the general TSP problem which gives each path a unit weigh and asks for the shortest Hamiltonian walk in a given graph. This special case has shown progress thanks to the breaking of the 3/2 factor barrier when applied on cubic, 3-connected graphs[3]. The best algorithm currently available has an approximation factor of 7/5ths[7].

Another line of research concerning itself solely with the application on cubic graphs has progressed even further, enabling us to achieve a 9/7-approximation[4] of a given problem if the graph itself is 2-connected. If we do restrict the domain further and focus on cubic bipartite graphs, we are able to achieve a factor of 5/4ths[8]. Even these restrictions have their limits, however, and it has been shown that the near-perfect approximations of graphic TSP (a factor of 535/534) and its cubic variant (a factor of

1153/1152) are NP-complete[6].

2.1.2 Computational complexity

Attempts have also been made to tackle the problem with heuristic approaches.These tackle the issue of practical use for the various algorithm presented so far. As the algorithms get closer to the optimal solution, they have been experimentally shown to get longer computation times. Likewise, the complexity of a given graph greatly affects the final computation, with complete graphs inflicting noticable slowdowns compared to sparse ones. An example of this is an algorithm which is able to solve instance of TSP in $O((2-\epsilon)^n)$. With an upper bound on the maximum degree of any given vertex influencing the epsilon.

Making use of this fact, the heuristic approaches try to reduce the number of edges in a graph through the use of probabilistic methods. The Lin-Kernighan Heuristic is capale of producing solutions within 5% of the optimum in close to $O(n^{2.2})$ time. Modern approaches use methods like frequencey quadrilaterals to strip down the graph, potentially even to a degree bounded form and then apply the relevant algorithm for the particular subproblem, like in the case of a $O(1.2312^n)$ time algorithm solving TSP for all subcubic graphs[9].

2.2 Approximation of Graphic TSP

The algorithm we are focusing on and currently provides the best known results is based on the use of linear time reductions. These routines are able to find a subgraph with specific properties and reduce it in linear time. The algorithm repeatedly calls upon these routines until the graph either becomes basic, or takes on several desired properties, achieving a so-called the "clean" state. Under these conditions, a polynomial algorithm is able to create a spanning Eulerian subgraph for the reduced graph which is then used to create a spanning Eulerian subgraph for the original graph[2].

2.2.1 Clean/Reduced Graphs

The class of graphs we are dealing with are 2-connected, subcubic graphs. The properties needed for a 2-connected graph G to be considered "clean" are as follows ?:

- It must be proper.
- No cycles of length at most 7 contain a vertex with a degree of 2.
- All cycles of length 6 which are not θ-cycles are pairwise disjoint.

- $*$ For every cycle $K = v_1...v_m$ where $m \leq 7$ in G, if both the edge v_1v_m and the edge v_2v_3 are contained in a 2-edge cut, then they form a 2-edge cut together.
- $*$ Every cycle $K = v_1...v_6$ satisfies at least one of the following:
    1. The cycle is a θ-cycle.
    2. Each of the edges exiting the cycle is contained in a 2-edge cut, but none of them form a 2-edge cut with any other.
    3. Each of the edges exiting the cycle is contained in a 2-edge cut and there exists a single pair of indices i and j which hold $1 \leq i < j \leq 6$. These two indices are exactly 3 apart and the exiting edges going out of their vertices form a 2-edge cut.
    4. Precisely one edge exiting the cycle is not contained in any 2-edge cut. Then there exists a partition of the vertices $G - V(K)$ into A and B such that $x_1, x_2, x_6 \in A$ and $x_3, x_4, x_5 \in B$. There exists only one edge connecting the two partitions and they both induce connected subgraphs of $G - V(K)$.

All of these requirements can be checked for in linear time and are results of analysis of the various forms cycles can take in 2-connected subcubic graphs. Each one represents a reduction which creates a new, reduced graph G', which is again checked to make sure whether it is clean, basic or requires further reductions to be made. The whole algorithm takes up $O(n^3)$ time.

2.2.2 Analysis of Cycles in Subcubic Graphs

In order to reduce a graph to a basic or clean state, we have to establish a set of methods which deal with the various situations on a case by case basis. An important observation is that when dealing with n-vertex, subcubic graphs, we can list all of their cycles containing at most k vertices of degree three in linear time as there can only exist at most $3 \times 2^{k-1}n$ of them.

Graphs of Variable Size

Case (1). *Linear time reduction for a non-basic, 2-connected, subcubic graph G that contains a cycle K with at most two vertices of degree three.*

Since G is 2-connected, K contains exactly two vertices of degree three, called v_1 and v_2. Let x_1 and x_2 be their neighbors outside of K. The algorithm needs to consider two special cases. The first one is when the cycle K forms a traingle. We get the reduction G' by simply removing the third vertex z. Once we obtain a spanning Eulerian subgraph F' for G', we will check for the v_1v_2 edge in it. If it exists, then the spanning Eulerian subgraph F for G will be given by removing the v_1v_2 edge and adding a v_1zv_2 path in its stead. If it doesn't exist, then we will have to add the whole cycle K to obtain a suitable F for G.

If instead K is not a triangle, we will create G' by removing all vertices of K and creating a new vertex z which will be connected to vertices x_1 and x_2. We will then check whether the spanning Eulerian subgraph F' contains this new x_1zx_2 path. If it does, we will obtain F by first removing the vertex z from F' and then adding back all of the vertices of K, but only adding edges x_1v_1, x_2v_2 and the longer of the two paths that existed between v_1 and v_2 in the original cycle.

Case (2)**.** *Linear time reduction for a non-basic, 2-connected, subcubic graph G that contains a cycle K with three vertices of degree three.*

The three vertices of degree three in K shall be called v_1, v_2 and v_3. Once again due to the 2-connected nature of K, each of the vertices has a neighbor outside of K called x_i. Furthermore, there must exist an internal path between each pair of the vertices of degree three which does not contain the third one. These are enumerated as P_i with the v_i not partaking in the particular route.

We construct the desired reduction G' by removing all the vertices of K and replacing them with a single new vertex z. This vertex will be joined to each of the neighbouring vertices x_i by new paths Q_i. Each of these paths must be as long as were the paths P_i inside the cycle itself with the exception of the shortest path which will be one edge longer. To get a spanning Eulerian subgraph F from the F' constructed for G', we will have to check for the inclusion of the vertex z. If it is isolated, we will create F by removing the newly added paths along with their vertices and substituting in the original cycle K. If it isn't and the subgraph contains any pair of the new Q_i paths, say $Q_i, Q_j; i < j$, we will remove the added vertices and replace them with the vertices of K, then adding edges for v_ix_i, v_jx_j and the edges from paths P_i and P_j.

Case (3)**.** *Linear time reduction for a non-basic, 2-connected, subcubic graph G that contains a cycle K with four vertices of degree three.*

We start by finding cycles in G which contains four vertices of degree three and pick the shortest one, called K. Once again, we will name these vertices $v_1, ...v_4$ and their nieghbors from outside K as $x_1, ...x_4$. Let P_i denotethe path between v_i and v_{i+1} in K with the indices looping around and enumerate the vertex lengths of these P_i as k_i. All of these summed togeter give us k.

At this point, we choose between one of two reductions dubbed $G_j, j \in 1, 2$ which create the new graph by removing P_j and P_{j+20} from the original graph G. We then check which of these reduced graphs is 2-connected and proceed with that one. The check for 2-connectedness fits into linear time.

Assume G_1 is the 2-connected one. If the removed paths both had $k_i = 0$, we will obtain G' by removing the vertices of K and adding new paths $x_1z_1x_4$ and $x_2z_2x_3$, z_i representing new vertices. Generating the spanning Eulerian subgraph F from F' will be done in one of these ways:

- If both z_i are isolated in F', then we'll create it by removing these two vertices and adding the whole cycle K back in.
- If only one of the z_i is isolated, say z_1 (the process is symmetric), we must consider two possible spanning subgraphs F_1 and F_2. F_1 is created by removing the added vertices, adding the vertices of K back into F' and edges x_1v_1, x_4v_4 and the ones from path P_4. F_2 is created in the same way, but instead of adding the edges of P_4, we add the edges of P_1, P_2 and P_3 instead. The final F will be picked based on the resulting excess of these two graphs with the smaller one being the solution.
- If neither of the z_i is an isolate in F', we generate F by removing them, adding back in all the vertices of K, connecting all v_i to their x_i neighbors and finally adding the edges from P_2 and P_4.

We then move on to cases where $k = 1$ or $k \geq 2$. The latter one allows us to create G' by removing K and connecting the neighbors of the cycle through x_1x_4 and x_2x_3. Once we obtain F', the construction of the spanning Eulerian subgraph F for G will follow different paths based on the existence of these edges in F':

- If neither of these egdes is in F', then F is obtained simply by adding the cycle K back in.
- If the edge x_1x_4 belongs to F', but x_2x_3 does not, we're going to create the two spanning Eulerian subgraphs F_1 and F_2, then pick the one with the smaller excess to be the F going forward.

  Somewhat irroring the previous construction, F_1 is obtained from F' by removing the x_1x_4 edge, adding in the vertices of K and then the edges for x_1v_1, x_4v_4 and the edges contained in path P_4. F_2 goes through the same process but adds the edges from paths P_1, P_2, P_3 instead.
- The case with $x_2x_3 \in F'$ and $x_1x_4 \notin F'$ is handled symmetrically.
- If both edges are a part of F', we once again go through with the construction of F_1, F_2 and picking one of these for their smaller final excess.

  The basic steps for the creation of both of these graphs are identical. We remove the new edges, add vertices from K and reconnect all of v_i with their respective neighbors x_i. F_1 adds edges from paths P_2 and P_4. F_2 adds edges from P_1 and P_3.

