Evolutionary Algorithm for the Weighted Connectivity Augmentation Problem

Nikita-Nick Funk

November 2, 2024

4117939

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Ernestine Großmann

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor Prof. Christian Schulz. He assisted and guided me throughout not only this thesis but also my journey as a student, by providing me with many opportunities to work on graph algorithms and algorithm engineering. Without him, this work would not have been possible. Furthermore, I am forever grateful for the unwavering support and guidance given to me by Ernestine Großmann. She always provided me with amazing feedback and ideas to solve the many complex problems that came with this work. Finally, I want to thank my family and friends who always supported me throughout my journey as a student.

Abstract

The Weighted Connectivity Augmentation Problem (WCAP) is the challenge of finding a subset of a given set of links with an associated cost which increase the connectivity of a graph while minimizing cost. This is a fundamental problem in robust network design. This work proposes an evolutionary approach utilizing state-of-the-art heuristic algorithms. By leveraging principles of natural selection, combination, mutation and fitness evaluation heuristic solutions can be refined to produce a new solution of higher quality. The algorithm described in this work implements a classic steady-state evolutionary algorithm using different combination and mutation operations on a selection of solutions computed by heuristic algorithms. Experiments show an improved solution quality of up to 20% depending on the link cost distribution.

Contents

Abstract v

1 Introduction 1
1.1 Motivation 1
1.2 Contributions 2
1.3 Structure 2

2 Fundamentals 3
2.1 General Definitions 3
2.2 Minimum Cuts 3
2.3 Cactus Graphs 4
2.4 Weighted Connectivity Augmentation Problem 4

3 Related Work 5
3.1 Minimum Cuts 5
3.2 Connectivity Augmentation 5
3.2.1 Approximation Algorithms 5
3.2.2 Randomized Algorithms 7

4 Engineering an Evolutionary Approach 9
4.1 Data Structures 9
4.2 High-Level Overview 10
4.3 Creating the initial Population 11
4.4 Selection 12
4.5 Combine Operators 12
4.6 Eviction Strategy 13

5 Experimental Evaluation 15
5.1 Methodology 15
5.2 Dataset 16
5.3 Evaluation 16

6 Discussion **21**
6.1 Conclusion 21
6.2 Future Work 21

Abstract (German) **23**

Bibliography **25**

CHAPTER 1

Introduction

1.1 Motivation

Graphs are used to model many real-world systems such as infrastructure, social or communication networks. The robustness of a given graph or network can be measured by analysing its connectivity. Ensuring robust and fail-safe systems is of particular importance in technological systems [15]. Using the example of a power grid by Freitas et al. [15] if a power-line fails alternative routes need to be used which increases stress on said lines and thus the possibility of them failing as well. To guarantee a robust and fail-safe network any given graph or network has to be sufficiently well-connected. The problem of increasing the connectivity of any given graph while minimizing some defined cost metric is known as the connectivity augmentation problem or the survivable network problem.

The weighted connectivity augmentation problem is known to be NP-hard. The decision problem of whether an augmentation of a given weight exists was shown by Eswaran and Tarjan to be NP-complete [11]. Furthermore, it has been shown by Frederickson and Ja'Ja' that for a tree with link weights being 1 or 2 this decision is NP-complete as well [14]. Moreover, the weighted connectivity augmentation problem as well as the weighted tree augmentation problem are APX-hard [22]. No polynomial time approximation with an approximation factor arbitrarily close to 1 has been found yet, but despite this, the connectivity augmentation problem has been discussed plenty. Recently, a novel heuristic approach based on minimum spanning trees which quickly yields high-quality solutions has been published by Fonseca et al. [12]. Furthermore, they also proposed an efficient ILP formulation to find optimal solutions and they gave a first implementation of better-than-2 approximations [5, 27, 30, 31].

1.2 Contributions

This thesis aims to contribute an evolutionary approach to improve solution quality for the weighted connectivity augmentation problem based on previous work done by Fonseca et al. [12]. The proposed algorithm heavily utilizes the minimum spanning tree heuristics, local search approach and flow refinements detailed by Fonseca et al. [12].

In each iteration of the algorithm two individuals are selected and one of four combine operations is performed. One performs the minimum spanning tree heuristic on only the links present in the individuals to refine the generated solutions. Another combines both individuals and performs the flow refinement [12] on this new solution. Furthermore, the third combine operation combines the individuals and greedily adds links from the combined solution satisfying a given heuristic. Lastly, another combine operation computes the cut of the selected individuals and uses the minimum spanning tree heuristic algorithm and local search to complete the cut to a feasible solution to the problem. At the end of each iteration the newly generated solution, called an offspring, is inserted into the population by evicting another individual. The individual which shares the largest cut with the offspring and has a worse solution quality, called fitness, is chosen to be evicted. If no such individual is found a new offspring with more lenient local search parameters is computed until an individual can be evicted.

The evolutionary algorithm is compared to heuristic algorithms which are run multiple times with noise on the link weights to induce tie-breaking. Real-world and generated graphs were used to perform these experiments. All algorithms were run with the same time constraint and thus only solution quality is compared. Furthermore, different link weight distributions were tested. These experiments show the evolutionary algorithm outperforming the pure heuristic algorithms with added noise.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2 all necessary fundamentals and definitions are described. Chapter 3 details related work on the weighted connectivity augmentation problem as well as related problems. The evolutionary approach is discussed in Chapter 4 and subsequently experimentally evaluated in Chapter 5. This work is concluded by a discussion regarding the experimental results as well as possible future work.

