Engineering Weighted Connectivity Augmentation Algorithms

Marcelo Fonseca Faraj
Heidelberg University, Heidelberg, Germany

Ernestine Großmann
Heidelberg University, Heidelberg, Germany

Felix Joos
Heidelberg University, Heidelberg, Germany

Thomas Möller
Heidelberg University, Heidelberg, Germany

Christian Schulz
Heidelberg University, Heidelberg, Germany

Abstract

Increasing the connectivity of a graph is a pivotal challenge in robust network design. The weighted connectivity augmentation problem is a common version of the problem that takes link costs into consideration. The problem is then to find a minimum cost subset of a given set of weighted links that increases the connectivity of a graph by one when the links are added to the edge set of the input instance. In this work, we give a first implementation of recently discovered better-than-2 approximations. Furthermore, we propose three new heuristics and one exact approach. These include a greedy algorithm considering link costs and the number of unique cuts covered, an approach based on minimum spanning trees and a local search algorithm that may improve a given solution by swapping links of paths. Our exact approach uses an ILP formulation with efficient cut enumeration as well as a fast initialization routine. We then perform an extensive experimental evaluation which shows that our algorithms are faster and yield the best solutions compared to the current state-of-the-art as well as the recently discovered better-than-2 approximation algorithms. Our novel local search algorithm can improve solution quality even further.

2012 ACM Subject Classification Mathematics of computing → Optimization with randomized search heuristics; Theory of computation → Approximation algorithms analysis; Theory of computation → Randomized local search

Keywords and phrases weighted connectivity augmentation, approximation, heuristic, integer linear program, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.11

Related Version *Full Version*: `https://arxiv.org/pdf/2402.07753.pdf`

Supplementary Material *Software (Source Code)*: `https://github.com/HeiConnect`

Funding We acknowledge support by DFG grant SCHU 2567/3-1.

1 Introduction

Many real-world coherences can be modeled as graphs, including technological, social, and biological networks. A common problem of interest is the robustness of such a graph. Particularly in technological networks this is important for creating systems that are robust and fail-safe [12]. An example is a power grid where single lines can fail, either randomly due to age, or by targeted attacks. If a line fails, alternative routes are used which is increasing the load on them and therefore the chance of failure. To obtain a fail-safe network that can

survive both, random failures and targeted failures of important lines, the graph needs to be well-connected. Increasing the connectivity and therefore improving the robustness at minimum cost is known as connectivity augmentation or the survivable network problem. Another technological example is a computer network like the internet which should be designed in a fail-safe way while reliable transportation networks can avoid traffic congestion.

However, it is well-known that the weighted connectivity augmentation problem is NP-hard. Eswaran and Tarjan [8] have shown that the decision problem, whether there is an augmentation of at most a given weight, is NP-complete. Frederickson and Ja'Ja' [11] have shown that this is also true for the simpler special case where the graph is a tree, with weights being only 1 or 2. This justifies the importance of good heuristic and approximation algorithms. Furthermore, the weighted connectivity augmentation problem is APX-hard, which was also shown for the weighted tree augmentation problem by Kortsarz, Krauthgamer and Lee [21]. Despite the fact that there is no polynomial time approximation algorithm with an approximation factor arbitrarily close to 1, there has been much progress in improving the approximation ratio. Recently, the connectivity augmentation problem has been discussed frequently in the context of approximation algorithms with approximation factors below 2 [2, 27, 29, 30]. This includes work on special cases like the tree augmentation problem [29], as well as the general case [2, 30].

In recent years, Henzinger et al. [16, 17] developed the leading codes for the minimum cut problem in graphs. This includes the development of cutting-edge shared-memory inexact algorithms, consistently delivering near-optimal results. Additionally, they engineered state-of-the-art shared-memory exact algorithms [14], surpassing the previous state-of-the-art by an order of magnitude in running time, as well as highly efficient approaches for tackling the broader all minimum cut problem [18]. It turns out that these algorithms, i.e. computing minimum cuts, (enumeration of) all minimum cuts, and the efficient computation of a cactus representation of a graph, are important subroutines for algorithms that tackle the connectivity augmentation problem. Thus in this work, we heavily employ these recently developed techniques to engineer efficient algorithms for the connectivity augmentation problem.

Our Results. Our contribution in this work is two-fold. First, we give the first implementation and experimental evaluation of two recently discovered connectivity augmentation approximation algorithms due to Traub and Zenklusen [30]. More precisely, Traub and Zenklusen describe two algorithms: a greedy $(1 + \ln 2 + \epsilon)$-approximation and the local search based $(1.5 + \epsilon)$-approximation, which we implement and evaluate.

Secondly, we propose three new heuristic and an exact algorithm. The first algorithm is a greedy heuristic considering link costs and the number of cuts covered by a link. This simple algorithm already outperforms all previous state-of-the-art algorithms by more than 22 % improvement in solution quality (reduced link costs) on instances where links have small costs. Our second strategy uses minimum spanning trees to find a feasible solution first and then greedily improves it. Additionally, we present a local search algorithm that can improve a given solution by replacing link sets with cheaper ones. On instances with large link costs, the minimum spanning tree algorithm has the overall best performance regarding solution quality, running time and memory consumption. It computes solutions 8 % better than the best performing previous state-of-the-art on these instances, while being a factor of 7 times faster on average. With our local search algorithm we can further improve these solutions on average by 2 %. Lastly, we introduce a new exact solver using an ILP formulation with efficient cut enumeration as well as a fast initialization routine, for which we utilize our fast minimum spanning tree heuristic. Especially on real-world instances, it is able to outperform the previous state-of-the-art heuristic solvers regarding running time for both small and large link costs.

2 Preliminaries

An undirected graph $G = (V, E)$ is a structure that consists of a set of vertices V and a set of edges $E \subseteq \binom{V}{2}$ connecting pairs of vertices. The number of vertices is denoted as n and the number of edges as m. The graph G is *connected* if there is a path between any two vertices. The *edge connectivity* of a graph is the maximal number of edge-disjoint paths that exist between any pair of vertices. A graph is *k-edge-connected*, if $k-1$ arbitrary edges can be removed without disconnecting the graph. A partition of a graph is a partition of the vertex set into mutually disjoint sets.

A *cut* of a graph is a partition of the vertex set into two disjoint subsets, also called a bipartition. Any cut can be represented as one of its two constituent vertex sets. Every non-empty proper subset of V is a cut. To prevent different representations of the same cut we use the notation where a cut is given as one set of the partition (only the representation that does not include an arbitrarily chosen root $r \in V(G)$ is used). The *size* or *weight* of a cut is the number of edges or the sum of the edge weights that have one endpoint in each subset. A cut is a *minimum cut* if there is no cut with smaller size or weight. The set of all minimum cuts is denoted as C_G and $\text{cut} : C_G \times \binom{V}{2} \to \{0, 1\}$ is a function that is 1 if and only if the endpoints u and v of an edge $e = uv \in \binom{V}{2}$ lie in different sets of the partition of a cut $c \in C_G$.

The goal of the *Weighted Connectivity Augmentation Problem (WCAP)* is to increase the edge connectivity of a graph. More formally, for a given k-connected graph G with a set of links $L \subseteq \binom{V}{2}$ and a cost function $c : L \to \mathbb{R}_{\geq 0}$, the task is to find the cheapest subset of links $S \subseteq L$ that will increase the edge connectivity to $k+1$. A link $l \in L$ *covers* a minimum cut $c \in C_G$ if the size or weight of the cut c is increased in the graph $G' = (V, E \cup \{l\})$. The graph $G_L = (V, L)$ is called the link graph. The set of links is disjoint with the set of edges, i.e. $L \cap E = \emptyset$. For the ease of notation the cost function is extended to sets, where it is the sum of the cost of all elements.

If the input graph is disconnected, the weighted connectivity augmentation problem coincides with the minimum spanning tree problem among its components. In this work we therefore only focus on connected graphs, as the other case is simple to solve via well-known efficient minimum spanning tree algorithms.