The final option with $k = 1$ is symmetrical, so we pick $k_1 = 1; k_i = 0, i \in 2, 3, 4$ to demonstrate the algorithm. G' is obtained by removing the cycle K and adding new paths x_1x_4 and x_2zx_3. Once that is done, we follow with another case breakdown based on their presence in F'.

- If neither of them is contained in F', we will construct a spanning EUlerian subgraph F by removing the vertex z and adding the cycle K back in.
- If one of them belongs to F', we will remove all three newly added edges and add the vertices of K back. Next we will add edges x_iv_i for those x_i which will end up with an odd degree. Finally, we will add in edges contained in three of the four paths P_i, selecting them to keep the Eulerian subgraph property of F.
- If both of them are contained in F', we will erase the newly added edges, return the vertices of K and add all the neighbor x_iv_i edges along with the edges of paths P_2 and P_4.

Case (4). *Linear time reduction for a non-basic, 2-connected, subcubic graph G of length five or six that contains a cycle K with five vertices of degree three.*

We denote the specific cycle of length five or six as K. If G contains cycles of both lengths, we'll pick the shorter one as our K. Without loss of generality, we can assume the existence of a path $v_1v_2v_3v_4v_5$ where v_i are the vertices of degree three. If K is of length five, then the edge v_1v_5 closes the cycle. If it is of length six, another vertex of degree two, z, and a path v_1zv_5 must exist. x_i will be the verticies neighboring v_i from outside the cycle.

We will then create two potential reductions in the form of G_1 ad G_2. G_1 is created by substituing the cycle K by edge x_5x_1 and a new vertex w connecting vertices x_2, x_3 and x_4. If G_1 is 2-connected, it is a viable reduction of graph G.

If it is not, we can construct G_2 by instead substituing edges x_2x_5 and a vertex w that is connected to all of x_1, x_3 and x_4. If K is of length six, we must also further subdivide the edge x_3w. This gives us a subgraph that is assuredly 2-connected and thus can be used as a reduction for G.

Proper Graphs of Size 6

The method gives further detail on the reductions possible for proper subgraphs of length six. These search for 2-edge cuts on paths between the v_i and their neighbors x_i. If found, they can then perform the needed reductions.

2.2.3 The Main Algorithm

Our overall goal is to construct a spanning Eulerian subgraph F of the original graph G with the following property:

$$exc(F) \leq \frac{2(n(G) + n_2(G))}{7} + 1$$

This property stems from the interaction between spanning Eluerian sugraphs and the minimum possible length of a TSP walk. Effectively, we're trying to reach the smallest possible of these which in turn yields the desired approximation.

We start off by examining G. If it is basic (a cycle, K_4, or a θ-graph), the creation of a subgraph is trivially done in polynomial time. If it is not, we must apply the polynomial reduction algorithm described earlier to create graph G' which can be either basic or clean. The basic case is handled in the same way as before. If it is clean, we must use another polynomial time algorithm which produces a spanning Eulerian subgraph F' which holds:

$$exc(F') \leq \frac{2(n(G) + n_2(G))}{7}$$

This algorithm retains the cleanliness of the given subgraph and works with the properties of 6-cycles inherent to the clean form. It takes the subgraph and outputs a collection of $m \leq n/2 + 2$ spanning Eulerian subgraphs $F_1, ..., F_m$ and probabilities $p_1, ..., p_m \geq 0, p_1 + ... + p_m = 1$ which signify the probabilty of the various F_i containing vertices from F. The algorithm then assigns charges to all vertices of degrees 2 and 3 in the subgraph. Chosing randomly using the provided $p_1, ..., p_m$ distribution, the algorithm then picks an F_i and updates the charges according to whether the vertex is a part of F_i or not, taking into account the length of the cycle the vertex is a part of. This has an expected decrease of charge being equal to $exc(F_i)$ and arrives at a non-zero selection of vertices to put in F. The final subgraph can then be used to find a spanning Eulerian subgraph for the graph G in polynomial time.

2.3 Approximation Through Heuristics

Taking a different look on the whole problem and even losing the requirement of restricting the problem size to a specific subset of problems with special properties like 2-connectedness, we reach heuristic methods. These aim to trim the given graph of many of its edges while preserving the important ones, those which are a part of the optimal Hamiltonian cycle and thus the optimal solution. There are multiple ways to achieve this while allowing the person using the algorithm to select a desired amount of reductions made to the graph. Unlike deterministic approaches like in the case of approximation, the assurances given are only probabilistic and so their use profits from knowledge of the underlying class of problems we are trying apply them to.

One of the most promising new methods of achieving this makes use of repeated observations of a set number of frequency quadrilaterals. Calculating the frequencies of edges appearing in specific subproblems and extrapolating that to select around one third of the total edges as unlikely to be in the final solution. This method can be iterated, but only a set number of times before the probabilty of removing important edges starts climbing. It manages to compute a sparse graph in $\mathrm{O}(Nn^2)$ time where N is the number of frequency quadrilaterals used in each iteration.

2.3.1 Frequency Quadrilaterals

Frequency quadrilaterals are a special kind of frequency graphs of the K_4 kind. To create one, we compute the six optimal 4-vertex paths between between each pair of the vertices given a certain distribution of weighed edges.

To do this, given a quadrilateral $ABCD$ in K_4, we start by selecting a pair of vertices, say A and B. We then evaluate both 4-vertex paths between these two endpoints, $ACDB$ and $ADCB$. The shorter of these paths will become a weighed edge AB in the frequency quadrilateral describing $ABCD$. We do this for each pair of vertices available until a new quadrilateral is formed. Within a frequency quadrilateral $ABCD$, all of the edges are going to have a frequency of either 5, 3 or 1 with pairwise non-adjacent edges sharing the same value. This leaves us with six distinct frequency quadrilaterals to consider going forward.

This three-way split of edges is going to be used going forward as we randomly compute N of these quadrilaterals for each considered edge. Given a TSP with n vertices, there are $\binom{n}{4}$ weighed quadrilaterals. Each of these contains six edges, so that every edge is included in $\binom{n-2}{2}$ quadrilaterals. Since the probability of each normal edge ending up with either of the three weighs is equal, the expected frequency of edge e is going to be $3N$. However, the edges belonging to the optimal Hamiltonian cycle are going to be different. Their frequencies will tend to 5 or 3, with the probabilities being X? $P(f=5) = P(f=3) = \frac{1}{3} + \frac{1}{3(n-2)}$, givig the expected frequency of $3N + \frac{2N}{n-2}$. Thus we arrive at the final observation which states that we can expect to preserve at most $\frac{2}{3}\binom{n}{2}$ edges in a single pass of the algorithm.

2.3.2 The Iterative Algorithm

The algorithm follows an iterative pattern. Given a graph G with vertices V and weighed edges E, it computes the average frequency of each edge $e \in E$ through N randomly chosen frequency quadrilaterals containing it. Then it orders the edges according to their average frequency and creates a new graph G' with the same set of vertices V, but only $\frac{2}{3}$ of the edges found in E getting into F'. This process keeps repeating until the number of edges in E drops below a set value, experimentally set to $nlog_2n$ with log_2n being the modifyable factor c.

Once the algorithm reaches a sparse graph with at most $nlog_2n$ edges, it ends and starts an algorithm of user's choosing which gives either an optimal or approximate solution on the new graph.

We can calculate the number of iterations given the formula:

$$k_{max} = \lfloor log_{\frac{2}{3}}(\frac{2c}{n-1})\rfloor$$

However, as we approach this value, a number of issues will start cropping up. To make sure the hieghtened frequency of edges contained in the optimal Hamiltonian cycle manifests itself, we try to compute as many frequency quadrilaterals as the computation time allows. But once the number of edges starts noticably decreasing, some of the generated frequency quadrilaterals will not be able to find the needed paths between the four chosen vertices. They can and will still be evaluated in this state, but their frequency readings will start skewing the results towards less accurate selections. It is because of this and similar problems related to higher frequencies starting to increase in number as we iterate further that the algorithm includes a stop mechanism. Once triggered, it will terminate the iteration even without having reached the $nlog_2n$ barrier. In practice, however, the graph will already be relatively sparse and further iterations would only lead to a high number of required edges getting cut.

2.4 Open Questions

The biggest open question at the moment is the final lower bound for these methods of finding approximate solutions both in the original problem and its slightly more specific variants.

Apart from these is the drive to find more efficient algorithms solving these problems with the same approximation factor. Most of the work done so far has focused on proving the possibility of solving these problems within certain error bounds, but with little regard for the complexity of the algorithms in question or their practical usage. Heuristic approaches tackle this problem with their own set of methods and there is also research being done in finding more succinct algorithms for solving the problems that have been proven as solvable with given assurances.

The latter of these drives also expands on the research of the reducibility of more complex subgraphs present within given problem sets. In particular, 6-cycles and 7-cycles inside 2-connected subcubic graphs and their provable properties. These are being expanded beyond the needs of immediate proofs for the algorithms' correctness in hopes of obtaining better results, or at the very least a look at the problem enabling these improvements from a wider perspetive.

Chapter 3

The Outline

In this chapter we will introduce the general outline of our work process. The first thing we will discuss is the principal theoretical basis for what we were trying to achieve and how we rationalised these moves. Next we will talk about the steps we had planned out, both practical and theoretical, to achieve the goals we have set for ourselves. Finally, we will discuss the interaction of these with the original theory and the goals of this work.

3.1 The Basis in Theory

The theory behind the way we look for reductions follows. We first make use of the conjecture1.2 and modify the factors by an ϵ value to make working with them easier. Any optimal TSP walk will use any edge up to 2 times. But as we are in a subcubic graph, these TSP will inevitably form cycles (creating a 2-factor). Thus we can reduce the TSP search to a search for these cycles.