CHAPTER 2

Fundamentals

Preliminaries needed for the weighted connectivity augmentation problem are described in the following chapter. As this work is based on previously conducted research by Fonseca et al. [12] we borrow definitions to related problems such as the minimum cut problem and cactus graph representation of minimum cuts.

2.1 General Definitions

Consider an undirected graph $G = (V, E)$ where V is the set of vertices and $E \subseteq \binom{V}{2}$ is the set of edges connecting pairs of vertices. The number of vertices is denoted as n and the number of edges is denoted as m. A graph G is *connected* if there exists a path between any pair of vertices. A graph G is called *k-edge-connected* if any two vertices of G can be joined by k edge disjoint paths. Intuitively, this means, that k arbitrary edges of G can be removed without disconnecting the graph. The greatest integer k for which G is k-edge-connected is called the *edge-connectivity* $\lambda(G)$ of G. Notice that if $\lambda(G) = 0$ then G is disconnected. In this work we only consider connected graphs, i.e. $\lambda(G) > 0$.

2.2 Minimum Cuts

A *partition* is a partition of the vertex set of a graph G into disjoint non-empty sets. A *cut* C is a partition of the vertex set into two disjoint subsets. This is also called a *bipartition*. Any cut C can be represented as one of its two vertex sets. The complementary vertex set is always only implied. The sum of the edge weights of a cut is called the *size* or *weight* of the cut. A cut is called *minimum* if no other cut of smaller size or weight exists. We call C_G the set of all minimum cuts of a graph G.

2.3 Cactus Graphs

A *cactus graph* is a connected graph, where any two cycles share at most one vertex. *Cycle edges* are edges that lie on a cycle and *tree edges* are edges that do not lie on any cycle. The *cactus graph representation* of the set of minimum cuts of a graph $G = (V, E)$ is a cactus graph $C = (V_C, E_C)$ paired with a function $\Pi : V \to V_C$ and its inverse $\Pi^{-1} : V_C \to 2^V$ which is defined as $v \mapsto \{u \in V : \Pi(u) = v\}$ [12]. This definition maps each minimum cut of C to a corresponding minimum cut in G. For further details and construction we refer the reader to [10].

2.4 Weighted Connectivity Augmentation Problem

Consider a k-connected graph $G = (V, E)$, a set of links $L \subseteq \binom{V}{2}$ and a cost function $c : L \to R_{\geq 0}$. Furthermore, $E \cap L = \emptyset$ and $E \cup L$ is always a solution to the following problem. The *weighted connectivity augmentation problem* asks us to increase the edge connectivity of G to $k + 1$ by adding the cheapest subset of links $S \subseteq L$. A cut $c \in C_G$ is *covered* by a link $l \in L$ if the size or weight of the cut c is increased in $G' = (V, E \cup \{l\})$. We define $G_L = (V, L)$ as the *link graph* that contains all links but not the edges of G. If the graph is disconnected, this problem coincides with the minimum spanning tree problem on its components. This work is built upon the assumption that the input graph is always connected.

CHAPTER 3

Related Work

Prior work done on the weighted connectivity augmentation problem is presented in this chapter. Furthermore, the state-of-the-art algorithm for minimum cuts and the cactus graph representation thereof are discussed, as this is a crucial part of solving the weighted connectivity problem.

3.1 Minimum Cuts

A linear time approach exists for computing near-minimum cuts based on cluster contraction using label propagation and contraction heuristics [19]. An efficient way to compute the cactus graph representation of all minimum cuts was proposed by Nagamochi, Nakao and Ibaraki [25]. Their approach computes the minimum cuts between two vertices s and t by running a s-t-flow algorithm and partitioning the resulting network into smaller networks on which the cactus representation can more easily be computed. Afterward, all cactus representations are combined. For further details, we refer the reader to [25]. Currently, the state-of-the-art algorithm is *VieCut* by Henzinger, Noe, Schulz and Strash [18, 20]. They adapt and employ reduction strategies by Nagamochi et al. [25, 26] and Padberg et al. [28]. An optimized version of the algorithm by Nagamochi et al. [25] is run on the resulting kernel. A detailed description of the algorithm can be found in [20].

3.2 Connectivity Augmentation

3.2.1 Approximation Algorithms

Since the weighted connectivity problem is APX-hard [22] several different approximation approaches have been proposed. An early approach by Frederickson and Ja'Ja' for the bridge connectivity augmentation, the case where a graph is 1-connected but not 2-connected, utilizes a greedy approach that connects leaves in a tree structure [14]. The problem is transformed into a directed graph problem where minimum cost arborescences are used. Later, Watanabe et al. proposed 4 approximation algorithms for the weighted connectivity problem called *FSA*, *FSM*, *SMC* and *HBD* [32]. *FSA* is based upon the work done by Frederickson and Ja'Ja' [14] and uses minimum-cost arborescences. *FSM* utilizes maximum-cost matchings to connect graph components. *SMC* greedily selects minimum-cost edges and *HBD* is a hybrid approach combining *FSA* and *FSM*. Watanabe et al. showed that *FSA*, *FSM* and *HBD* guarantee an approximation ratio of at most 2 for the bridge-connectivity augmentation, but this bound does not hold for $k > 1$ [32].