A *cactus graph* is a connected graph, such that any two cycles have at most one vertex in common. To distinguish between edges that lie within a cycle and those that do not, they are called cycle edges and tree edges, respectively. The *cactus graph representation* of the set of minimum cuts C_G of a graph $G = (V, E)$ is a cactus graph $C = (V_c, E_c)$ with a function $\Pi : V \to V_c$ and its inverse $\Pi^{-1} : V_c \to 2^V$ which is defined as $v \mapsto \{u \in V : \Pi(u) = v\}$. The functions Π and Π^{-1} are defined such that each minimum cut in C corresponds to a minimum cut in G, i.e. for all $c \in C_C : \bigcup_{v \in c} \Pi^{-1}(v) \in C_G$ and each minimum cut in G is represented in the cactus graph, i.e. for all $c_G \in C_G$ there exists a cut $c_C \in C_C$ such that $\Pi(v) \in c_C$ for all $v \in c_G$. Figure 1 gives an example for a graph, with its minimum cuts and the corresponding cactus graph representation. Dinitz et al. [5] have shown that all minimum cuts of a connected graph $G = (V, E)$ can be represented as a cactus graph $C = (V_c, E_c)$. For more details on the computation of the cactus graph, we refer the reader to [18]. Analogous to the link graph G_L, we define the cactus link graph $C_L = (V_c, L_c)$. The set $L_c \subseteq L$ is defined as $L_c = \{(\Pi(v_1), \Pi(v_2)) : (v_1, v_2) \in L\}$ where we only keep the cheapest link between each vertex pair $(\Pi(v_1), \Pi(v_2))$.

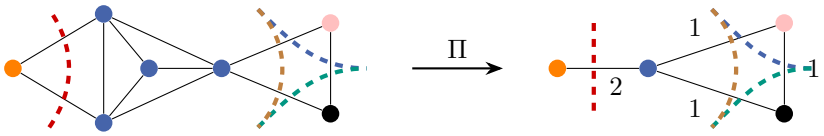

Figure 1 A graph and its weighted cactus graph with corresponding minimum cuts drawn as dashed lines of same color. Vertex colors encode the function Π.

3 Related Work

This paper is a summary and extension of the master theses [24]. In this section, we present the state-of-the-art for computing all minimum cuts of a graph in the cactus graph representation, followed by research and implementations in the field of connectivity augmentation problems.

3.1 Minimum Cuts

Computing all minimum cuts is usually a fundamental step in connectivity augmentation. Nagamochi, Nakao and Ibaraki presented an efficient algorithm to compute all minimum cuts in the cactus graph representation [25]. A cactus representation can be computed in $\mathcal{O}(mn + n^2 \log n + n^* m \log n)$ time where n^* is the number of cycles in the cactus representation. They observed that all minimum cuts between two vertices s and t can be computed by running a maximum s-t-flow algorithm, and edges that are cut by no minimum cut can be contracted.

The current state-of-the-art algorithm to compute all minimum cuts is VieCut by Henzinger et al. [14, 18]. It uses linear time edge contraction based reduction rules and an optimized version of the algorithm by Nagamochi, Nakao and Ibaraki. For example, an edge uv can be contracted if the connectivity between u and v is larger than the minimum cut. Such edges could be found by computing k edge-disjoint spanning trees where k is the size of the minimum cut [14, 26]. Furthermore, reduction rules by Padberg and Rinaldi [28] were adapted from the problem of finding one minimum cut to the problem of finding all minimum cuts. Lastly, edges that form a trivial minimum cut are contracted and remembered. These cuts are reintroduced at the end of the algorithm. The reduction rules are used exhaustively as long as a significant number of edges is contracted. The remaining kernel is solved based on the algorithm by Nagamochi, Nakao and Ibaraki [25]. In this work, we use VieCut [14, 18] to compute the cactus representation of a graph, which has a much better performance in practice than what the worst-case analysis predicts. Henceforth, when we analyse the complexity of an algorithm, we do not include the complexity to compute the cactus, as this is the same for all algorithms.

3.2 Connectivity Augmentation

There have been several approximation algorithms for the connectivity augmentation problem in the past. An early approach is using minimum cost arborescences, which was introduced by Frederickson and Ja'Ja' [11] for bridge connectivity augmentation, the case where the graph is 1-connected but not 2-connected, and generalized for the WCAP by Watanabe et al. [32]. The bridge connectivity algorithm results in a 2-approximation while the generalization cannot guarantee an approximation factor. There are well-known 2-approximations for the WCAP. One possibility was discovered in 1992 and reduces the problem to a directed version by replacing each undirected edge with two directed edges [20]. The directed version can be

solved in polynomial time based on minimum-cost flows [10] or by using a linear program which has integral solutions for the cactus augmentation problem [3]. Another approach involves the LP relaxation of an ILP formulation combined with iterative rounding techniques [19].

Only recently, progress has been made on various approximation algorithms regarding special cases of the connectivity augmentation problem, as well as the general case. For the unweighted version of the connectivity augmentation problem the first approximation with factor below 2 was found in 2020 by Byrka et al. [2, 27]. The WCAP is reduced to the steiner tree problem, for which a specialized approximation gives an approximation factor of 1.91.

For the tree augmentation problem (TAP), where the cactus graph is a tree, and the unweighted connectivity augmentation problem an approximation factor of 1.393 was found in 2021 [3]. For the weighted tree augmentation problem, Traub and Zenklusen [29] discovered a $(1 + \ln 2 + \epsilon)$-approximation, which builds upon the 2-approximation reducing the problem to a directed one and greedily improves this solution. Afterwards, they transferred the algorithm to the weighted connectivity augmentation problem and refined it to a $(1.5 + \epsilon)$-approximation [30], which improves an arbitrary solution through local search. However, no implementations or experimental results of those algorithms existed until this point.

There has been recent work on randomized Monte Carlo algorithms that give a solution with high probability on graphs with integer edge weights based on maximum flow computations. The state-of-the-art is an $\tilde{\mathcal{O}}(m)$ time algorithm that gives a near-linear running time by Cen et al. [4]. This shows that the connectivity augmentation problem is simpler than the maximum flow problem as there is no known $\tilde{\mathcal{O}}(m)$ time maximum flow algorithm.

Experimentally Evaluated Algorithms

There have been practically applicable heuristic algorithms in the past, however, there has not been much progress on the WCAP recently. Watanabe et al. [23, 34, 33, 32] proposed five different approaches, called FSA, MW, FSM, SMC and HBD, including experimental evaluation. An observation used for all algorithms is that there is a subset of all vertices of the cactus graph representation that must be an endpoint in any augmentation. The algorithm FSA uses minimum cost arborescences based on the ideas of Frederickson and Ja'Ja' [11]. MW is a 2-approximation also based on arborescences. FSM is based on maximum cost matchings. The third approach, SMC, is a greedy strategy adding the cheapest incident link for each vertex of the cactus graph representation. HBD is a combination of FSM and SMC. Experimental results showed that the solution quality of FSM is the best, followed by HBD, SMC, FSA and lastly MW [23, 32]. Regarding running time, SMC is the fastest algorithm, followed by FSA, HBD, FSM and lastly MW. HBD is considered the best general algorithm, because it prevents arbitrary bad solutions that may be produced by FSM or SMC. MW is the only algorithm with a guaranteed approximation factor, however, in practice it is slower and the solution quality of the other algorithms is better [23]. To the best of our knowledge, no other experimentally evaluated algorithms are mentioned in the literature.

4 Approximation Algorithms

We now describe the approximation algorithms implemented in this paper in more detail. First we describe the 2-approximation [3] followed by brief discussions on two approximation algorithms with approximation factors $(1+\ln 2+\epsilon)$ and $(1.5+\epsilon)$ by Traub and Zenklusen [30] for which we present first implementations and experimental evaluations. For further, highly detailed information see [3, 30].

4.1 LP-based 2-Approximation

For the 2-approximation each undirected link $l = uv$ is replaced by two directed links $l_1 = (u, v)$ and $l_2 = (v, u)$, resulting in a set $\vec{L}$ with $|\vec{L}| = 2 \cdot |L|$. Then, the easier directed problem can be solved. Later, a directed solution is transferred to the undirected problem by replacing each link in the directed solution with the undirected one while removing duplicates. To solve the directed problem, a linear program based algorithm is implemented. This approach was first proposed by Jain [19], and Cecchetto, Traub and Zenklusen [3] provided a definition where the solution is integral for any cactus augmentation instance. Using Brand's $\mathcal{O}(N^{2.37} \log^2 N \log N/\delta)$ algorithm [31] where N is the number of variables, the time complexity of the 2-approximation is $\mathcal{O}(n^{4.74} \log^2 n \log n/\delta)$.

4.2 Relative Greedy $(1 + \ln 2 + \epsilon)$-Approximation

Traub and Zenklusen presented in [30] a greedy algorithm that improves upon the 2-approximation described above. It begins by exactly reducing a cactus graph representation to a ring graph and then replacing directed links in the solution with so called shadows, to form an arborescence. These shadows are links, that have the same weight as the original links and together still form an augmentation. The algorithm proceeds by greedily substituting sets of directed links with undirected ones resulting in mixed solutions. If all directed links are replaced by undirected ones, the solution is a solution to the original problem. The greedy objective is the ratio of the cost of the added undirected links and the cost of the directed links that are not needed anymore. Since this is difficult to compute, they only consider link sets that can be constructed iteratively by a dynamic program. Furthermore, it is easier to check if a given ratio is better or worse than the optimum. The algorithm uses binary search with respect to the ratio to determine the optimum along with a set of links that achieves this ratio. For each bisection a dynamic program is run.