As each factor can thus be described as having $G - 2 + F_i + 2 * F_k$ edges, where G denotes the number of vertices in the whole graph, F_i denotes the number of vertices outside of the 2-factor and F_k denotes the number of circles formed by the 2-factor F. We can thus combine this with the conjecture to arrive at $G - 2 + F_i + 2 * F_k \leq \frac{5}{4}n + \frac{1}{4}n_2 - 1$. Multiplying this by 4 we will rearrange the inequality to arrive at the final $1 <= G + n_2 - 4 * F_i + 8 * F_k$. Any proper reduction will thus have to check that as it changes the numbers of vertices and factors, it does not violate this property.

3.2 Main Guideline

Seeing that the problem space ahead would not be feasibly searched through by hand, we set out to create a set of tools which would allow us to, at first check, and then help in the searching of suitable reductions on any given graph. Our tool needed to be able to search through the entire problem space with as much efficiency as possible and provide confirmations of a given pair of graphs being in a reductive relationship. Or provide a conunterpoint with human-readable output that would allow us to find the scenario which broke the constraints imposed on both as a general reduction.

3.2.1 Initial Development

The first step would form the baseline of all of our further work. We needed to pick a suitable representation for the vertices and edges in the studied graphs. This would take note of additional requirements placed on these, in particular the separation of the so-called 'internal' and 'external' edges which would go on to serve a particular purpose unrelated to efficacy.

Building on this would be the first algorithm, a graph walking method that could take a set situation (regarding the graph itself and one specific factor on it) and calculate the approximate excess of a TSP walk along this factor as discussed in the section on theoretical grounds.

Once we had achieved this, we were able to test the implementation on the first real object, an exemplary reduction seen in the image below.

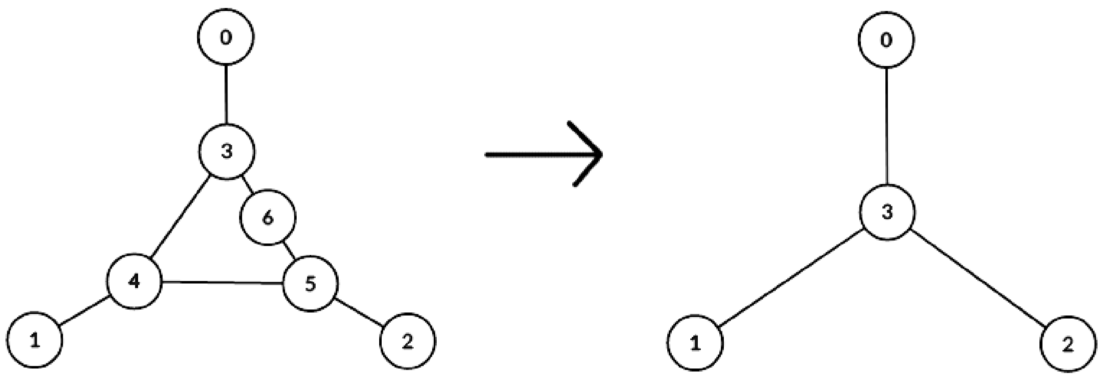

Figure 3.1: An exemplary reduction which removes 3 vertices

3.2.2 Intermediate Development

The next few steps would go on to expand on this functionality in an incremental fashion.

The first of these focused on finding and generating suitable 2-factors for the original graph should we provide a factored reduced grap and the external edges (identical) for both of them. Of course, such a factor need not exist we would need to make use of the previous method to check whether a given factor satisfies the constraints.

Then we would increase the scope even further and ask the general question of whether a given pair of graphs could serve as an example of a reduction. This time without providing any additional constraints. Here the plan would call for an unlimited search, once again making use of the previously developed methods to check its own outputs.

3.2.3 Final Developments

After this had been achieved, we would then start expanding the scope to stop asking for user's input altogether. At this point we planned to make use of specifically designed subgraphs to construct possible reductions and test them all en-masse. The choice of the subgraphs would fall on trees with properties best suited for possible reductions, particularly keeping the amount of stray edges down to a minimum.

Achieving that, we could then start our search for general reductions on graphs which have not been studied in detail up until this point. Focusing our attention primarily on instances of 7-cycles under various configrations.

Additionally, if we were to find any class of reductions, we could later try to prove their existence theoretically, or include additional restrictions into the program's algorithm and check whether they were completely general. A reduction accepted by our programme could still fail under additional assumptions, specifically when it creates a new cut in the studied graph.

Chapter 4

Implementation

We will now proceed to talk about the way we have created the software needed to fulfill the practical needs of our work. This chapter talks about the details regarding choice of the used data structures, the structure of the main class and its primary and secondary methods, the test suite developed to confirm the correctness of results, optimisations which were added to deal with the non-trivial computing workload and finally additional tools used to speed up the input process.

4.1 The Graph Class

The "Graph" class is the principal class in our program, its instances representing the various graphs we use throughout the operation of any of our experiments. Its main purpose is to hold all data structures pretaining to a single graph in a unified location as well as to provide facilities to pose individual questions to the graph. Its implementation is contained within the included file “graph.hpp”.

4.1.1 Fields and Data Structures

Edge Representation

The class's two main fields, 'edges' and 'outEdges', hold separate vectors of integers for the edges of the graph, one for the internal edges connecting the vertices of degree 2 and 3 within the subgraph's boundaries, and one for the so-called 'outgoing edges' of degree 1 which represent the fixed border area providing connections to the rest of the graph.

These arrays are arranged as neighbor lists in a single dimension to save on performance during the many traversals our algorithms will require. The array for internal edges is indexed by 3s with numbers ≥ 0 indicating either a neighboring vertex or -1 if there are no more edges present. The excess memory footprint created by these -1s is acceptable for the scale of inputs we are running the algorithms with, having no more than 12 vertices of degree 1 and generally as few vertices of degree 2 as manageable due to their negative contribution to reduction's worst case factorings. The external array is indexed by 1s as we assume there will only ever be one Eulerian cycle entering the subgraph at any given border/outgoing vertex. The rest of the graph not pretaining to the chosen subgraph is thus abstracted to a single edge connection. A practical example follows below:

```
std::vector<int> edges =
{3,-1,-1,    4,-1,-1,   5,-1,-1,   0,4,6,   1,3,5,   2,4,6,    3,5,-1};

std::vector<int> outEdges =
{2, -1, 0};
```

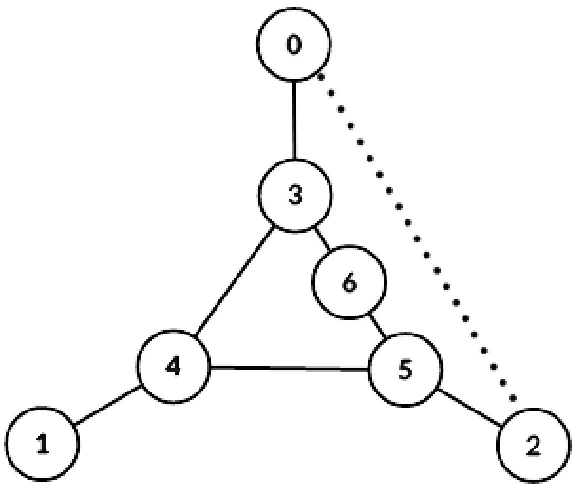

Figure 4.1: An exemplary graph created by these field parameters

When creating the internal edge neighbor list called 'edges', we will be following two additional requirements for its structure. The first of these is that outgoing edges of degree 1 must always be listed first, only then followed by the unsorted arrangement of the edges with higher degree. As long as we maintain this arrangement, we can use the number of these vertices, stored as the 'outVertCount' field created during instantiation and maintained through any changes, to determine whether the index we are at belongs to an outgoing vertex or an internal one. We can also use this fact to iterate up to said number to iterate through all vertices of this type. The other structural requirement is that the neighbor lists are ordered in ascending order, ignoring negative numbers which always come last. This is done to make finding outgoing connections faster, as these edges get priority during regular tree traversal, but are not otherwise marked in the array itself except by their index. Thus saving us the need to do a separate pass for outgoing and internal vertices on each vertex.

The separate list for outgoing edges, named 'outEdges' is another vector, this time representing a single connection through the portion of the graph outside of our scope. Numbers in it are complementary indices of two-way relationships between two different outgoing vertices. A -1 signifies the nonexistence of a path through the external graph from the vertex at a particular index back into the subgraph.

Methods' Internal States

The trio of integer vectors called 'orderedEdges', 'edgeOrder' and 'orderPartSums', left uninitiated during the creation of a Graph instance, represent internal state that needs to be preserved between calls of the two permutation methods: 'permuteEdges()' and 'permuteEdgesExtra()'. This allows said methods to provide stream outputs necessary for faster runtimes and ease of debugging. Their initiation, use cases and disposal will be covered in the section about these methods. Their purpose is as follows:

* 'orderedEdges' represents the internal edges of each outgoing vertex as they were before the first call of the methods.
* 'edgeOrder' represents a selector acting on the previous field, which decides the new order of all the outgoing vertices based on permutations of itself.
* 'orderPartSums' holds the offsets for the starts of each selector category, allowing the algorithm to rearrange the chosen vertices along the lines chosen by 'edgeOrder'.

Following these is the final trio of Graph object fields. Two stacks containing vectors of boolean values called 'maskStack' and 'visitStack', and a stack of integers called 'positionStack'. As with the three vectors before them, their purpose is to hold additional state between calls of a method. The method in question is a generator function named 'nextMask()' which will be detailed in the next section. The fields were chosen to be implemented by stacks due to the recursive nature of the algorithm. Their purpose is as follows:

* 'maskStack' holds a stack of booleans vectors, each of which represents a memorised mask of all the marked edges during a split situation in the 2-factor's path.