In 1993 Khuller and Thurimella proposed a 2-approximation for any $k > 0$ by first transforming the undirected graph into a directed graph [21]. Each undirected edge $e = (u, v)$ is replaced by directed edges in both directions and the directed version is solved using matroid intersection. Furthermore, the directed problem can be solved in polynomial time using minimum-cost flows [13]. A polynomially bound linear program for the cactus augmentation problem can also be used to solve this case [7].

Work on approximation algorithms with an approximation factor better than 2 has been done recently. Byrka, Grandoni and Ameli proposed a 1,91-approximation algorithm for the unweighted connectivity augmentation problem by reducing it to a Steiner Tree problem [4, 6]. After reducing an instance to a Steiner Tree problem instance they utilize an iterative randomized rounding approach and an adjusted linear program relaxation for said problem to achieve this approximation bound. A 1,393-approximation algorithm was proposed by Cecchetto et al. [7] for the tree augmentation problem, the case where $k = 1$ and therefore essentially augmenting a tree, as well as the unweighted connectivity augmentation problem. A greedy approach with an approximation factor of $(1 + ln(2) + \epsilon)$ has been proposed for the weighted tree augmentation problem by Traub and Zenklusen [30]. They transferred this approach to the weighted connectivity augmentation problem and proposed a $(1.5 + \epsilon)$-approximation algorithm [31]. A first implementation and experimental evaluation of both the $(1 + ln(2) + \epsilon)$ as well as the $(1.5 + \epsilon)$ approximation algorithms has been given by Fonseca et al. [12]. In addition to these implementations, they propose a new exact solver using an integer linear program, as well as new heuristic approaches which aim to quickly deliver high quality solutions. The first approach they proposed is a greedy heuristic called *GWC* which adds links based on cost-effectiveness, which considers both the link cost as well as the number of cuts this link covers. Their second approach *MSTConnect* starts with an initial feasible solution to the weighted connectivity augmentation problem by computing a minimum spanning tree and removing unnecessary links using a flow-based approach. Furthermore, they employ a local search which can improve any given solution by replacing sets of links with cheaper ones. Unlike the previously mentioned approximation algorithms, these heuristic algorithms cannot guarantee an approximation factor. Their approaches are faster and yield higher quality solutions than the previous state-of-the-art algorithms.

3.2.2 Randomized Algorithms

A randomized Monte Carlo algorithm that solves the weighted connectivity problem in $\tilde{\mathcal{O}}(m + n^{3/2})$ has been proposed by Cen, Li and Panigrahi [8] which improves the previous best time complexity of $\tilde{\mathcal{O}}(n^2)$ established by Benczúr and Karger [3]. They showed that the weighted connectivity problem can be solved by using polylog(n) maximum flow computations. The current state of the art is an $\tilde{\mathcal{O}}(m)$-time algorithm by Cen, Li and Panigrahi [9].

CHAPTER 4

Engineering an Evolutionary Approach

An evolutionary algorithm draws heavy inspiration from mechanisms of biological evolution, such as selection, recombination, mutation and survival of the fittest. An initial population of individuals (in our case sets of links that improve the edge-connectivity of the input graph) is acted upon by the evolutionary algorithm through selection and recombination. In each iteration of the algorithm, some individuals are selected based on their fitness (in our case the summed cost of links in the set) and combined to form an improved offspring [16]. We do not need a penalty function as each created offspring is a feasible solution to the weighted connectivity problem. This will become clear in the following sections. After the successful creation of an offspring, an individual of the population is chosen by an eviction rule to be replaced by the new offspring. This eviction rule has to consider the fitness of an individual and how similar the individual is to the offspring, as we want to keep the diversity of the population high. This is important to avoid possible premature convergence of the algorithm, i.e. to avoid getting stuck in a local optima [1]. We implement a *steady-state* evolutionary algorithm meaning, we only generate one offspring in each iteration of the evolutionary algorithm. We borrow a depiction of a typical structure of a steady-state evolutionary algorithm from Sanders and Schulz from their work on distributed evolutionary graph partition [29], which is described by Algorithm 1.

The work in the following chapter is based on previous work done by Fonseca et al. [12]. A brief description of the data structures used by Fonseca et al. [12] is given first. Afterward, a high-level view of the evolutionary algorithm and its parts is introduced. The following sections detail all parts of the evolutionary algorithm.

4.1 Data Structures

It suffices to do computations on the cactus graph C, as all minimum cuts of a given graph G can be represented as a potentially significantly smaller cactus [10]. To compute the cactus graph *VieCut* [18, 20] is used. The original graph G is stored using an adjacency

Algorithm 1 A classic general steady-state evolutionary algorithm [29]

```
procedure STEADY-STATE-EA
    create initial population $P$
    while stopping criterion not fulfilled
        select parents $p_1, p_2$ from $P$
        combine $p_1$ with $p_2$ to create offspring $o$
        mutate offspring $o$
        evict individual in population using $o$
    return the fittest individual that occurred
```

list. A function $\Pi : V(G) \to V(C)$ mapping vertices of G to vertices of C is defined and modeled by an array using vertex IDs as indices. For every link $l = (u, v) \in L$ the function $\Pi(l) := (\Pi(u), \Pi(v))$ maps the link l to its corresponding link $l_C := \Pi(l)$ in the cactus graph C. This can lead to parallel links, i.e. $g, h \in L, g \neq h$ with $\Pi(g) = \Pi(h)$. This set is optimized by only keeping minimum-cost links for each vertex pair in C. The link set L_C is stored using an adjacency matrix. Fonseca et al. also employ a dynamic cactus data structure which can be updated efficiently [12]. This data structure extends and modifies the approach by Henzinger, Noe and Schulz [17], which utilizes a union find data structure. For implementation details, we refer the reader to [24].