The $(1 + \ln 2 + \epsilon)$-approximation algorithm has a polynomial running time, but it is computationally expensive, especially due to the complexity of the dynamic program involved. Let $\alpha = 4\lceil \frac{2}{\epsilon} \rceil$, then, for graphs with $n > \alpha + 2$ the size of the dynamic programming table is $\mathcal{O}(n^{2\alpha+2})$. With this, the overall computational complexity of the algorithm for integer weighted problems is $\mathcal{O}(n^{4\alpha+7} \ln(OPT))$, where OPT is the optimal augmentation weight. Note that the approximation ratio can only be improved if $1 + \ln 2 + \epsilon < 2$ requiring $\alpha \geq 28$.

4.3 Local Search $(1.5 + \epsilon)$-Approximation

The state-of-the-art approximation algorithm by Traub and Zenklusen [30] is a $(1.5 + \epsilon)$-approximation. A more detailed description and correctness proofs can be found in [30]. The algorithm is based on the ideas and the dynamic program of their relative greedy $(1 + \ln 2 + \epsilon)$-approximation described previously. The main difference is that the algorithm does not only greedily replace all links of a directed solution with undirected links. Instead, replaced links should iteratively improve the solution and can themselves be replaced in further iterations. The main part of this algorithm, the dynamic program, is the same as for the $(1 + \ln 2 + \epsilon)$-approximation. Therefore, the size of the dynamic programming table is bounded by $\mathcal{O}(n^{2\alpha+2})$. The total running time is bounded by $\mathcal{O}(n^{4\alpha+7}/\epsilon)$. In contrast to the $(1 + \ln 2 + \epsilon)$-approximation this is independent of the (upscaled) augmentation weight OPT.

5 Weighted Connectivity Augmentation Algorithms

This section describes our heuristic approaches and our exact algorithm for the WCAP. The data structures used for these algorithms are briefly described in Section 5.1. Unlike the approximation algorithms in the previous section, the heuristics cannot give guarantees on the solution quality, but aim at being fast or giving good solutions for many real-world cases. For all considered algorithms we only compute a solution on the cactus graph representation which can be transformed to the solution on the original graph, as stated in Theorem 1.

▶ **Theorem 1** (Dinic, Karzanov and Lomonosov [5]). *Let G be a graph and L the set of links. Furthermore, let C be the corresponding cactus graph representation of G and C_L be the link graph of C. Then, a valid solution for the WCAP on C is a valid solution to the WCAP on G.*

5.1 Data Structures

All minimum cuts of a graph G can be represented as a (potentially significantly smaller) cactus graph C, it is sufficient to do computations on the cactus graph C. To be able to give an augmentation for the original graph G, it is stored in adjacency list representation along with an array modeling the function $\Pi : V(G) \to V(C)$. Additionally, the link set L must be transferred to a link set L_C for the cactus graph C. A link $l = (u, v)$ is translated to a link $l_C = \Pi(l) := (\Pi(u), \Pi(v))$ in the cactus graph. To be able to reverse this function and obtain a link in G, the endpoints of the original link l are stored as well. This can lead to parallel links $g, h \in L, g \neq h$ with $\Pi(g) = \Pi(h)$ in the link graph G_{L_C} (with not necessarily equal weight). We drop all parallel links in G_{L_C} except for one of smallest weight. We store the link set L_C in an adjacency matrix and only keep one link of minimum weight per vertex pair.

Dynamic Cactus / Updating Cactus

For a dynamic cactus representation we use a similar approach for updating the cactus as proposed by Henzinger, Noe and Schulz in [15], where a union find data structure is used to keep track of the function Π that associates each vertex of G with a vertex of the cactus C.

When adding a link $l = (u, v)$ that crosses a minimum cut, the cactus shrinks. Dinitz has shown that cuts on the uv-path are affected [6]. In particular, vertices that lie on every uv-path in the cactus are contracted to update the cactus [15]. Those can be found by computing a path in the *tree of cycles* $\mathcal{T}$. This is the graph that contains all cycles as vertices and has an edge between two vertices if the corresponding cycles share a vertex. Since $\mathcal{T}$ is a tree, the paths are unique. For every cycle in the path, the shared endpoints are contracted. This graph $\mathcal{T}$ is maintained by our data structure.

To be able to efficiently compute the number of minimum cuts that a given link crosses, this data structure is extended and modified. The naive approach of counting the number of cuts that a given link crosses evolves checking every possible cut and takes $\mathcal{O}(|V_c|^2)$.

▶ **Theorem 2.** *Using our data structure, we can compute the number of cuts that a given link crosses in $\mathcal{O}(|V_c|)$.*

Proof. A cut always cuts two edges of the same cycle, therefore each cycle can be considered separately. As a link $l = (u, v)$ affects all cuts on the uv-path [5], all cycles on the uv-path need to be considered. These cycles are found by a BFS in $\mathcal{T}$ taking $\mathcal{O}(|V_c|)$ time because $\mathcal{T}$ is a tree and there cannot be more cycles than vertices in the cactus graph. Counting the

Algorithm 1 GWC.

input $G = (V, E)$, L, $c : L \to \mathbb{R}_{\geq 0}$
output augmentation $S \subseteq L$
procedure GWC(G, L, c)
  $C = (V_c, E_c) \leftarrow$cactusByVieCut($G$)
  $C_L = (V_c, L_c) \leftarrow$buildLinkGraph($V_c, L$)
  $S \leftarrow \emptyset$
  while $L_c \neq \emptyset$ **do**
    $l \leftarrow \text{argmin}\{c(l)/\text{coveredCuts}(C, l) \mid l \in L_c\}$
    $S \leftarrow S \cup \{l\}$
    $C \leftarrow$updateCactus(C, l)
    $L_c \leftarrow \{l \in L_c \mid l \text{ covers a mincut in } C\}$
  return S

number of cuts that the computed path crosses can be done in linear time too. For each edge in the path we go through all vertices contained in the corresponding cycle to compute the distance of the articulation points. As this is done at most once per cycle, we are able to compute the number of cuts that a given link crosses within $\mathcal{O}(|V_c|)$. ◀

For further implementation details, we refer to [24].

5.2 Heuristic Algorithms

In this section we introduce a new heuristic GreedyWeightCoverage (GWC), where links are added to the solution greedily based on the costs per augmented cut, an algorithm based on minimum spanning trees called MSTConnect as well as a local search algorithm LS(k).

5.2.1 GWC

As the problem aims at minimizing the cost of an augmentation, it is natural to add links of small weight. At the same time, we want to minimize the number of links added. This leads to the heuristic GWC, where we greedily pick the link, for which the cost per augmented minimum cut, i.e. $c(l)/a_l$ with $a_l = \text{coveredCuts}(C, l) := |\{c \in C_G : \text{cut}(c, l) = 1\}|$ is minimal. If $a_l = 0$ the link l is not considered. This regards listing all minimum cuts in the cactus graph computed using VieCut [14, 18].

A naive bound for the complexity of this is $\mathcal{O}(|V_c|^5)$, because the solution has at most $\mathcal{O}(|V_c|)$ links, and in each iteration the heuristic is computed for $\mathcal{O}(|L_c|) = \mathcal{O}(|V_c|^2)$ links by checking $\mathcal{O}(|V_c|^2)$ minimum cuts. Using our custom data structure, see Section 5.1, we can run the algorithm in $\mathcal{O}(|V_c|^4)$.

The bound is very pessimistic and in practice the performance of the algorithm is much better, especially since we use efficient algorithms to enumerate all minimum cuts [18].

We also tested other heuristics such as choosing the link with the smallest weight, such that at least one cut is covered, or picking an arbitrary uncovered minimum cut c and choosing the smallest weight link that covers c. However, GWC produced significantly better results, which is why we only report details of this algorithm here.

Algorithm 2 MSTConnect.

input $G = (V, E)$, L, $c : L \to \mathbb{R}_{\geq 0}$
output augmentation $L' \subseteq L$
procedure MSTConnect(G, L, c)
  $C = (V_c, E_c) \leftarrow$cactusByVieCut($G$)
  $C_L \leftarrow$buildLinkGraph(V_c, L)
  $L_{MST} \leftarrow$ MST(C_L, c)
  $L' \leftarrow$ sortDesc(L_{MST}, c)
  for $l \in L'$ **do**
    if disposableLink(l, C, L')
    $L' \leftarrow L' \setminus \{l\}$
  return L'

5.2.2 MSTConnect

After computing the cactus graph representation using VieCut [14, 18], the greedy strategies described above work by adding links to a set until this set is a valid connectivity augmentation. For MSTConnect, see Algorithm 2, we have a different approach. Here, we start with a (possibly much larger) set of links that increases the connectivity when added to the cactus graph. Then, we reduce this link set, while keeping a valid solution.