∗ 'visitStack' holds a stack of boolean vectors, also representing an instance in time but memorising the vertices which had been used by that point.

∗ 'positionStack' holds integers inserted and taken out as pairs of positions, one for the position at the time of insertion and one for the position from which it had came from.

This concludes all the fields present in a single instance of the Graph object.

4.1.2 Methods

The methods of the Graph class can be subdivided into two categories, 'main' and 'helper' methods. 'Main' methods are responsible for answering the main queries regarding progressively broader subset of the reduction relation. 'Helper' methods, meanwhile, provide additional functions and operations on the graph itself, giving us the option to add more edges, print out the current ones, generate the next mask for a certain 2-factor and set a new one. These are used internally by the 'main' methods but are given as public to be usable and testable without modifying the class itself. We will start by describing all the 'helper' methods in detail and then follow with the 'main' functions which also make use of them.

nextMask()

The first 'helper' method to introduce is 'nextMask()'. It's purpose is to progressively generate all viable 2-factors of the inherent subgraph and return an empty vector once there are none more available. To achieve this, the method interacts with two further methods, 'nextMask_reset()' and 'nextMask_walker()'. It also makes use of the three stacks outlined in the previous section to keep a certain internal state before returning a viable factor or the final empty vector. The full call signature of the method is thus:

```
std::vector<bool> nextMask(std::vector<bool> const &extMask)
```

To call the method, we must first make a call to 'nextMask_reset()' which will initialise the 'maskStack', 'visitStack' and 'positionStack' while giving them values representing a clean slate at the 0 starting position, an unreachable previous position of -2 and a pair of clean masks of appropriate size. This is invariant to instantiating a separate object with its own internal state which would act as a generator for a given graph. Creating an object like that was considered but decided against due to the additional overhead which would serve no purpose for our specialised class created for a very specific goal, rather than as the base of later expansion into a general purpose graph library.

Once it has been initialised, we can make a call to the 'nextMask()' method. The algorithm works in two distinct phases. The first phase goes through all of the outgoing vertices and tries to create a viable 2-factor. To do so, it attempts to find a cycle staring in every outgoing vertex that has an external edge marked as traversable (specifically used by a hypothetical TSP walk going through this subgraph). If this process fails, the algorithm looks at the stack of memorised situations where a split decision happened. These get inserted into the stack at every vertex with several viable ways forward. If it can't find one, it will return an uninitiated 'emptyMask' vector that will signal the end of the generator's usefulness to the callee. If it succeeds and finds a 2-factor which makes use of all the marked outgoing vertices, it will then step into the second phase of the process.

First it needs to make a temporary reference to the current stacks since these can still contain viable alternative paths that will have to be traversed in later calls of the method. Adding a new bottom to the three stacks would require more extensive changes at the level of the walking algorithm and we decided against it. This phase also copies the 'retMask' and 'visitMark' vectors used in the first phase. At this point it will run a modified version of the factor searching algorithm signalled by the changed 'sidepath' flag. This algorithm will be called on all the remaining edges to find any possible 2-factors among those internal edges that had not been used in the first phase. Unlike the first phase, it can greedily search through all the alternatives in one run and pick the one that covered the most vertices. The 'retMask' and 'visitMark' specific to this instance will be merged into the original ones and the algorithm will move forward. There is also no fail condition here. Whether a factor is found or not, it will return a viable result. Once all of this is done, 'retMask' is returned as a boolean vector marking all edges contained in the generated 2-factor of the subgraph.

The algorithm used to search for 2-factors is contained within the private 'nextMask_walker()' method. It is a graph walker which operates with a set of assumptions about the structure of the graph and a boolean flag 'sidepath' that alters one of these assumptions to allow it to handle starting both in an outgoing vertex and an internal one. The assumptions are thus:

- Any outgoing connections, should they exist and have not been used yet, are given preference over internal ones.
- The only viable sequence is such that contains one path into and from a given vertex. If we cross this boundary, the branch is terminated.
- Vertices of degree 2 naturally only have 1 way forward.
- Vertices of degree 3 have one branching opportunity.
- If the flag is set, we can travel to previously visited vertices as long as it happens along an unused edge.

The method terminates and returns *true* if it returned to an edge that has one of its edges marked (therefore completing the cycle). It returns $false$ if it ends up in a situation where it shorts a cycle, creating a vertex with three marked edges, or if it expends all of its movement options without achieving either of the previous finish states.

expandGraph()

The second of the 'helper' methods. Its purpose is straightforward. Given the necessary data, expand the present graph with a new set of vertices. The call signature follows:

```
void expandGraph(std::vector<int> const &addEdges,
                 std::vector<int> const &addMask)
```

As a specialised function developed for the needs of joining existing trees into forests, it requires the representation of the added graph's edges and the number of its outgoing vertices. It critically ignores the 'outEdges' field as this is fully overwritten by the method for finding reductions in a pair of graphs. Expanding the method to require updates to this field would be possible, but there is currently no use case for such a modification (which would add unnecessary overhead to every call). Internally, this method creates a composite neighbor list which reorders and appropriately renames the vertex references. It does this to maintain the strict order we assume in all of our neighbor lists. The new edges get added to the end of the respective neighbor chains with those of degree 1 being placed at the end of the outgoing vertex section and those of other orders being placed at the end of the list.

permuteEdges() and permuteEdgesExtra()

The third and final of the discussed 'helper' methods is 'permuteEdges()' along with its twin 'permuteEdgesExtra()'. Both of these methods serve the same purpose, generating all "interesting" permutations of the given graph representation. This is done because the individual trees in the reduction can map on the original graph in various ways. Comparing a certain graph and a forest that is potentially reducing it, we must cover all of the possible matings of the two. As this raises the factor in the overall algorithm complexity, we employ optimisation strategies to make sure only relevant permutations are evaluated further. During operation, these methods make use of the 'orderedEdges', 'edgeOrder' and 'orderPartSums' fields. The call signatures are listed below:

```
bool permuteEdges(std::vector<int> const &treeList)
bool permuteEdgesExtra()
```

Both of these methods, as in the case of 'nextMask' function as generators which could, if needed, be turned into its own class. We have decided to not do so because it was not necessary for our goals and would impart additional overhead in the overall runtime. Of these two functions, 'permuteEdges()' is the default choice while 'permuteEdgesExtra()' is the method as it used to be before optimisations. We have kept this functionality in a separate method due to our optimisation method possible not covering every single viable permutation available. It is used in more constricted scenarios to cover these within a more limited problem set where its wasteful algorithm does not make the computation unfeasible from a time standpoint. It has an additional restriction in that it can't be called on graphs which feature directly connected outgoing vertices. On the other hand, it does not require additional information about the permutations provided, allowing it to be called in different portions of the code.

permuteEdgesExtra() The algorithm chosen for 'permuteEdgesExtra()' makes use of the 'std::next_permutation()' function from the standard C++ <algorithm> library. On its first call, the method initialises the 'orderedEdges' field with the original edges of all the outgoing vertices. It then lexicographically sorts them using 'std::sort()' and rearranges the neighbor lists 'edges' accordingly before returning true. All subsequent calls of the method comprise of a call to 'std::next_permutation()' acting on the 'orderedEdges' field, rearranging the internal neighbor list according to the new permutation of vertex names and returning the obtained boolean value as output. $false$ return value signifies that all of the possible permutations have been achieved. This approach, however, is wasteful due to internal symmetries of both the cycles and forests we are comparing in our experiments.

permuteEdges() The algorithm chosen for 'permuteEdges()' seeks to address this issue. Unlike 'permuteEdgesExtra()', which generates all possible permutations, this algorithm seeks to generate all combinations of the underlying vertices. In many of the trees we use, the internal arrangement of vertices is highly symmetrical, presenting us with the opportunity to treat all internal rearrangements as identical and therefore lose the ordering requirement. To achieve this, we again make use of the standard 'std::next_permutation()' function but instead of permuting the edges themselves, we create a mask assigning vertices to each tree in the resulting graph. This is marked by the way of numbers $0...n$, n being the number of trees covering the graph. This mask is initialised into the 'edgeOrder' field and will be fed to 'std::next_permutation()'. Additionally, because of the way we create the forests, it is already sorted when we initialise it. The vertices are arranged in ascending manner with regards to the trees of the forest. The field 'orderedEdges' is initialised as before, though it is no longer directly responsible for the arrangement of outgoing vertices in the graph. Finally we initialise the 'orderPartSums'. This vector will serve as a reference to the position of the first vertex for each selector category.

All subsequent runs of the algorithm then create permutations of the selector vector, combine the three fields to figure out the relative positions of each vertex taking a position inside the the selector-defined tree. While this is running, we also check for and cull all repetitions of the same neighbor list as before, which can arise in graphs with multiple outside vertices connecting to the same internal vertex. A situation that occurs commonly in the composed forests. Once we run out of ways to permute the selector, it will return to the initial state and give us $false$ which we pass on as the return value.

'Main' Methods

Returning to the methods directly implementing required functionality, we will now discuss a trio of methods used to answer various questions about the status of reducibility.

evalMetric()

The first of these is 'evalMetric()'. The sole responsibility of this function is to walk through the entire graph according to the porvided 2-factor (divided between internal and external edges), count the number of cycles and isolates within it and then make a final tally of these numbers together with the counts of vertices of degree 2 and 3 to come to the final value of the estimated length of a TSP walk through this graph according to a modified version of equation$/refequation$. We quadruple the terms of the equation to get uniformly integer numbers, avoiding any potential pitfalls of floating point number comparisons. Its call signature is thus:

```
int evalMetric(std::vector<bool> const &intMask,
               std::vector<bool> const &extMask)
```

The algorithm used needs to walk through all the marked edges and determine the number of cycles they create. Since a 2-factor only allows for one incoming and one outgoing edge at any point along its length, the walking process can be kept fairly simple. It looks for edges going out of it, pruning all vertices which have already been marked as visited. Depending on the type of vertex the edge points to, it checks whether they are contained in the 2-factor. Should it run out of edges to traverse, it assumes it's in an outgoing vertex and checks the neighbor list of external edges.