In addition to the data structures used by Fonseca et al. [12] we keep one copy of the original link set with the original link costs. During the evolutionary algorithm, a small amount of noise is regularly added to the link costs to induce tie-breaking when running the heuristic algorithms and combine operators. This copy is used to remove the added noise and evaluate the produced offspring on the original link costs.

4.2 High-Level Overview

An overview of our evolutionary algorithm can be found in Algorithm 2. All parts of the algorithm, such as how selection is performed or what combine operators are used, are detailed in the following sections. First, we create an initial population. Afterward, the main evolutionary loop starts. At the start of each iteration, two individuals from the population are selected based on the tournament selection rule [23]. Then, noise is added to the link costs. This noise is a random number between 0 and $1/100$ of the smallest occurring link cost. This is done to induce tie-braking wherever possible. One of four combine operators is chosen and the previously chosen individuals are combined to produce an offspring. Afterward, the added noise is removed and one individual from the population is swapped with the new offspring based on an eviction rule. This cycle is repeated until a time limit is reached, after which the fittest individual of the population is returned.

Algorithm 2 Overview of our evolutionary algorithm for WCAP

```
procedure WCAPEvo(Graph $G$, Cactus $C$)
    create initial population $P$
    while time left
        select parents $p_1, p_2$ from $P$
        add noise to links of $C$
        choose random combine operator $comb$
        combine $p_1$ with $p_2$ using $comb$
        remove noise from links of $C$
        evict individual in population using $o$
    return the fittest individual that occurred
```

4.3 Creating the initial Population

In the initial creation of our population, 63 individuals are created by using 3 different heuristic algorithms. The first run of these algorithms is computed without any noise on the link costs. All following individuals are subjected to the previously discussed noise to induce tie-braking. A population size of around 63 individuals seems to strike a good balance between having a large enough and diverse population while also being computed quickly. Early experiments with a population size of 100, 300 and 500 did not yield any significant improvements regarding final solution quality. The noise on the link costs regularly changes during the initial population fill. After the initial population is filled all noise is removed. The algorithms used are as follows:

MSTConnect and Local Search. We use *MSTConnect* and the local search approach by Fonseca et al. [12] extensively. First, a minimum-spanning tree L_{MST} (or a minimum-spanning forest if the link set is not complete) is calculated on the cactus link graph C_L. The links in L_{MST} are sorted by weight in descending order. For each link in $l = (u, v) \in L_{MST}$ a u-v-flow is computed to check if this link can be removed from the solution, i.e. *dropped*. This can be done in linear time. A proof is given by Fonseca et al. [12]. Afterward, a *local search* [12] is performed on the solution, which aims to replace sets of links with cheaper ones. The number of links within a swap is limited by a parameter k, the *local search depth*, which in our case is set to 3 during the initial population fill. Experiments done by Möller [24] show a local search depth of 3 providing significant improvements. Depth-limits of 4 and 5 seem to further find small improvements, while depth-limits larger than that do not yield significant improvements. Therefore, a local search depth of 3 strikes a good balance between possible improvements and running time. Sets of links are swapped until no more feasible swaps can be found, after which the local search terminates. Feasible swaps are found by using an adapted depth-first search. The depth for this adapted depth-first search is also limited by the local search depth parameter k. For a detailed description of the local search, we refer the reader to [12, 24]. We modified the local search to allow us to set a time limit specifically on the local search. This is done to avoid hitting the global time limit during the initial population fill and to ensure that the algorithm starts the main evolutionary loop. During the creation of the initial population we aim to quickly compute feasible solutions by setting a very strict time limit. Let t be the global time limit for the evolutionary algorithm. The time limit for the local search during the initial population fill is set to $max(1, t/1000)$.

MSTConnect without dropping links. Secondly, we utilize *MSTConnect*, but without dropping any links and without using local search. This approach is fast and may include links that could have been dropped. This leads to a slightly more diverse population which may induce more mutations during the combine stage.

FastGreedy. Lastly, we utilize an approach that does not rely on minimum-spanning trees. We do not directly use *GWC* by Fonseca et al. [12] since *MSTConnect* is considerably faster. *GWC* could be made feasible by enforcing a time limit or finding reductions for the link set on which the greedy algorithm is run. Instead of using the weight coverage heuristic employed by *GWC* we just sort the link set by their cost in ascending order and add links to the solution until all cuts are covered. This is generally very fast and provides some diversity in our population, but produces low-quality solutions [24]. For graphs with a large number of links, this approach still takes a considerable amount of time, which often leads to the algorithm not starting the main evolutionary loop on said graphs. Therefore, a time limit has to be enforced. If this time limit is reached the currently unfeasible solution is made feasible using *MSTConnect* without dropping links or any local search.

4.4 Selection

In each iteration of the evolutionary algorithm, we have to select individuals, in our case 2 individuals p_1 and p_2, to perform combine operations on. We utilize the selection process by Miller and Goldberg [23]. To select an individual from our population a *tournament* is held between two randomly selected individuals r_1 and r_2. The fitter of the two individuals is selected. Another tournament between two new randomly selected individuals is held to select another individual.

4.5 Combine Operators

After two individuals have been selected noise is added to the link costs before choosing a combine operation. One of four combine operators is chosen randomly.

MST-Combine. The idea of this approach is to combine the selected individual into one new solution and remove duplicates in the process. A minimum-spanning tree is then computed on only the links in the combined solution. This will lead to unnecessary links being dropped. As this is a quite simple approach it is very fast.