The complete set of links can have $\mathcal{O}(|V_c|^2)$ size. As checking if a link can be removed from a set can also be expensive, a small initial solution is essential. Using Theorem 3, an intuitive starting point is a minimum spanning tree (in case of incomplete link sets a minimum spanning forest) L_{MST} of the cactus link graph C_L. For each link in this set, we check whether it can be removed in disposableLinks of Algorithm 2. Since we start with a MST the remaining links L' are always bounded such that $|L'| < |V_c|$. Therefore, this process needs linear time, as shown in Theorem 4. If a link is disposable, we exclude it from the solution. Figure 2 (left) illustrates an example.

▶ **Theorem 3.** *Let $C = (V_c, E_c)$ be a cactus graph and $C_L = (V_c, L_c)$ the link graph of C, such that a feasible solution to the WCAP exists. Then, a minimum spanning forest $L_{MST} \subset L_c$ for C_L is a feasible solution to the WCAP on C.*

Proof. See Appendix B. ◀

▶ **Theorem 4.** *Let $G = (V, E)$ be a k-connected graph and $C = (V_c, E_c)$ its cactus graph representation, $L' \subset L$ an augmentation on the link graph C_L and $l \in L'$. If $|L'| < |V_c|$, then we can check if $L' \setminus \{l\}$ is still an augmentation in $\mathcal{O}(|V_c|)$.*

Proof. See Appendix B. ◀

A minimum spanning forest on the cactus link graph C_L is computed using Kruskal's algorithm with a complexity of $\mathcal{O}(m \log m)$ [22], where m is the number of links in C_L. Checking if $\mathcal{O}(|V_c|)$ links can be removed from the solution is done in $\mathcal{O}(|V_c|^2)$, see Theorem 4. This results in an overall complexity of $\mathcal{O}(|V_c|^2 \log |V_c|)$ for MSTConnect.

As in GWC, we also tested the cost per augmented minimum cut $c(l)/a_l$ as weight to compute the minimum spanning forest as well as using $c(l)/a_l$ to sort the links to be removed in Algorithm 2. However, our experiments showed that using the weight $c(l)$ of the links performed significantly better in both cases and thus the other versions are omitted.

Algorithm 3 LS(k).

input $G = (V, E)$, L, $c : L \to \mathbb{R}_{\geq 0}$, S
output improved augmentation $S' \subseteq L$
procedure LS(k)(G, L, c, S)
  $C = (V_c, E_c) \leftarrow$ cactusByVieCut(G)
  $L' \leftarrow$ reduceLinkSet(C, L, c)
  $X \leftarrow$ getSwapCands(L', S)
  $S' \leftarrow S$
  while $X \neq \emptyset$ **do**
    $(L_{in}, L_{out}) \leftarrow$ bestCandidate(X)
    $X \leftarrow X \setminus (L_{in}, L_{out})$
    if isSwapValid(L_{in}, L_{out})
      $S \leftarrow$ swap(S', L_{in}, L_{out})
      $X \leftarrow$ getSwapCands(L', S')
return S'

5.2.3 Local Search

The core idea of our local search algorithm LS(k) is the following. We want to remove links from our solution and replace them with a lighter set of links. The parameter k limits the number of links within such a swap. Since these swaps can result in infeasible solutions, we have to check feasibility for each swap. This way a non-optimal solution $S \subset L$ can be improved. Next we describe the different steps of LS(k) in detail, an overview is given in Algorithm 3.

We first compute the cactus graph representation C of G using VieCut [14, 18]. Afterwards, we reduce the link set L to the union of t disjoint minimum spanning forests in the link graph C_L. Here, the next minimum spanning forest is computed on C_L after removing the links of the previously computed minimum spanning forest. This way the average degree is a small constant depending on t. We set $t = 2$, since larger values did not improve the performance.

The main idea of the remaining part in LS(k) is the following. We search for non-solution links $L_{in} \subset L \setminus S$ and solution links $L_{out} \subset S$ with $|L_{in}| \leq k_{in}$, $|L_{out}| \leq k_{out}$ and $k_{in} + k_{out} \leq k$. If the swap (L_{in}, L_{out}) is feasible, we can create a new improved solution by swapping these sets, i.e. $S' \coloneqq S \setminus L_{out} \cup L_{in}$. In getSwapCandidates we compute possible swap candidates (L_{in}, L_{out}). These have to fulfill the following conditions:

- $L_{in} \cup L_{out}$ form an alternating path from L_{in} and L_{out} of length at most $k_{in} + k_{out} = k$
- $c(L_{in}) - c(L_{out}) < 0$, i.e. the swap improves the solution
- non-triviality, i.e. $\forall v \in V(L_{out}) : \{v\}$ is no minimum cut

We restrict the search for swaps to alternating paths. Since endpoints of these paths in L_{out} lose an adjacent link, it is likely that a new adjacent link is necessary to keep the cut covered. Interior links of the path cover two vertices at once. Utilizing the specified criteria, we determine the sets L_{in} and L_{out} within the getSwapCandidates function. This is achieved through an adapted depth-first search process, where traversal is restricted to alternate edges that are part of the solution and those that are not. The depth of the search is restricted by the parameter k. This modified search is executed starting at each vertex in $|V_c|$. At every vertex encountered, we verify the adherence to the second and third condition. If they are met, the path currently identified is included in the collection of potential swap candidates.

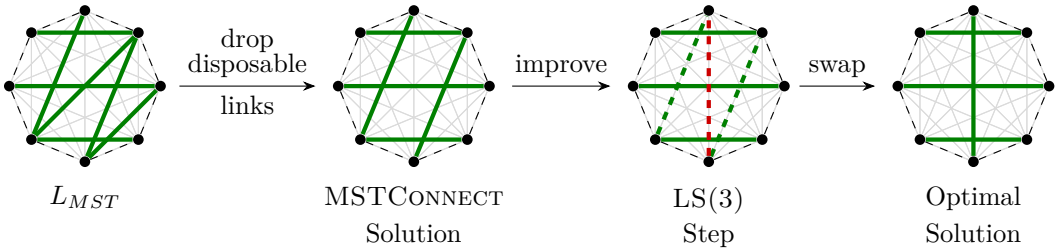

Figure 2 Example of applying MSTConnect (right) and improving its solution with our local search LS(3). All links have weight 1. Green edges represent the current solution, black dashed edges are the edges of the cactus graph representation, light gray edges are the non-solution links. The dashed green (L_{out}) and red (L_{in}) edges show the swap found in the local search.

These conditions are not sufficient to form a feasible swap, they rather help to prune the set of possible candidates in advance. For the best candidate, i.e. the candidate where $c(L_{in}) - c(L_{out})$ is lowest, we check the validity of the swap in isSwapValid. Using a maximum flow approach, similar to MSTConnect, this check can be done efficiently as shown in Theorem 4. If the swap is valid, we swap links, recompute possible candidates and start over. If the swap is not valid, we check the next best candidate.

Figure 2 shows an example of how the LS(3) algorithm improves the solution computed by MSTConnect. We further improved the algorithm LS(k) by integrating path caching, where we maintain vertex sequences of a path already checked using a hash table. Using Theorem 5, the running time for LS(3) is $\mathcal{O}(|V_c|^3 \times \Delta^2 + |V_c|^4)$ and for LS(5) $\mathcal{O}(|V_c|^4 \times \Delta^2 + |V_c|^5)$.

▶ **Theorem 5.** *LS(k) has a complexity of* $\mathcal{O}(|V_c|^2(|V_c|^{\lfloor \frac{k}{2} \rfloor} \times \Delta^2 + |V_c|^{\lceil \frac{k}{2} \rceil}))$.