This process runs seprately for the outgoing vertices and internal ones with slight modifications to the logic of the walker. Mainly due to the fact that while the external mask directly marks an outgoing vector as traversable, the mask of internal edges requires a different evaluation of this condition. Likewise, the insertion into the rest of the graph differs between the two.

Once the algorithm checks the final vertex of the graph, it has the number of cycles encountered along the way stored. It will then iterate through all the interal vertices of the graph and check whether they have been visited (meaning they are present in some 2-factor cycle) or not. Along the way it also counts the number of vertices of degrees 2 and 3 using the presence of -1 at the end of their neighbor list as the guideline. When all of this is said and done, it returns the final integer number according to the $X = 4 \cdot \mathit{isolates} + 8 \cdot \mathit{cycles} - 2 \cdot \mathit{doublets} - 1 \cdot \mathit{triplets}$ formula.

findFactor()

The other one of the 'main' methods is 'findFactor()'. Unlike all the other methods, this one is *static* and takes the instances of the Graph class it is evaluating as arguments. The purpose of this method is to determine whether the second graph given is a reduction of the first one. In doing so it will make use of most of the methods mentioned up to this point. It's call signature looks like this:

```
std::vector<bool> Graph::findFactor(Graph &origGraph,
                                    Graph &reducedGraph)
```

The first thing the method does is that it makes a copy of the existing 'outEdges' field from one of the graphs. As a reduction must share the same external connections as the graph it is reducing, it is assumed these are identical for both objects entiering this function. This is done because it will be changing these on both graphs during its operation. Next it prepares a vector of vectors of integers called 'iterStates'. These will serve as a pseudo-stack during algorithm's recusrive building and testing of elements. Its size being that of the field 'outEdges' and the initial value being set to all -1, effectively creating a graph with no external paths connecting any two parts of it.

It then enters a loop which will repeat $\mathit{size}(\mathit{outEdges})/2$ times. This iteration will thus fire off once for each of external edges (connecting a pair of outgoing vertices together) that could be found on either of these graphs. Within each of these iterations, the method automatically iterates through all the vectors currently available in 'iterStates'. It then tries out all the possible connections available in a forward fashion (meaning trying to connect each index number with all indexes higher than itself). Once it finds a suitable pair, it creates a new neighbor list exanding the one in the current iteration of 'iterStates' vectors and points both of the graphs' 'outGraph' fields to it.

Once it has found and set the new neighbor lists, the method prepares a mask to cover all the edges. This is required for the subsequent calls of the 'nextMask()' and 'evalMetric()' functions, while providing the option of constraining the masks in certain experiments.

reducTest() and minReducTest() are the two private functions called at this point. The purpose of 'minReducTest()' is to use the mask its been given to iterate through all the possible factors of the graph and find the one with the smallest estimated TSP length. To do so it makes succeessive calls to 'nextFactor()' and comparing the resulting metrics with the currently known minimum. This number is initialised with the INT_MAX value. Here it is called by the potential reduction graph as we are trying to find its worst-case scenarios for the given arrangement of external edges. If the method terminates without finding a single viable 2-factor, it returns a -3 which signals the 'findFactor()' algorithm to add this arrangement of external edges into the next itertation, but skip the later test against the values of the graph being reduced. We can safely ignore these as only viable 2-factors of the reduced graph are up to consideration when determining the 'reduction' relationship.

If the method terminates with a positive integer, we follow it up with a call to 'reducTest()' on the original graph. This function takes both a mask and the boundary number, gained from 'minReducTest()' to determine if any 2-factor of this instance of the original graph is being reduced by said instance of the reduction one. It does so by trying to find a 2-factor with a loweror equal estimate value to that of the 2-factor of the reduction. The method repeats the same process as 'minReducTest()' but this time, a failure to find and provide the needed 2-factor results in the termination of the whole algorithm, as it provides a counter-example to the 'reduction' relationship.

EX_DEBUG() When it comes to gaining insight as to why a certain reduction failed or succeeded, the method has an in-built macro called 'EX_DEBUG(x)'. Definiting the macro to evaluate to 'x' turns on the logging of each of these principal steps. We can thus clearly see the arrangement of the external edges, the estimated minimal value for that arrangement, or its non-existnce, and either a 2-factor which satisfied the relation or giving a message about the failure to find one. This functionality could also be implemented as a boolean switch in the fuction itself but its origin as a debugging tool made us leave it as it is.

PARTIAL_EXPERIMENT() The mathod also holds another macro, this time specialised in the search of the so-called 'partial' reductions. It is of fairly limited use and requires the help of the Python script 'PartialsSearch.py' to be used. What it does is create its own set of outputs, this time massive solely aimed at further automatic processing. When enabled, in conjunction with the change of early return statements to continue and the setting of the 'stopAtReduction' flag introduced later to $false$, the program's outputs start listing reductions along with situations that are unhandlable by them. We can thus gain an insight as to whether a certain problem even has a reduction in every conceivable scenario and where to look for them.

'findFactor()' thus repeats this process and successively creates and tests all the possible pairs, pairs of pairs, trios of pairs, et cetera until it exhausts the search space for these two graphs. If it fails along the way, it first retrieves the stored value of 'outEdges' field to the respective graphs and returns an empty vector of booleans. If it succeeds, it will return the very last 2-factor capable of matching or beating the boundary set for it while also restoring the graphs to their original states.

4.2 Correctness

Once we have established the general structure, algorithms and code behind the Graph and all of its methods, we turned our attention to creating an appropriate test suite. This was needed both as an aid during the debugging process as well as a necessary prerequisite to confirm the legitimacy of our findings. For this reason we've chosen two functions which run ahead of any experiments, confirming that no regressions had appeared if we changed the code and that we've indeed set up all the necessary components correctly. The two functions, called 'validityTests()' and 'minorTests()' each go through a set of predetermined problems we know the correct answers to and assert their responses accordingly.

4.2.1 validityTests()

The first of these functions tests the basic functionality of the Graph library. To do so, it goes through a number of scenarios trying to make sure the major components are behaving as expected, giving particular attention to the main reduction method, 'findFactor()'.

∗ That 'evalMetric()' method returns the same values we expect on a selection of tested graphs.

∗ That the 'nextMask()' method finds all of the possible 2-factors on one of these and ends correctly.

∗ That the 'nextMask()' works on graphs with multiple external edges.

∗ That 'findFactor()' confirms a known reduction seen in 3.1.

∗ That 'findFactor()' denies a known non-reduction from a modified 3.1 where the vertex number 7 gets replaced by an edge.

∗ That 'findFactor()' confirms a the identical graph to be a reduction of itself.

∗ That 'findFactor()' confirms two more reductions taken from the text of [2].

The function returns a *true* boolean value if all of its checks execute correctly. If one finds a problem, it will return *false* which is intercepted by an assertion of *true* being returned in the main function. We haven't created separate assertions for most of these tests due to their small number, but a person trying to confirm the correctness of our findings might want to add additional checks here. If it then becomes impractical to manually check which of the tests failed, the modification to create separate assertions is possible.

4.2.2 minorTests()

The second test function deals with particular reduction scenarios. As the main experiment involves functions which iterate through and combine a large number of different reductions, testing any one in particular is left to a separate function created for this purpose. All of its findings are then also incorporated into the overall validation proccess since we have reasoned about the expected outputs ourselves. The list of currently tested reductions on a six-cycle, seven-cycle and a ten-cycle graph is listed below:

- We confirm that the graph connecting neighboring outgoing vertices together into pairs is not a reduction of the full ten-cycle.
- We confirm that six-cycle with an edge of degree 2 is reducible.
- We confirm that a six-cycle with an added double edge is reducible with the graph mentioned by [2].
- We try several different versions of reductions on these three, to make sure only the correct one goes through.
- We finally make sure that reductions which came up during improper initiations of the seven-cycle do not work with the program now fixed.

As before, this list is freely extensible by a potential user of this program. The reason for its logical separation from the previous section was to provide a separate place to store and test redcutions which came up during experimentation and we were not sure whether they were true or simple bugs. Unlike in the case of 'validityTests()', there is no structure beyond creating a specialised space for these targeted tests. The tests can be commented out and it does not compromise the validity of provided outputs, though they are usually left on, unless they would create unnecessary clutter in the debugging logs.

4.3 Forest Reductions

The final portion of the main source files are the functions which handle the actual experiments. They are called through another function separating the exectuion and source files into clearly divided portions called 'majorTests()'. In this function we provide experimental problems provided as distinct sets of named vectors of integer vectors. These integer vectors serve as stripped down templates for creating and testing the desired graphs. An example of such a template is provided below:

```
std::vector<std::vector<int>> altSevenCycle_threeMixed = {
  {17,-1,-1,...  11,16,17,  10,12,18,  11,13,19,...  4,5,12},
  {17,-1,-1,...  11,16,17,  10,12,18,  6,11,13,...  4,5,13},
  {17,-1,-1,...   9,11,16,  10,12,18,  6,11,13,...  4,5,13},
};
std::vector<std::vector<int>> altSevenCycleOuts_threeMixed = {
  {20,10},{20,10},{20,10},
};
```

The same format is generated by the Python tool mentioned later in the text and is the format of choice for all the trees created as the building blocks for the reduction graphs/forests in the main experiment. Each of these is first defined, announced by a write into the logs or output announcing the start of a test using them. They are then fed to a function meant to unwrap them, create a desired Graph instance and start a search for reductions on each of these. This function, called 'groupTest()' also takes an additional boolean flag 'stopAtReduction'. The purpose of this flag is to change whether the searching algorithm stops at a first confirmed reduction in each individual graph, or continues after finding one and eventually lists out all that have been found.