Drop-Combine. This combine operation relies on the flow approach by Fonseca et al. [12] to drop links and improve our solution. Again, the selected individuals are combined into one large solution. This link set is then sorted in ascending order using the weight coverage heuristic from *GWC* [12], i.e. for a link $l \in L$ we divide the cost of the link $c(l)$ by the number of cuts covered by said link $a_l := \{c \in C_G : c \text{ is covered by } l\}$. If $a_l = 0$ the link l is not considered. The flow approach by Fonseca et al. [12] is then applied to find links that can be removed from the new solution.

Recombine. Again, the selected individuals are combined into one large solution. Analogously to *GWC* we greedily pick links l from the combined solution which minimize $c(l)/a_l$ until all cuts are covered.

Intersect-Combine. Lastly, the idea of this operation is not to combine the selected individuals, but to compute the intersection. This intersection is completed into a feasible solution by running *MSTConnect* with the flow approach. This new solution is then subjected to a local search. Again, this local search is given a time limit. This time limit is less strict than the time limit given in the initial population fill. When first creating the initial population we want to quickly compute feasible solutions, but now we allow more time for the local search to compute solutions of higher quality. Let t be the global time limit again. The time limit for the local search during the *Intersect-Combine* is set to $max(1, t/10)$.

4.6 Eviction Strategy

Before the eviction process is started, all noise is removed from the link costs. We want to replace one individual in the population with our new offspring. At the same time, we want to keep the population diverse as well as only replace individuals who have worse fitness than our offspring. Our eviction strategy is based on a round system. First, we try to find the individual in the population with the largest intersection with our new offspring. Preferably, we want to swap these to keep diversity high. We do this only if the fitness of our offspring exceeds that of the chosen individual. Should this not be the case a round-based approach is started. In each round, a random individual in our population is chosen and a new offspring is generated by adding noise to the link costs and subsequently using *MSTConnect* and local search. This time no special time limit, other than the global time limit, is used for the local search. Recall from section 4.3 that the local search can terminate even if no time limit is given, by stopping if no more feasible swaps can be found. This is done to increase the probability of the local search to compute a solution of high solution quality. Afterward, the noise is again removed and if the fitness of the new offspring exceeds that of the randomly chosen individual to be evicted, we swap them and the eviction process is finished. If this is not the case we proceed to the next round. Every time we proceed we increase the local search depth by one. Recall from section 4.3 that the local search depth determines the limit of links in a swap. The initial local search depth is set to 3. If we are unable to evict any individual after 4 rounds we stop the eviction process without changing the population. An overview of this process is depicted in Algorithm 3.

Algorithm 3 Overview of the eviction process

```
procedure EVICT(offspring $o$, population $P$)
    find individual $p \in P$ with largest intersection with $o$
    $round \leftarrow 0$
    $depth \leftarrow 3$
    do
        if fitness of $p <$ fitness of $o$ then
            evict $p$ and insert $o$ into $P$
            return
        else
            pick new $p \in P$ randomly
            add noise to link costs
            generate new offspring $o$ with MSTConnect
            run local search with depth limit $depth$ on $o$
            remove noise from link costs
            if fitness of $p <$ fitness of $o$ then
                evict $p$ and insert $o$ into $P$
                return
            else
                $round \leftarrow round + 1$
                $depth \leftarrow depth + 1$
    while $round < 4$
    return
```

CHAPTER 5

Experimental Evaluation

We now discuss the experimental evaluation of the previously detailed evolutionary algorithm. First, the methodology as well as some information on the test instances is given. Afterward, the evolutionary algorithm is compared to the pure heuristic algorithms in terms of solution quality.

5.1 Methodology

The experiments are performed on a computer with an Intel(R) Xeon(R) Silver 4216 CPU with 32 cores running at 2.10GHz and 93GB of main memory running Linux. The C++ code is compiled using *gcc 9.4.0* with optimization level *O3*. The running time is limited to 150 minutes.

Each instance is run twice using two different link cost distributions. We denote the link cost distributions as w_2 and w_9, respectively. Link costs in w_2 are given by the set $\{0.5, 1\}$ and link costs in w_9 are given by the set $\{\frac{i}{10} : i = 1..10\}$. These are picked uniformly at random. We compare our evolutionary algorithm against the simple greedy approach (called *greedy* in the following tables), a simple minimum-spanning tree on the cactus graph (called *mst* in the following tables), and *MSTConnect* with the flow refinement and *local search* [12] (called *mst-ls-flow* in the following tables). It is important to note that, just like in the eviction process detailed in section 4.6, the local search is not time-limited and allowed to terminate naturally. The heuristics are run for the same amount of time as the evolutionary algorithm. Before each run of the heuristic algorithm noise is added to the link costs. The solution is evaluated on the original link costs and the best found solution is returned once the time limit is reached. This approach allows us to see if our evolutionary algorithm can find better solutions given the same time constraint. The tables also include the time, when the evolutionary found the best solution. This is given in seconds.