Proof. See Appendix B. ◀

5.3 Exact Algorithm

We now describe an efficient implementation of an integer linear program for the problem, which can be used to solve (small) instances to optimality. The formulation is inspired by the linear program used for the 2-approximation in [3]. To the best of our knowledge, this formulation itself has not been used before. For the formulation of the WCAP, we introduce binary variables $x \in L$ that decide which links are added to the augmentation. The objective given in (1) sums up the weight of all selected links. The constraint ensures that each minimum cut in the graph G is increased by at least one link added to the augmentation.

$$\min_x \sum_{l \in L} x_l c(l) \quad \text{s.t.} \quad \sum_{l \in L} \text{cut}(c, l) x_l \geq 1 \quad \forall c \in C_G; \quad x_l \in \{0,1\}^{|L|}. \tag{1}$$

Our ILP formulation is run on the cactus representation and uses efficient enumeration algorithms on it to list all minimum cuts C_G of VieCut [18]. Moreover, our solver, which we call eILP, uses an initial solution computed by MSTConnect, which improves the running time by 5.5% on cycle graphs, which are the most difficult instances to solve with eILP. We choose MSTConnect since it is the fastest heuristic approach already giving good results (see Section 6).

6 Experimental Evaluation

We now experimentally evaluate the algorithms described in the previous sections. The approximation algorithms are evaluated and compared in terms of quality and running time in Section 6.2. Afterwards, our new proposed algorithms GWC, MSTConnect and MSTConnect+LS(k) are compared to the state-of-the-art solvers HBD, FSM and SMC in terms of solution quality, running time and memory consumption.

6.1 Overview

6.1.1 Methodology

The experiments are run on a computer with an AMD EPYC 9754 128 core processor with 256 threads running at 3.1 GHz and 755 GB of main memory running Linux. The C++ code is compiled using `gcc 11.4.0` with optimization level `O3`. The memory for each process is limited to 50 GB and the running time is limited to 3 hours. We run our algorithms on each instance with 5 different seeds to generate link costs as described below or in cases of generated cactus graphs we create 5 different graphs with the same number of vertices and cycles. The objective, i.e. the weight of the augmentation, the running time and the maximum memory used is measured. We use the geometric mean when averaging over different seeds or instances such that every instance has a comparable influence on the result.

Different algorithms are compared using performance profiles [7]. These plots use the best algorithm as baseline for each instance and relate the other algorithms to this baseline. A performance profile can use the objective function to compare quality, running time and memory consumption. The x-axis shows a parameter $\tau \geq 1$. On the y-axis the fraction of instances whose objective is at most $\tau \cdot$ best is plotted, in particular $\#\{\text{objective} \leq \tau \cdot \text{best}\}/\#\text{instances}$. For running time and memory usage, time and memory are used instead of the objective, respectively. At $\tau = 1$ the plot shows the fraction of instances where the algorithm is able to find the best solution / has the fastest running time or lowest memory consumption. Some algorithms are not able to solve every instance due to constraints on memory and time. We give more details in the respective sections. To solve the integer linear program in eILP and the linear program in the 2-approximation algorithm we used the Gurobi Optimizer [13]. The integer linear program solver uses pre-solving and the MSTConnect solution as initial solution.

6.1.2 Instances

The algorithms are evaluated using two different sets. The first set consists of three types of *generated graph instances*: cycles or ring graphs, stars and cactus graphs. Cycles and stars represent edge cases of cactus graphs, with an amount of minimum cuts between $\mathcal{O}(|V_c|)$ and $\mathcal{O}(|V_c|^2)$. To be able to test the algorithms on instances that represent more complex and larger cactus graphs, we generated cactus graphs by the algorithm described in Appendix A. All instances are listed in Table 1 including their properties. The second set are *real-world instances*. Many real-world graphs have unique or very few distinct minimum cuts, which leads to very small cactus graphs with only a few vertices. We used all connected graphs with non-trivial cactus graph representations from the 10th DIMACS Implementation Challenge [1] for which the cactus graph representation has at least 100 edges and at most 40 000 vertices. In general, we use VieCut [18] to compute the cactus representation of a graph. We only compute the cactus representation once for every graph and henceforth only report running times of the algorithms when run on the cactus representation.

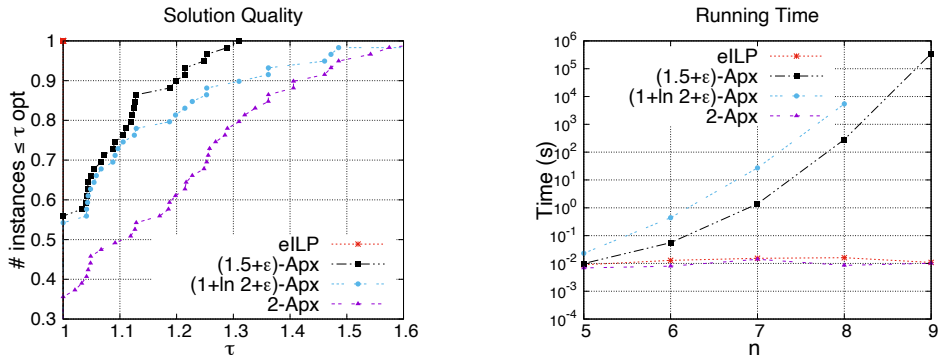

Figure 3 Performance profile comparing solution quality of approximations with $\epsilon = 0.15$ on tiny cycle and star instances on the left. The right plot shows the running times.

6.1.3 Link Sets

The algorithms can perform differently based on the distribution of the link costs. For instance the performance of MSTConnect is affected due to different structured minimum spanning trees and lower cardinality of optimum solutions make matching-based approaches like FSM and HBD or our greedy heuristic GWC more efficient. Thus, we choose a uniform distribution in different intervals. First, we have a set of *small link cost* choosing the costs uniformly from the sets $\{1, 2\}$, $\{1, \ldots, 9\}$ and $\{1, \ldots, 99\}$ as used in [32, 34] to reproduce their results. The second set of *large link costs* consists of links chosen uniformly from the set $\{1, \ldots, 100\,000\}$. Since our algorithms need link costs within the interval $[0, 1]$, we scale those costs by dividing through the largest occurring link cost. Furthermore, all instances have complete link sets, i.e. $G = (V, E \cup L)$ is a complete graph.

6.2 Approximations

We first analyse the performance of the different approximation algorithms described in Section 4. Here, we can only consider very small graphs, as the approximation algorithms (see below) do not scale at all. For following experiments, we only report the results for $\epsilon = 0.15$. This is due to the fact that ϵ is only used to limit the number of considered links crossing cuts to $4\lceil\frac{4}{\epsilon}\rceil$ for the $(1.5 + \epsilon)$-Approximation and $4\lceil\frac{2}{\epsilon}\rceil$ for the $(1 + \ln 2 + \epsilon)$-Approximation. However, on small graphs the number of links that may cross a minimum cut is already limited by the number of vertices. Thus, although we tested values for ϵ in $[0.1, 0.5]$, it did not yield different result in terms of running time or link cost.

Figure 3 (left) shows a performance profile of the solution quality, for tiny graphs. As the optimal solution was computed by eILP, the x-axis gives the approximation ratio. The 2-approximation consistently gives the worst solutions, which means both, the $(1 + \ln 2 + \epsilon)$-approximation and the $(1.5 + \epsilon)$-approximation, can improve this initial solution.

Figure 3 (right) shows the running time of the approximation algorithms as well as our solver eILP with respect to the graph size. The graphs are cycle graphs with a complete set of links. The running time increases exponentially for the approximation algorithms. This is expected for graphs with fewer than $\alpha \geq 28$ vertices as the computational complexity is exponential in $4\min(n, \alpha) + 7$. Both, the $(1 + \ln 2 + \epsilon)$-approximation and the $(1.5 + \epsilon)$-approximation are orders of magnitude slower than the optimal integer linear program. Hence,

we do not consider them further when comparing against other state-of-the-art algorithms. The 2-approximation is slightly faster than eILP, but the difference is negligible and both can easily solve tiny graphs solvable by the dynamic program based approximations. Overall, we *conclude* that the approximations have very little relevance for solving connectivity augmentation problems and may only have theoretical value. Since the 2-approximation is able to solve larger graphs it is also used for the comparisons in the following.

6.3 State-of-the-Art Comparison

In this section we compare our algorithms GWC, MSTConnect and MSTConnect combined with LS(k) as well as eILP against the performance of the 2-approximation algorithm as well as to the best, i.e. FSM and HBD, and the fastest, i.e. SMC, state-of-the-art solvers from Watanabe et al. [32, 33, 34]. For the algorithms from Watanabe et al. [32, 33, 34], neither the instances used in their experimental evaluation nor source code or binaries for the algorithms are available.[1] Hence, we compare them against *our* implementation of these algorithms. For the comparison of solution quality, running time and memory consumption we give performance profiles for generated and real-world instances in Figure 4 (small link cost) and Figure 5 (large link cost).

General Remarks. First note that the 2-approximation algorithm yields the worst results (highest cost) on both data sets and link costs. Additionally, it is also the slowest and most memory consuming algorithm. The high memory consumption results from a) the reduction of undirected to directed links, which doubles the size of the link set which roughly gives factor of two and b) more importantly from running the linear program solver. We now analyse results for the remaining algorithms for different link costs and instances types separately.