4.3.1 Trees and Forests

treeTest()

'groupTest()' itself calls 'treeTest()', a function which serves as another middle-man. It's purpose is more involved, however, as it contains a large list of precomputed ways that any specific number of outgoing vertices can be covered with trees containing from two to seven outgoing vertices. We consider only up to seven outgoing vertices because that is the maximum we have obtained when creating trees under certain restrictions for this experiment. This list was automatically generated by a Python script not included in the work itself, but generating all ways a constrained set of integers can generate a given number is an elementary task. For the purposes of this experiment, this precomputed field contains instructions for up to thirteen total outgoing vertices, as any more is strictly out of scope for any of the experiments considered.

The function itself first determines how many outgoing vertices the input graph has. It then uses this as an index into its list of predetermined combinations and iterates through all of these. While doing so, it will call another function, 'forestBuilder()', to generate all the possible forests with the given profile. Given a profile such as '{2,2,5}', the 'forestBuilder()' function would be tasked to create and test all possible forests generated with two trees containing two outgoing vertices and one tree containing five outgoing vertices.

forestBuilder()

The final function in the chain of operations is the 'forestBuilder()'. Unlike most of the functions and methods used so far, it is implemented as a truly recursive function with a boolean return value. It serves two main purposes. One is to add the correct trees to build the desired forests. The other is to recognise when a certain forest has been built to completition and running 'findFactor()' on it and all of its reasonable permutations. These permutations are generated by a while loop served by the 'permuteEdges()' method called on the forest itself. The 'findFactor()' static method is run in each instance to find out whether the input graph currently being tested is reduced by the newly built forest. Depending on the value of the 'stopAtReduction' flag passed down to it by the previous enclosing functions, it will either terminte the moment a reduction has been found or continue to test all the forests available to it. The function is set up so that a *true* value at any of its outputs stops further recursions from happening. These reductions are, of course, communicated in a human-readable form as output.

When not at the bottom of the recursive chain, the algorithm has to follow the profile given to it as the input variable 'treeList'. This serves as a tuple of indices pointing to the collection of all the trees created for the experiment and contained in the linked file "trees.hpp". As these indices are sorted, we later added an optimisation to prevent creation of duplicate trees generated by permuting the order of the trees in question. We only need to test combinations with repetition, that is the relative order of the trees with the same number of outgoing vertices is unimpotant. This is due to the permutation function effectively testing both relative positions at some point in time. It itself still creates some wasted effort when any two trees are identical and this could be optimised further if needed.

What the 'forestBuilder()' function does to achieve this is that it looks for any chains of the same type of tree in the 'treeList()'. Once it has done that, it calls a helper function called 'nonRedundantCombos()'. This function generates the combinations with repetition on the number of elements matching the length of the chain and using all the options avialble for that particular type of tree, obtained as a parameter from the file "trees.hpp". Any such optimisation cuts down the number of recursions from the sheer exponentiatial permutations with repetitions to a smaller fraction of combinations with repetition. This list of combination is then iterated on and the recursion controlling index is moved along to account for the skip. The trees are always added to the forest by calling the 'expandTree()' method and feeding it data from the "trees.hpp" file at the correct index.

At the end of all of this, the program outputs any reductions found listing both the graph they reduce and their edge representation.

File "trees.hpp"

The basis for all of our experiments are reducctions created from various forests. In order to build such forests, we needed a set of precomputed trees and that is where the "trees.hpp" file comes in. It's purpose is to hold all of the trees created for this purpose in a separate namespace and provide a navigable interface so that other parts of the program can draw upon them.

We will start off by describing the trees. The boundary we have set for ourselves were trees with a maximum of twelve distinct vertices. As the number of such trees even under the assumption of subcubicity is still too large to be generated manually and we wanted to avoid any potential redundancies caused by listing a number of graphs that are bijective with each other, we further set a boundary for the length of any single edge. These were to never contain more than a single vertex of degree two in succession. The rationalle for this restriction came from the way vertices of degree two interact with the TSP length approximation used throughout. As we are looking at worst-case (that is minimal) scenarios in our reductions, adding any vertices of this kind only lowers this number further. While there exists a possibility that they could help by introudcing isolated vertices in specific scenarios, the practical results seen in previous research suggested that going beyond adding one extra vertex had no use.

With these restrictions in place, we have manually created all the subcubic trees, recreated them as graphs in visual graph editors and used our supplementary Python script to add their templates into the file itself. Using the format mentioned at the beginning of this section. These graphs were then further divided into subsets by the number of outgoing edges they contained and grouped together in three dimensional vectors of integers called 'treeGraphs' and 'treeMasks' which would serve as the interface for the rest of the program's functions. Whenever one referenced these vectors, the shift by 2 resulting from the lack of reductions creating individual points and deleting edges entirely. We made a quick test adding dummy values to get around this issue but the optimisations of the compiler took these out and we decided to leave it as it was, largely owning to the fact the program itself references these fields only in a minimal number of functions.

4.4 Additional Tools

On top of the source files surrounding the project in C++, we have developed a few small utilities in Python meant to automate several of the tasks regarding the input of graphs in our non-standard formats. The tool included in the appendix serves as a converter of the more widely used list of unidrectional edges denoted by the two end vertices int our nighbor list representation. We have made extensive use of it in conjunction with online graph drawing tool 'Graph Editor' serviced by Csacademy.com, though any visual graph editor which is capable of outputs in this format of edge representation could be used instead.

The operation of the tool is simple. The user changes the number of edges to however many are used in the graph they have drawn. They then run the algorithm on this set of edges, listed in the aforementioned endpoint format, delimited with the word "end". The tool then outputs the neighbor list, the number of all vertices and the number of outgoing vertices in the desired format, directly copy-pastable into the main program. Do note, however, that the tool provided does not account for the reordering of outgoing edges coming first. Any graphs fed into it must thus make sure to have the vertices correctly labelled, reserving the starting labels for those of degree one.

The other tool developed in Python had a specific name, 'PartialsSearch'. As the name implies, it was created as a tool for automated search for partial reductions. The tool is set up with the particular outgoing edge situations we want to find suitable partial reductions for. These come from regular outputs of the reduction searching algorithm telling us why a certain reduction failed. Operating on a much larger file generated with the 'PARTIAL_EXPERIMENT()' macro enabled, this script will generate a list of reductions that were successfull in this particular scenario. Not turning out with zero viable 2-factors, nor failing at reducing some 2-factors of the original graph.

Chapter 5

Results

Having finished the development and testing of the validity of the overall program, we started conducting experiments on the cycles of degree six and above. We paid particular attention to the seven-cycle with all edges of degree three, also denoted as ”7-7” in the source files, but not to the exclusion of all others.

5.1 Methodology

All of the tests were conducted as fixed additions to the ’mainTests()’ function in the main file of the source code. They were compiled with the GCC suite and ”-O3” optimisations due to the heavy workloads associated with any of the tests. ”7-7” cycle was tested both using the full edge permutation method and the optimised one. All the other tests, unless particularly small, were carried out with the optimised version only.

5.2 Expanded Search

The first set of experiments we have conducted concerned known general reductions we had known of before. This was both to confirm that our program can indeed find these and also to confirm or deny existence of any other ones matching the profile of our forests.

5.2.1 Six-cycle

As we have established in the section about the chapter about the current state of this problem, finding a general reduction for a six cycle with no vertices of degree two should be impossible without it creating an additional bridge in the graph.

Our program at this point found none. Drawing upon the same source, we also know that if we model a single edge beyond the six-cycle we can create a reduction.

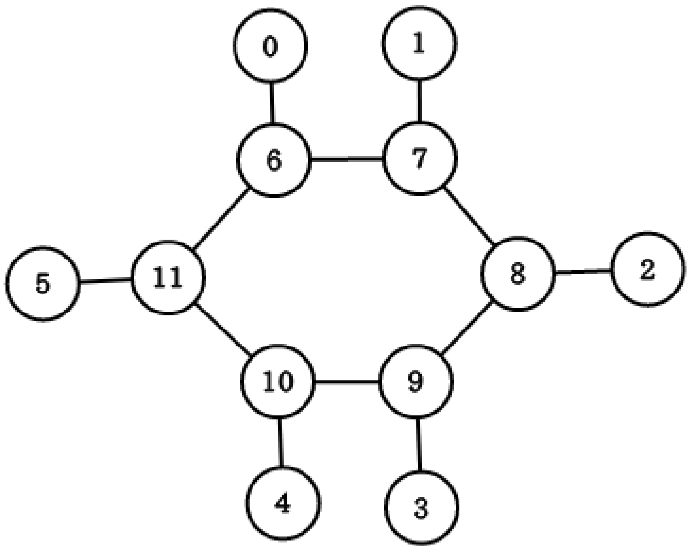

Figure 5.1: The full 6-cycle graph

This model requires the added vertex outside of the six-cycle to be of degree three. The reduction is shown in the picture (img.5.2).

Our program finds an instance of this general reduction. When we set set it to search for all possible reductions, however, no further reductions have been found. Providing us with the first practical result of the work.