Table 5.1: Properties of Instances G and their Corresponding Cactus Graph C

Graph	$\lvert V(C)\rvert$	$\lvert E(C)\rvert$	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	**Description**
coPapersCiteseer	6 372	6 371	434 102	16 036 720	Social network
t60k	1 136	1 332	60 005	89 440	Sparse matrix
vibrobox	625	624	12 328	165 250	Sparse matrix
delaunay_n14	181	180	16 384	49 122	Delaunay graph
email	156	155	1 133	5 451	Social network
jazz	6	5	1 133	5 451	Social network
star-5000	5 000	4 999			Generated graph
star-1000	1 000	999			Generated graph
cycle-1000	1 000	1 000			Generated graph

5.2 Dataset

Two types of graphs are used in the evaluation: generated cycle and star graphs as well as real-world instances. Cycles and stars are edge cases of cactus graphs with a number of minimum cuts between $O(|V_C|)$ and $O(|V_C|^2)$ [12]. Fonseca et al. recognized, that many real-world instances have a low number of distinct minimum cuts, which leads to very small cactus graphs [12]. Therefore, we follow the selection of graphs by Fonseca et al. [12]. They picked connected graphs with non-trivial cactus graph representation, which have at least 100 edges and at most 40 000 vertices, from the *10th DIMACS Implementation Challenge* [2]. All cactus representations are computed by using *VieCut* [20]. These are computed once for every instance and done in advance. All instances, including all instance properties, are listed in Table 5.1.

5.3 Evaluation

Table 5.2 shows the fitness values (summed link costs of the best found solution) using the $w2$ link cost distribution. It is immediately obvious, that the *jazz* graph is small enough for the greedy approach, as well as *MSTConnect* with flow refinements and local search to find an optimal solution. Therefore, our evolutionary algorithm also finds this solution instantaneously. It is clear, that *MSTConnect* is the best heuristic algorithm. These observations are in line with the observations by Fonseca et al. [12]. Our evolutionary approach is able to refine the solution. The improvement is consistently around 20%. This is easily apparent for large graphs, such as *coPapersCiteseer*. It is important to note that the evolutionary algorithm finds the best solution rather quickly and is unable to improve upon it until the time limit is reached. Recall, that the time limit is set to 150 minutes which is equivalent to 9 000 seconds. This may imply that more aggressive mutation and combination approaches might be necessary to further improve the solution. The greedy approach

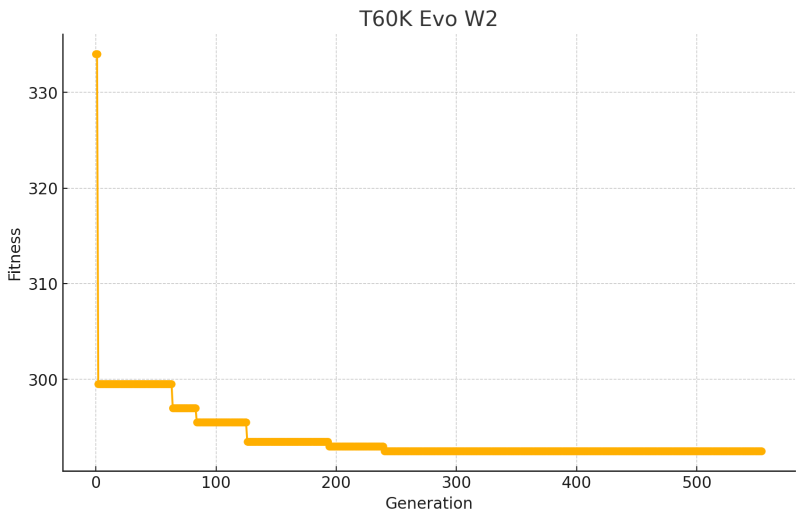
Table 5.2: Augmentation weights using $w2$ Distribution

graph	evo	greedy	full-mst	mst-ls-flow	time-found [s]
jazz	1,5	1,5	3,0	1,5	$<$ 0,01
email	38,0	97,5	77,5	44,0	116
vibrobox	158,5	615,0	312,0	197,0	5 564
t60k	292,5	1 317,0	567,5	369,0	1 378
coPapersCiteseer	1 598,5	9 997,0	3 185,5	2 075,5	1 582
coAuthorsCiteseer	8 552,5	15 160,5	15 160,5	9 979	5
delaunay_n14	45,0	117,0	90,0	56,5	228
delaunay_n20	3 490	5 869,5	5 869,5	4 104	2 792
cycle-1000	265,5	1 191,5	499,5	322,5	1 132
star-1000	256,5	1 105,5	499,0	322,5	3 036
star-5000	1 370,5	8 083,0	2 499,0	1 727,5	3 345

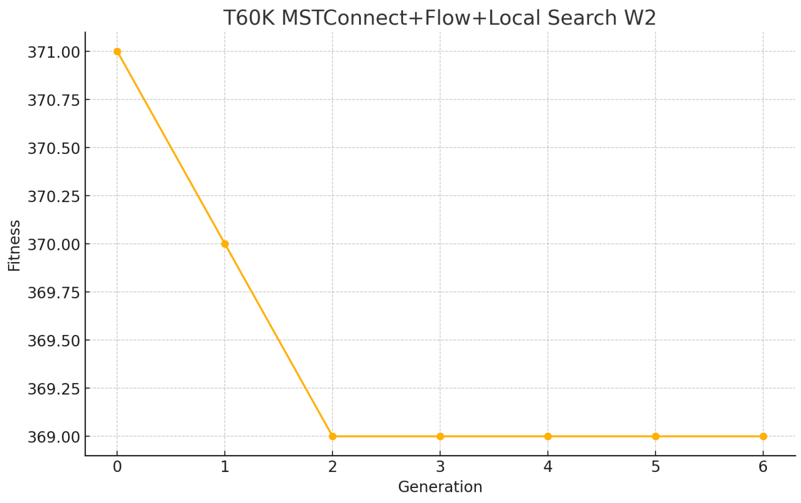
which adds the lowest cost links until all cuts are covered expectedly performs the worst.