6.3.1 Small Link Costs

The best performing *heuristic* in terms of solution quality on the graphs with small link costs is GWC, see Figure 4. Unlike the other algorithms, GWC uses the cost per augmented minimum cut to decide for the solution links. For small link costs, there are a lot of links having the same or similar costs. Therefore, the cost per augmented minimum cut is more impactful than relying only on the cost of a link. For small link costs the MSTConnect algorithms are outperformed by the competitors FSM and HBD regarding the solution quality. On the instances solvable by all algorithms, GWC achieves solutions that are on average only 4% larger than the optimum solution, while the results computed by the competitors FSM and HBD are on average 28 % larger than the optimum solution value. The SMC algorithm computes worse results than any of our algorithms, performing only slightly better than the 2-approximation. Overall, our exact solver eILP can solve 85.6 % of these instances to optimality. As can be seen in Figure 4 (middle), MSTConnect is the overall fastest algorithm. On *generated* instances with small link costs GWC, which yielded the overall best results on these graphs, is on average the second fastest algorithm for these instances. On *real-world* instances on the other hand, our exact solver eILP is on average more than 3 times faster than GWC and also outperforms all the previous state-of-the-art competitors regarding running time. Since the cactus graph representation of real-world instances usually do not have large cycles, this benefits our solver eILP. The number of minimum cuts and therefore the number of constraints of the ILP (1) grows quadratically with the size of the largest cycle and only linearly with tree edges.

[1] We contacted the authors, however, we did not get an answer.

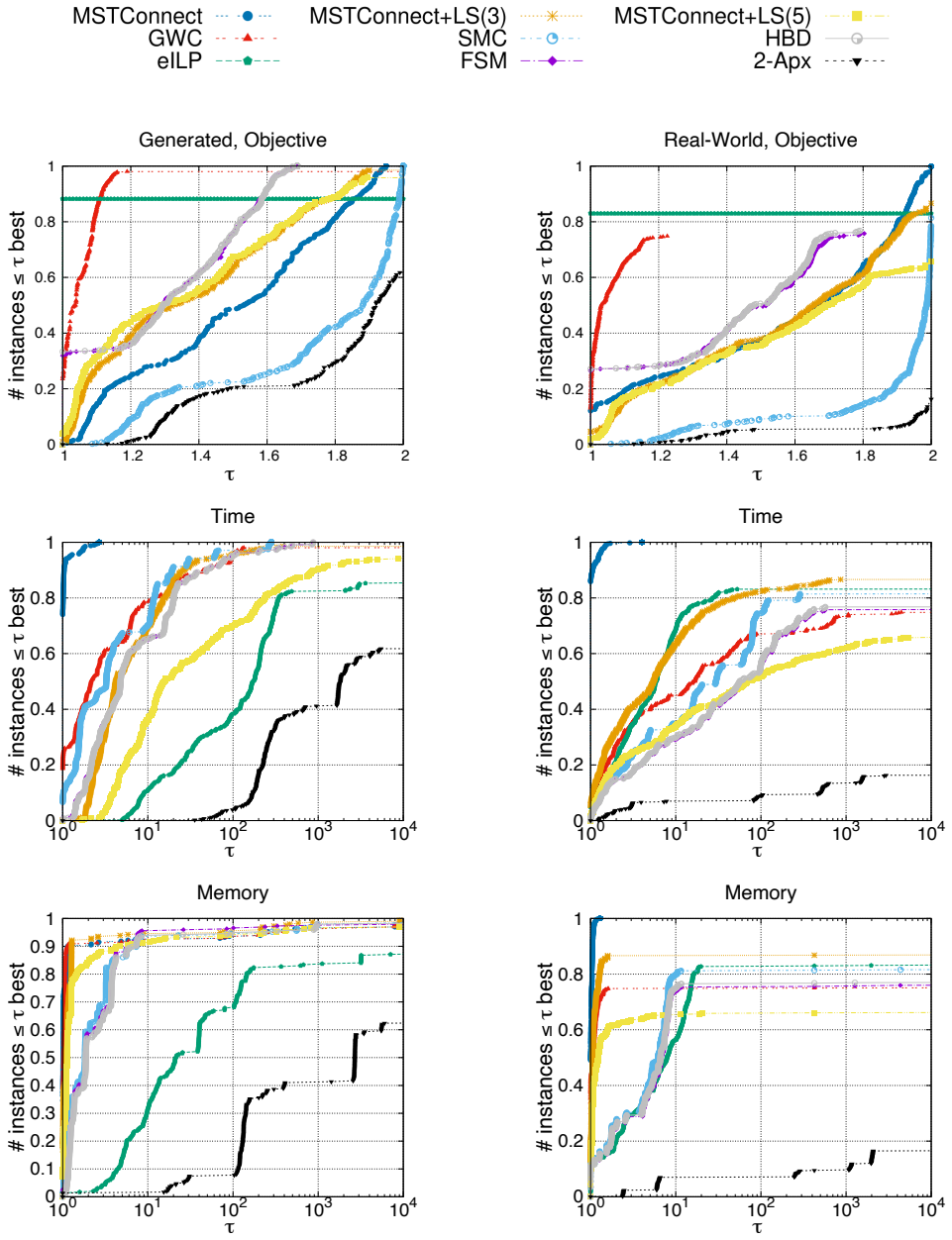

Figure 4 Performance profile for the state-of-the-art comparison on solution quality, running time and memory consumption on instances with *small cost links*.

6.3.2 Large Link Costs

The best (heuristic) results for graphs with large link costs are achieved by MSTConnect improved by LS(k) for $k = 5$ closely followed by $k = 3$, see Figure 5 (top). On this set, the wide range of different link costs makes the cost per link more significant, which benefits MSTConnect. *All* competing algorithms HBD, FSM and SMC yield worse results compared to all our MSTConnect approaches including MSTConnect without local search and GWC. In particular, our fastest algorithm MSTConnect, is on average 8 %

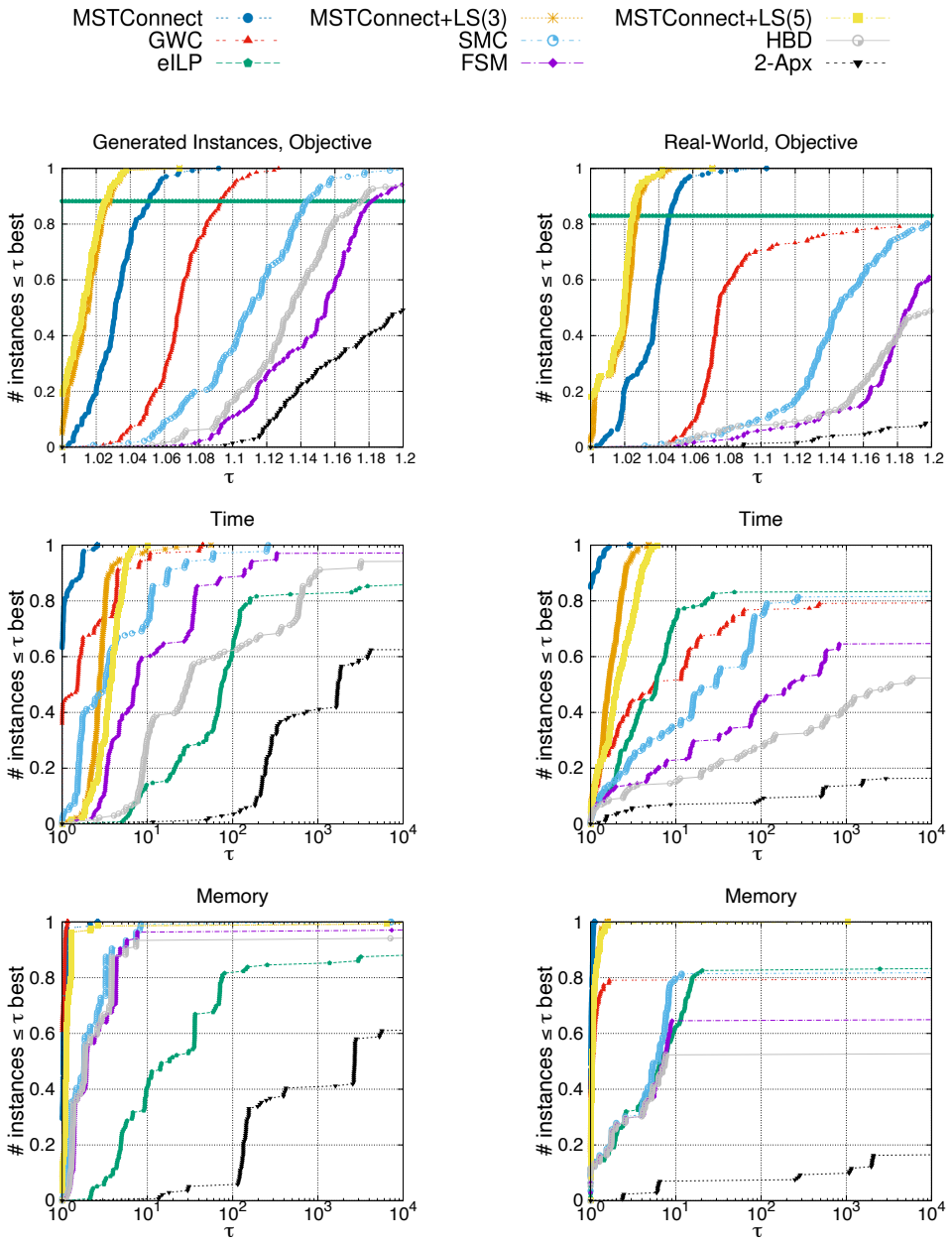

Figure 5 Performance profile for the state-of-the-art comparison on solution quality, running time and memory consumption on instances with *large cost links*.