5.2.2 Seven-cycle

While there was not much that could be said about full seven circles up to this point, it was known that a seven-cycle with at least a single vertex of degree two must contain a reduction[2]. Our program has been able to provide three distinct reduction for this, listed first in their text representation below and reconstructed graphically in the next image (img5.3). We observed that the structure of these graphs was relatively simple and seemingly lacked several more options which would still fit within the apparently lenient target for a reduction.

1. {1 -1 -1, 0 -1 -1, 3 -1 -1, 2 -1 -1, 5 -1 -1, 4 -1 -1}
2. {7 -1 -1, 7 -1 -1, 8 -1 -1, 8 -1 -1, 6 -1 -1, 6 -1 -1, 4 5 -1, 0 1 8, 2 3 7}
3. {7 -1 -1, 7 -1 -1, 8 -1 -1, 6 -1 -1, 6 -1 -1, 6 -1 -1, 3 4 5, 0 1 8, 2 7 -1}

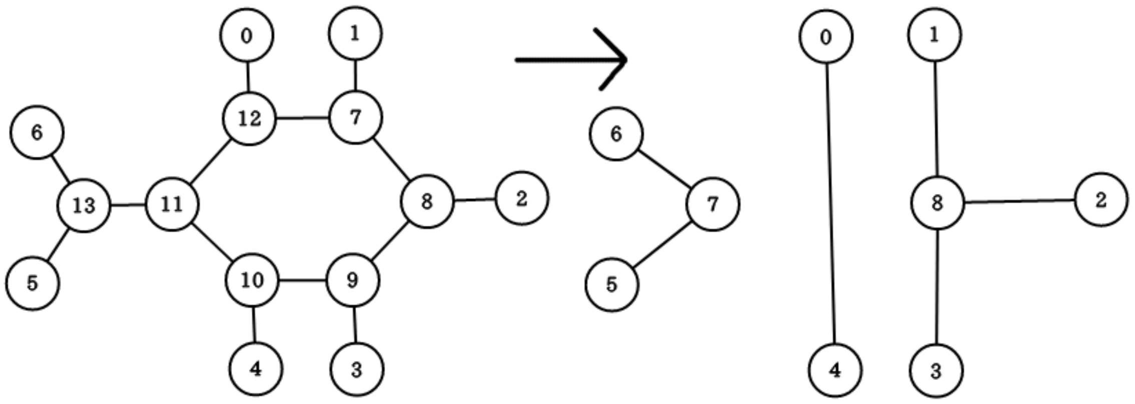

Figure 5.2: A general reduction on the expanded 6-cycle

Acting upon this suspicion and making use of the fact that a graph with seven outgoing vertices was well within computable range for our algorithm, we changed the permutation algorithm to 'permuteEdgesExtra()' and looked for additional outputs on graphs that are internally not completely symmtric. This search provided us with one additional reduction (#2+ in the image): {6 -1 -1, 6 -1 -1, 7 -1 -1, 8 -1 -1, 9 -1 -1, 10 -1 -1, 0 1 7, 2 6 -1, 3 9 10, 4 8 -1, 5 8 -1 }.

Not only has this given us another way of reducing the graph, we have have thus been forced to check for additional graphs wherever it was still feasible. Returning to the six-cycle, we were given a pair of new reductions, one for the full 6-cycle we assume is a reduction which creates an additional bridge, and one for the extended six-cycle. these are listed below:

{7 -1 -1, 7 -1 -1, 8 -1 -1, 9 -1 -1, 11 -1 -1, 11 -1 -1, 9 -1 -1, 0 1 8, 2 7 -1, 3 6 10, 9 11 -1, 4 5 10} for the extended six-cycle.

{6 -1 -1, 6 -1 -1, 7 -1 -1, 8 -1 -1, 8 -1 -1, 9 -1 -1, 0 1 7, 2 6 -1, 3 4 9, 5 8 -1} for the original one.

5.3 Full Reduction Search

Having expended our limited amount of previously known reductions relating to these problems, we set out to experiment on graphs where no results have been published yet. Starting from the seven-cycle and working our way up to eight-cycles and ten-cycles. Though still keeping the main focus on seven-cycle research.

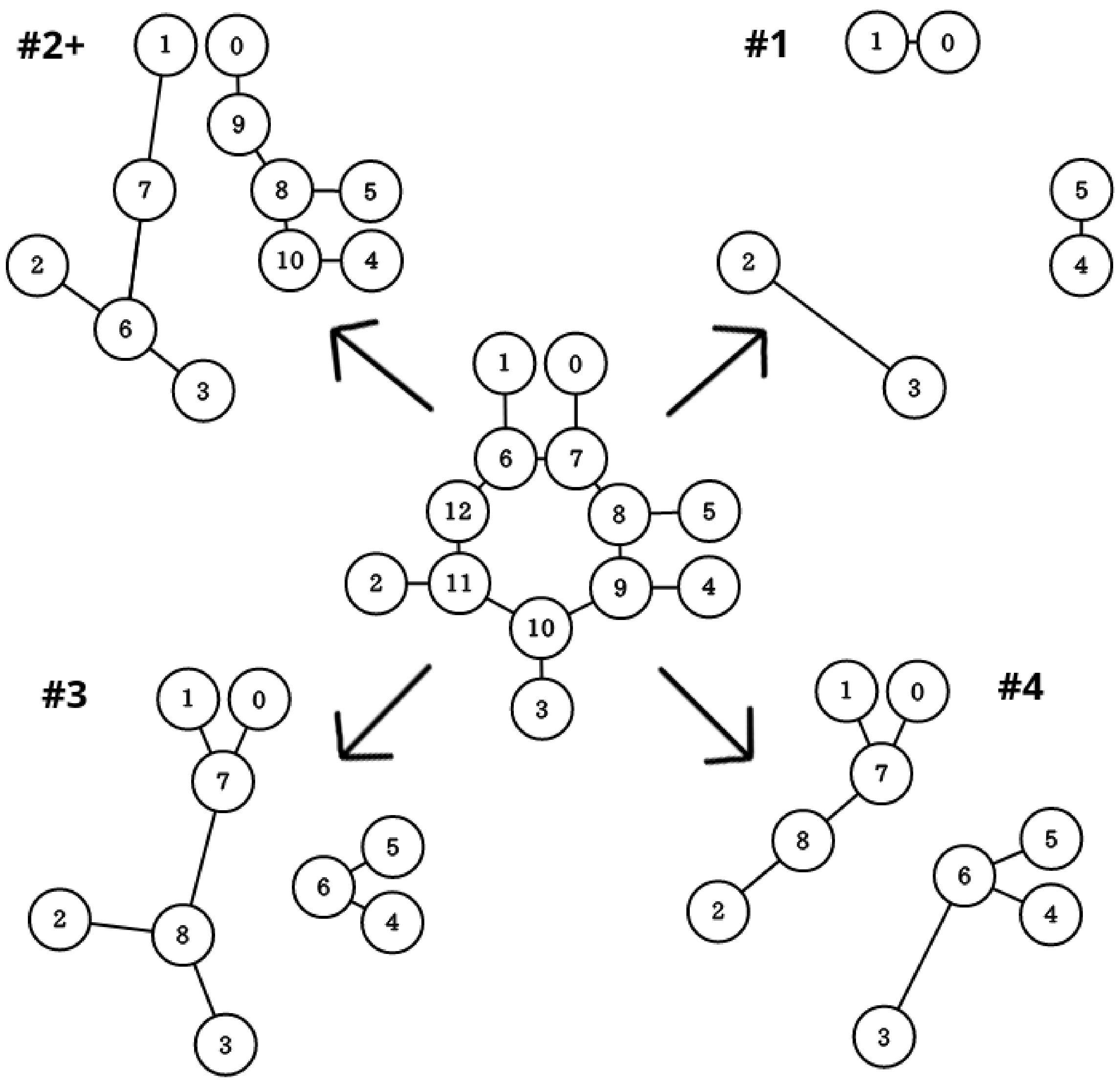

Figure 5.3: The four reductions of a 7-cycle with an edge of deg(2)

5.3.1 Full Seven-cycle

The first obvious target of experiments was the full seven-cycle with no edges of degree two. After running a full test suite with both the faster but limited and slower but thorough version of the permutation function, we have not been able to find any reductions at all. This has provided us with a negative result for most reductions which are normally used and tested on these problems.

The next set of experiments focused on trying out an analogy with the six-cycle, where to find a true general reduction, the authors of other research tried to fix a certain situation at subgraph's immediate border. Our experiments involved fixing such one, two and three outgoing edges to ones of degree three with in all all arrangements that could give us a result (img5.4). We were able to run the once-extended versions of these with the old permutation function, the ones with nine and ten outgoing vertices had to be handled by the more limited, but still generally accurate one that has given us runtimes of around eight minutes per experiment. None of the tests have revealed a single general reduction, continuing the trend set by the original seven-cycle runs.