Figure 5.1 shows plots for the progression of fitness values for the evolutionary algorithm, the greedy heuristic and *MSTConnect* on the *T60K* graph (see Table 5.1 for more information on this instance). The greedy approach in Subfigure 5.1c seems to react to the added noise, but quickly converges to the final fitness value. Since the local search is not time limited *MSTConnect* does not produce many solutions as can be seen in Subfigure 5.1b. One minor jump in solution quality can be observed. The evolutionary algorithm quickly finds a large improvement in the beginning. A steady improvement in solution quality follows until the algorithm converges. Here, it is easy to see, that for most of the time given to the evolutionary algorithm, no improvement can be observed.

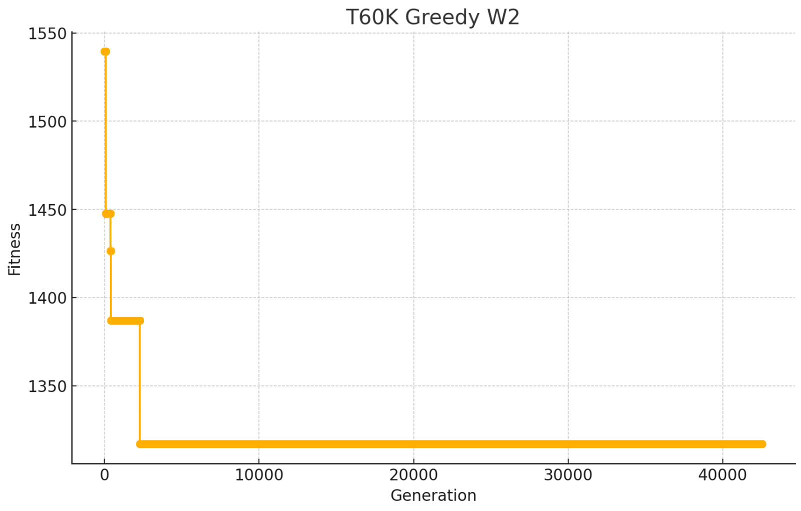
When using the $w9$ distribution, as can be seen in Table 5.3, the evolutionary algorithm struggles to significantly improve the solution when compared to *MSTConnect*. While better solutions are found, the margins are much smaller compared to when using the $w2$ distribution. This is especially clear for graphs with small cactus representations. A more significant improvement in solution quality can be seen for graphs with larger cactus representations, *coPapersCiteseer* and *star-5000* being good examples. For these graphs, the evolutionary algorithm is able to find solutions that improve the solution quality by around 10%. It is again possible to observe, that the evolutionary algorithm quickly converges and is unable to improve the solution further for most of the given time.



(a) T60K Evo $w2$



(b) T60K MSTConnect+Flow+Local Search $w2$



(c) T60K Greedy $w2$

Figure 5.1: Comparison of Fitness Progression for the T60K Graph

Table 5.3: Augmentation weights using $w9$ Distribution

graph	evo	greedy	mst	mst-ls-flow	time-found [s]
jazz	1,33	1,67	2,22	1,33	$< 0{,}01$
email	8,44	21,22	17,22	8,56	4
vibrobox	35,22	132,56	69,33	37,11	8 996
t60k	64,78	291,44	126,11	68,33	2 312
coPapersCiteseer	350,00	2 310,22	707,89	385,22	6 773
coAuthorsCiteseer	1 896,67	3 369	3 369	1 938,78	9
delaunay_n14	10,00	27,22	20,00	10,33	389
delaunay_n20	774,56	1 304,33	1 304,33	777,11	2 772
cycle-1000	59,11	269,22	111,00	59,78	727
star-1000	57,00	247,44	110,89	59,89	4 835
star-5000	290,67	1 760,56	555,33	316,44	4 368

CHAPTER 6

Discussion

6.1 Conclusion

A lot of research and work has been done recently on heuristic and approximation algorithms for the weighted connectivity augmentation problem. However, little to no research has been done on an evolutionary approach to this problem. Previously done work on fast heuristic algorithms that deliver high-quality solutions serve as a strong base to develop promising evolutionary algorithms to solve the weighted connectivity problem. The contribution of this thesis aims to provide a starting point to further develop and refine evolutionary approaches and tailor them specifically to the weighted connectivity augmentation problem. By using simple combine operations measurable improvements can be observed when comparing the evolutionary approach to the heuristic algorithms. This improvement is currently highly dependent on the distribution of link costs.

6.2 Future Work

This thesis proposes a first, simple evolutionary approach to solve the weighted connectivity augmentation problem. Graphs with a large link set pose a great challenge. This is especially noticeable when running greedy heuristics. One way to improve this is to perform some preprocessing on the link set by finding reduction rules. Currently, no reduction rules have been established or evaluated. Another way of improving this algorithm is to find more aggressive mutations. This may be useful if the evolutionary algorithm is stuck in a local optima. One way to achieve this might be to forcefully insert new links into a solution. The questions of how many and which links to choose for mutation have to be looked into.

Moreover, parallel algorithms are not covered in this work. A parallel approach would significantly speed up the initial creation of the population, which by itself would mean, that the evolutionary algorithm spends more time in the main evolutionary loop. Furthermore, multiple offsprings could be generated in parallel, further speeding up the algorithm. Currently, the heuristics used in this thesis are also not yet parallelized.