(restricted to the instances solvable by SMC) better compared to SMC, the best performing competitor on these instances, while also being 7 times faster. When comparing to the exact solution on the instances solvable by EILP, MSTCONNECT is on average 3.8 % away from the optimal solution. When additionally using our local search to improve the solution, i.e. using LS(5), the result is only 1.8 % away from the optimum solution. All of our heuristic algorithms outperform even the fastest competitor SMC with regard to running time on most large cost real-world instances. On all instances even our slowest algorithm GWC is 1.7 times faster than SMC, while our fastest algorithm MSTCONNECT outperforms SMC by

a factor of 6.3. When only considering real-world instances, the difference gets even larger. Here, MSTConnect is on average 9.1 times faster than SMC and even our exact solver eILP is a factor of 2.6 times faster than SMC.

6.3.3 Memory Consumption

MSTConnect requires overall the least amount of memory. Indeed, improving the solution with LS(k) increases the memory consumption with increasing k, especially on small link cost instances. Still, the memory usage for MSTConnect + LS(5) is on average 1.8 times better than the memory consumption of the previous state-of-the-art algorithms. On average all our algorithms outperform the competitors regarding memory consumption on *generated* graphs. For the *real-world* instances only MSTConnect and MSTConnect + LS(3) (with large link costs also LS(5)), need less memory than the other algorithms on almost every instance. Overall, MSTConnect uses 2.6 times less memory than the competitor SMC with the lowest memory consumption. Lastly, we like to note that most of the real-world instances not solvable by eILP are only solvable with our MSTConnect approaches.

6.3.4 Advise for Practitioners

A critical factor for the algorithms is the cost associated with the links of the augmentation problem. When dealing with scenarios where the link costs are very large, it is advisable to lean towards MSTConnect algorithms. Conversely, for cases where the link costs are relatively small and instances are similar to the generated ones used in this work, the GWC algorithm emerges as a preferable choice. For other real-world instances with small link cost, the eILP method is the algorithm of choice. In general, local search can help to improve the result at the expense of running time.

7 Conclusion

In recent years, new scalable algorithms for the minimum cut and the all minimum cut problem have been engineered [16, 17]. These algorithms are important subroutines for algorithms that tackle the connectivity augmentation problem. This inspired us to engineer novel efficient algorithms for the connectivity augmentation problem. In this work, we implemented recently published approximation algorithms, as well as new heuristic strategies and an exact approach for solving the weighted connectivity augmentation problem in large graphs efficiently. Our greedy heuristic GWC excels in solving small link cost instances, while our minimum spanning tree-based algorithm MSTConnect is the top choice for large cost instances in terms of solution quality, running time, and memory consumption. Additionally, we introduce a novel local search algorithm LS(k) that enhances existing solutions – the first local search algorithm in the literature. Lastly, we engineer an exact solver eILP, which is also able to compete regarding running time on real-world instances with small link costs. We conducted experiments comparing our implementations with our best faith implementation of Watanabe et al.'s state-of-the-art solvers. Our algorithms significantly surpass their results in solution quality, running time, and memory consumption and are very close to optimal solutions. Surprisingly, on real-world instances all previous state-of-the-art algorithms, i.e. the algorithm by Watanabe et al., are even outperformed in terms of running time by our exact solver eILP on the (large number of) instances that it could solve. Instances that the eILP approach could not solve, have also not been solved by the algorithms by Watanabe et al.. Important future work includes exploring additional local search algorithms and reducing its search space through additional pruning. Additionally, we want to improve the scalability of these algorithms even further.

References

1 David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018. doi:10.1007/978-1-4939-7131-2_23.

2 Jaroslaw Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: A reduction to steiner tree. *SIAM J. Comput.*, 52(3):718–739, 2023. doi:10.1137/21M1421143.

3 Federica Cecchetto, Vera Traub, and Rico Zenklusen. Bridging the gap between tree and connectivity augmentation: unified and stronger approaches. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 370–383. ACM, 2021. doi:10.1145/3406325.3451086.

4 Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Edge connectivity augmentation in near-linear time. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 137–150. ACM, 2022. doi:10.1145/3519935.3520038.

5 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,*, January 1976.

6 Yefim Dinitz. Maintaining the 4-edge-connected components of a graph on-line. In *Second Israel Symposium on Theory of Computing Systems, ISTCS 1993, Natanya, Israel, June 7-9, 1993, Proceedings*, pages 88–97. IEEE Computer Society, 1993. doi:10.1109/ISTCS.1993.253480.

7 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002. doi:10.1007/S101070100263.

8 Kapali P. Eswaran and Robert Endre Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976. doi:10.1137/0205044.

9 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

10 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. doi:10.1016/0024-3795(89)90469-2.

11 Greg N. Frederickson and Joseph F. JáJá. Approximation algorithms for several graph augmentation problems. *SIAM J. Comput.*, 10(2):270–283, 1981. doi:10.1137/0210019.

12 Scott Freitas, Diyi Yang, Srijan Kumar, Hanghang Tong, and Duen Horng Chau. Graph vulnerability and robustness: A survey. *IEEE Trans. Knowl. Data Eng.*, 35(6):5915–5934, 2023. doi:10.1109/TKDE.2022.3163672.

13 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

14 Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory exact minimum cuts. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 13–22. IEEE, 2019. doi:10.1109/IPDPS.2019.00013.

15 Monika Henzinger, Alexander Noe, and Christian Schulz. Practical fully dynamic minimum cut algorithms. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 13–26. SIAM, 2022. doi:10.1137/1.9781611977042.2.

16 Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. In *ALENEX*, pages 48–61. SIAM, 2018.

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

18 Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ESA.2020.59.

19 Kamal Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Comb.*, 21(1):39–60, 2001. doi:10.1007/S004930170004.

20 Samir Khuller and Ramakrishna Thurimella. Approximation algorithms for graph augmentation. *J. Algorithms*, 14:214–225, 1993. doi:10.1006/JAGM.1993.1010.

21 Guy Kortsarz, Robert Krauthgamer, and James R. Lee. Hardness of approximation for vertex-connectivity network design problems. *SIAM J. Comput.*, 33(3):704–720, 2004. doi:10.1137/S0097539702416736.

22 Harry R. Lewis. *Ideas That Created the Future: Classic Papers of Computer Science*. The MIT Press, February 2021. doi:10.7551/mitpress/12274.001.0001.

23 Toshiya Mashima and Toshimasa Watanabe. Approximation algorithms for the k-edge-connectivity augmentation problem. In *1995 IEEE International Symposium on Circuits and Systems, ISCAS 1995, Seattle, Washington, USA, April 30 - May 3, 1995*, pages 155–158. IEEE, 1995. doi:10.1109/ISCAS.1995.521474.

24 Thomas Möller. *Engineering Weighted Connectivity Augmentation Problems*. master thesis, Universität Heidelberg, 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, April 2012. doi:10.1007/BF03167346.

26 Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Program.*, 67:325–341, 1994. doi:10.1007/BF01582226.

27 Zeev Nutov. Approximation algorithms for connectivity augmentation problems. In Rahul Santhanam and Daniil Musatov, editors, *Computer Science - Theory and Applications - 16th International Computer Science Symposium in Russia, CSR 2021, Sochi, Russia, June 28 - July 2, 2021, Proceedings*, volume 12730 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2021. doi:10.1007/978-3-030-79416-3_19.

28 Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, 1991. doi:10.1137/1033004.