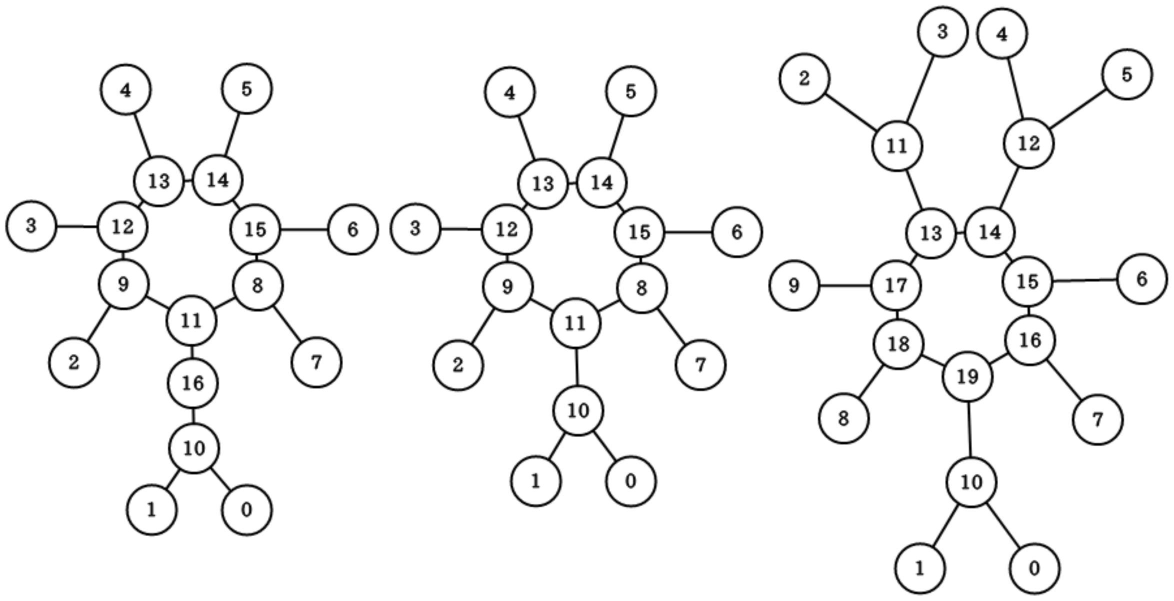

Figure 5.4: An example of some of the extended seven-cycles

Instead of trying to run more brute-force tests by adding even more fixed vertices, we turned our attention to a slightly different construction. This time, we tested a seven-cycle that has been joined with a six cycle on a single edge. Effectively expadanding the graph with one that is reducible and looking for what changes it makes. And it did, as this confiugration has given us a reduction which contracted both cycles down (img5.5) to ten vertices:

{1 -1 -1, 0 -1 -1, 3 -1 -1, 2 -1 -1, 5 -1 -1, 4 -1 -1, 9 -1 -1, 9 -1 -1, 9 -1 -1, 6 7 8}

Returning for the tinal time to the problem of the seven-cycle and seeing that under certain condictions, we were able to obtain reductions, we set up an experiment which looked for partial reductions on the original problem. Using the outputs of our program, we created a script that would parse them and find reductions which could solve certain situations that acted as counterexamples to the otherwise most successfull reduction produced: {1 -1 -1, 0 -1 -1, 3 -1 -1, 2 -1 -1, 7 -1 -1, 7 -1 -1, 7 -1 -1, 4 5 6}. The script found a series of reductions which are included in the appendix. We have not delved into finding what rules governed these and further looking into the matter would be a natural next step forward for anyone interested in reducing the full seven-cycle graph algorithmically.

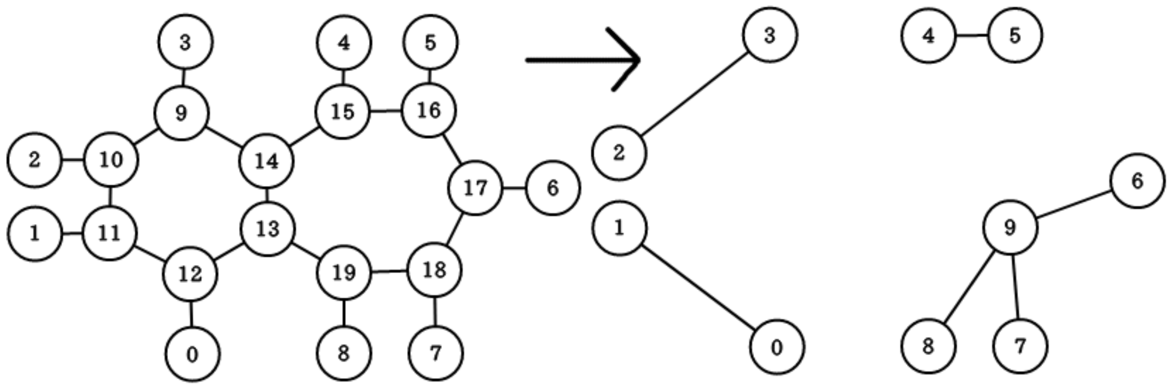

Figure 5.5: The 6-7 cycle reduction

5.3.2 Eight-cycle

The second kind of graph we have tried experimenting on was the eight-cycle. We focused on this graph because our program was capable of handling the necessary workload in a timely manner and because we quickly recognised a reduction inherent to it which simply connected all of the neighboring vertices together (img5.6). Just like in the case of the seven-cycle, we ran a series of tests which first tried to look for general reductions of the full eight-cycle, and later also included the eight-cycle extended to fix one of its border edges into a vertex of degree three. However, none of the experiments provided any positive results. Thus we gained another negative result for a large class of graphs.

5.3.3 Ten-cycle

The third and final kind of graph we tried to extend the analogy to was the ten-cycle. As before, it was within the reach of the program, though this time only in its non-thorough form. This time, the trick with a reduction that connects neighbors did not work thanks to a counter-example which connected the edges into a six-cycle, jumping over a pair of edges that would then create an unsurmountable barrier for the approximations of TSP walks on the original ten-cycle. Having finished this set of tests, we concluded our experiment.

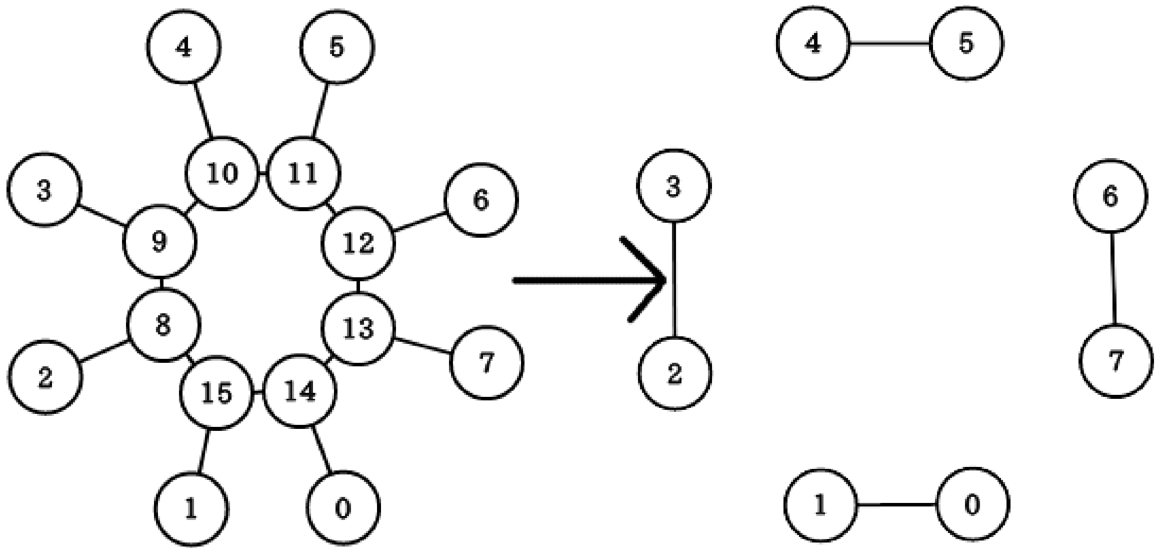

Figure 5.6: The neighbor reduction on an eight-cycle

Summary

The goal of this work was to provide more research into reducible configurations of the TSP problem in subcubic graphs. We were to expand upon the work of researchers who had found reductions for subcubic graphs containing no cycles of length seven or more. Primarily focusing on these and larger instances, we went looking for general reductions of these subgraphs.

To achieve this goal, we first needed to develop and test an appropriate automatic testing suite. This involved both the production of performance critical code as well as scripts which would serve to help with the input of large and detailed graphs. Finally we ahev created a reference set of trees of specific properties, which were used throughout as our building blocks for the reductions.

Once we have done that, we conducted a series of experiments involving the desired graphs. First we conducted a series of tests meant to verify our ouputs and attempt to look for reductions which have been missed by the previous research. We tested both the underlying graphs as well as various modifications of them which provided slightly less general reudctions usable in specific scenarios on the subgraph's boundary. Eventually we found a few reductions in these instances while mostly disproving the existence of such for the general case, for the kind of graphs used as reduction candidates in our program.

Bibliography

[1] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. 1976.

[2] Zdenek Dvorák, Daniel Král, and Bojan Mohar. Graphic TSP in cubic graphs. *CoRR*, abs/1608.07568, 2016.

[3] David Gamarnik, Moshe Lewenstein, and Maxim Sviridenko. An improved upper bound for the tsp in cubic 3-edge-connected graphs. *Operations Research Letters*, 33:467–474, 09 2005.

[4] Jeremy Karp and R. Ravi. A 9/7-approximation algorithm for graphic TSP in cubic bipartite graphs. *CoRR*, abs/1311.3640, 2013.

[5] Marek Karpinski, Michael Lampis, and Richard Schmied. New inapproximability bounds for tsp. *Journal of Computer and System Sciences*, 81(8):1665–1677, 2015.

[6] Marek Karpinski and Richard Schmied. Approximation hardness of graphic TSP on cubic graphs. *CoRR*, abs/1304.6800, 2013.

[7] András Sebö and Jens Vygen. Shorter tours by nicer ears: 7/5-approximation for graphic tsp, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs. *CoRR*, abs/1201.1870, 2012.

[8] Anke van Zuylen. Improved approximations for cubic and cubic bipartite TSP. *CoRR*, abs/1507.07121, 2015.

[9] Mingyu Xiao and H. Nagamochi. An exact algorithm for tsp in degree-3 graphs via circuit procedure and amortization on connectivity structure. 74:713–741, 02 2016.

Appendix A

This work comes with a DVD which contains the source files of the developed program and the two supplementary Python scripts.