Finally, this work only focuses on the weighted connectivity augmentation problem and not on related problems, such as the survivable network design problem. There, the goal is not to increase edge-connectivity, but vertex-connectivity. These problems may have overlapping features and assumptions which can be made.

Zusammenfassung

Das Problem der gewichteten Konnektivitätserhöhung (weighted connectivity augmentation problem, WCAP) ist die Forderung, eine Teilmenge einer gegebenen Menge von Verbindungen mit entsprechenden Kosten zu finden, welche die Konnektivität eines Graphen erhöht und gleichzeitig die Kosten minimiert. Dies ist ein grundlegendes Problem beim Entwurf robuster Netzwerke. In dieser Arbeit wird ein evolutionärer Ansatz vorgestellt, welcher aktuelle heuristische Algorithmen nutzt. Durch die Anwendung von Prinzipien der natürlichen Selektion, Kombination, Mutation und Fitnessbewertung können heuristische Lösungen optimiert werden, um eine neue Lösung von höherer Qualität zu erzeugen. Der in dieser Arbeit beschriebene evolutionäre Algorithmus implementiert einen klassischen, kontinuierlichen (steady-state) Evolutionsalgorithmus, welche verschiedene Kombinations- und Mutationsoperationen auf eine Auswahl von Lösungen anwendet, die von heuristischen Algorithmen berechnet wurden. Experimente zeigen eine verbesserte Lösungsqualität von bis zu 20%, abhängig von der Verteilung der Verbindungskosten.

Bibliography

[1] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Inc., USA, 1996.

[2] David Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. *Benchmarking for Graph Clustering and Partitioning*, pages 1–11. 01 2017.

[3] András Benczúr. Augmenting undirected edge connectivity in Õ(n2) time. *Journal of Algorithms*, 37, 10 2000.

[4] Jaroslaw Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: a reduction to steiner tree. *CoRR*, abs/1911.02259, 2019.

[5] Jarosław Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: A reduction to steiner tree. *SIAM Journal on Computing*, 52(3):718–739, 2023.

[6] Jarosław Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: A reduction to steiner tree. *SIAM Journal on Computing*, 52(3):718–739, 2023.

[7] Federica Cecchetto, Vera Traub, and Rico Zenklusen. Bridging the gap between tree and connectivity augmentation: unified and stronger approaches. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 370–383, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. *Augmenting Edge Connectivity via Isolating Cuts*, pages 3237–3252.

[9] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Edge connectivity augmentation in near-linear time. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 137–150, New York, NY, USA, 2022. Association for Computing Machinery.

[10] E. Dinic, Alexander Karzanov, and M. Lomonosov. The system of minimum edge cuts in a graph. *In book: Issledovaniya po Diskretnoĭ Optimizatsii (Engl. title: Studies in Discrete Optimizations), A.A. Fridman, ed., Nauka, Moscow, 290-306, in Russian,*, 01 1976.

[11] Kapali P. Eswaran and R. Endre Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[12] Marcelo Fonseca Faraj, Ernestine Großmann, Felix Joos, Thomas Moller, and Christian Schulz. Engineering weighted connectivity augmentation algorithms, 2024.

[13] András Frank and Éva Tardos. An application of submodular flows. *Linear Algebra and its Applications*, 114-115:329–348, 1989. Special Issue Dedicated to Alan J. Hoffman.

[14] Greg N. Frederickson and Joseph Ja'Ja'. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

[15] Scott Freitas, Diyi Yang, Srijan Kumar, Hanghang Tong, and Duen Horng Chau. Graph vulnerability and robustness: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5915–5934, 2023.

[16] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley series in artificial intelligence. Addison-Wesley, 1989.

[17] Monika Henzinger, Alexander Noe, and Christian Schulz. *Practical Fully Dynamic Minimum Cut Algorithms*, pages 13–26.

[18] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory branch-and-reduce for multiterminal cuts. In *Proc. of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2020*. SIAM, 2019.

[19] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM J. Exp. Algorithmics*, 23, October 2018.

[20] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. 2020.

[21] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.

[22] Guy Kortsarz, Robert Krauthgamer, and James R. Lee. Hardness of approximation for vertex-connectivity network design problems. *SIAM Journal on Computing*, 33(3):704–720, 2004.

[23] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Syst.*, 9, 1995.

[24] Thomas Möller. Engineering weighted connectivity augmentation problems, 2023.

[25] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17:245–264, 04 2012.

[26] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Program.*, 67(1–3):325–341, October 1994.

[27] Zeev Nutov. Approximation algorithms for connectivity augmentation problems. In Rahul Santhanam and Daniil Musatov, editors, *Computer Science – Theory and Applications*, pages 321–338, Cham, 2021. Springer International Publishing.

[28] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[29] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. *CoRR*, abs/1110.0477, 2011.

[30] Vera Traub and Rico Zenklusen. A better-than-2 approximation for weighted tree augmentation. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–12, 2022.

[31] Vera Traub and Rico Zenklusen. A (1.5+ϵ)-approximation algorithm for weighted connectivity augmentation. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 1820–1833, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Toshimasa Watanabe, Toshiya Mashima, and Satoshi Taoka. The k-edge-connectivity augmentation problem of weighted graphs. In Toshihide Ibaraki, Yasuyoshi Inagaki, Kazuo Iwama, Takao Nishizeki, and Masafumi Yamashita, editors, *Algorithms and Computation*, pages 31–40, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.