29 Vera Traub and Rico Zenklusen. A better-than-2 approximation for weighted tree augmentation. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1–12. IEEE, 2021. doi:10.1109/FOCS52979.2021.00010.

30 Vera Traub and Rico Zenklusen. A (1.5+ϵ)-approximation algorithm for weighted connectivity augmentation. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1820–1833. ACM, 2023. doi:10.1145/3564246.3585122.

31 Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 259–278. SIAM, 2020. doi:10.1137/1.9781611975994.16.

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.

33 Toshimasa Watanabe, Toshiya Mashima, and Satoshi Taoka. Approximation algorithms for minimum-cost augmentation to k-edge-connect a multigraph. In *1993 IEEE International Symposium on Circuits and Systems, ISCAS 1993, Chicago, Illinois, USA, May 3-6, 1993*, pages 2556–2559. IEEE, 1993.

34 Toshimasa Watanabe, Satoshi Taoka, and Toshiya Mashima. Maximum weight matching-based algorithms for k-edge-connectivity augmentation of a graph. In *International Symposium on Circuits and Systems (ISCAS 2005), 23-26 May 2005, Kobe, Japan*, pages 2231–2234. IEEE, 2005. doi:10.1109/ISCAS.2005.1465066.

A Cactus Graph Generation

This section describes an algorithm that is able to generate graphs with given properties, namely the number of vertices and the number of cycles.

A.1 Generating Cactus Graphs

Given two integers $n, c \in \mathbb{N}$, $n > c$, the goal is to generate a cactus graph C with n vertices and c cycles. This is done by generating the cycles iteratively. The average number of vertices per cycle is n/c. To get a larger amount of possible graphs the number of vertices per cycle is randomly distributed around the average n/c. A Poisson distribution turned out to yield a higher variety of graphs than a uniform distribution. To ensure that the correct number of cycles will be achieved, the distribution range is bounded such that, considering cycles already generated, at least one vertex for every remaining cycle is available. The first generated cycle is used as the base graph. Each consecutive cycle must additionally use an existing vertex to connect to the base graph. This vertex is chosen uniformly among existing vertices.

A.2 Graph with given Cactus Graph

Given a cactus graph C, one might ask how a graph G of which C represents all minimum cuts could look like. Trivially G could be equal to C. Different graphs could be constructed by reversing the process of edge contractions during the computation of a cactus graph. In particular, each vertex of the cactus graph C could be replaced by a dense subgraph. Let k be the desired connectivity. Then, each vertex can be replaced by an at least $(k+1)$-connected subgraph while each link is replaced by k unweighted links in case of a tree edge or by $k/2$ links in case of a cycle edge between corresponding dense subgraphs. However, for all algorithms considered in this paper, neither the structure of the original graph nor the connectivity k matter as they are abstracted in preprocessing steps. Therefore, only the simplest case of cactus graphs with connectivity $k = 2$ is considered.

B Omitted Proofs

▶ **Theorem** (3). *Let $C = (V_c, E_c)$ be a cactus graph and $C_L = (V_c, L_c)$ the link graph of C, such that a feasible solution to the weighted connectivity augmentation problem exists. Then, a minimum spanning forest $L_{MST} \subset L_c$ for C_L is a feasible solution to the connectivity augmentation problem on C.*

Proof. If the graph C_L is connected, L_{MST} is a tree. In this case there is a path in C_L between any two vertices and therefore every cut in C is covered. Therefore, L_{MST} is a feasible solution to the weighted connectivity augmentation problem. Assume C_L is not connected and L_{MST} is not a feasible solution to the WCAP, i.e. the connectivity of $\tilde{C} = (V_c, E_c \cup L_{MST})$ is not increased compared to the connectivity of C. Then, there must be a cut c which is not covered by the links in L_{MST}. However, since there exists a solution to the problem there must be a link $l = (u, v)$ in C_L covering the cut c. Since L_{MST} is a minimum spanning forest

u and v must be in the same connected component. Therefore, there is a path between u and v in L_{MST} which results in c being covered. This forms a contradiction and therefore L_{MST} is a solution to the weighted connectivity augmentation problem. ◀

▶ **Theorem (4).** *Let $G = (V, E)$ be a k-connected graph and $C = (V_c, E_c)$ its cactus graph representation, $L' \subset L$ an augmentation on the link graph C_L and $l \in L'$. If $|L'| < |V_c|$, then we can check if $L' \setminus \{l\}$ is still an augmentation in $\mathcal{O}(|V_c|)$.*

Proof. A link $l = (u, v) \in L'$ can be removed, i.e. is disposable, if $L' \setminus \{l\}$ is still an augmentation. Therefore, we check the connectivity between u and v in the graph $C' = (V_c, E_c \cup L' \setminus \{l\})$ using a maximum flow algorithm. A link $l = (u, v)$ can be removed, if the u-v-flow and therefore the minimum u-v-cut in C' is larger than k. Let $m = |E_c \cup L'|$. To compute a maximum u-v-flow we use Ford-Fulkerson with complexity $\mathcal{O}(m \cdot f)$, where f is the flow value [9]. We can improve this factor by initially computing augmenting paths only using edges from the cactus graph. Since these edge weights are in $\{\frac{k}{2}, k\}$, there can be at most two rounds of augmenting paths, each with weight at least $\frac{k}{2}$. Therefore, this can be done in $\mathcal{O}(|V_c|)$ time. Afterwards, to determine whether the u-v-connectivity in C' is larger than k, we only need to do one iteration of Ford-Fulkerson which takes $\mathcal{O}(m)$ time. As $|L'| < |V_c|$ and E_c are the edges of a cactus graph, it holds $m \in \mathcal{O}(|V_c|)$ and we can therefore determine if a link $l = (u, v)$ is disposable within $\mathcal{O}(|V_c|)$. ◀

▶ **Theorem (5).** *LS(k) has a complexity of $\mathcal{O}(|V_c|^2(|V_c|^{\lfloor \frac{k}{2} \rfloor} \times \Delta^2 + |V_c|^{\lceil \frac{k}{2} \rceil}))$.*

Proof. The running time for LS(k) is dominated by the while loop in Algorithm 3. We have at most $|V_c|$ re-computations of the set X. Therefore, we need to check validity for at most $|V_c| \times |X|$ swaps. This results in a running time of $\mathbb{O}(|V_c|^2 \times |X|)$. To estimate $|X|$ we only consider the swaps using exactly k links, since these are the dominating factor. Each path of length exactly k has to have at least $k_1 = \lfloor \frac{k}{2} \rfloor$ and at most $k_2 = \lceil \frac{k}{2} \rceil$ edges from the current solution S. There can be $\mathcal{O}(|V_c|^{k_1} + |V_c|^{k_2})$ swaps in X using different links from S. Now we estimate the number of different paths using the same links from S. Let k be odd and there are k_2 equal solution links in the path. Then, there can only be one such path, since the endpoints are fixed. If there are only k_1 equal solution links in the path, then both endpoints can vary. This results in at most Δ^2 different paths, where Δ is the max degree in $(V_c, L' \cup S)$. When k is even, one endpoint of the path is free and there are at most Δ different paths. Overall, this yields $|X| \in \mathcal{O}(|V_c|^{\lfloor \frac{k}{2} \rfloor} \times \Delta^2 + |V_c|^{\lceil \frac{k}{2} \rceil})$ and results in a complexity for LS(k) of $\mathcal{O}(|V_c|^2(|V_c|^{\lfloor \frac{k}{2} \rfloor} \times \Delta^2 + |V_c|^{\lceil \frac{k}{2} \rceil}))$. ◀

C Instance Details

Table 1 Properties of all graphs G and their corresponding cactus graph C used in the evaluation.

class	graph	$\|V(C)\|$	$\|E(C)\|$	$\|V(G)\|$	$\|E(G)\|$
Real-World	coAuthorsCiteseer	30 322	30 321	227 320	814 134
	preferentialAttachment	28 530	28 529	100 000	499 985
	delaunay_n21	23 719	23 718	2 097 152	6 291 408
	luxembourg.osm	23 077	23 076	114 599	119 666
	kkt_power	22 388	22 387	2 063 494	6 482 320
	delaunay_n20	11 740	11 739	1 048 576	3 145 686
	coPapersDBLP	10 244	10 243	540 486	15 245 729
	as-22july06	7 999	7 998	22 963	48 436
	hugetric-00000	6 617	8 040	5 824 554	8 733 523
	coPapersCiteseer	6 372	6 371	434 102	16 036 720
	delaunay_n19	5 977	5 976	524 288	1 572 823
	PGPgiantcompo	5 513	5 512	10 680	24 316
	vsp_vibrobox_scagr7-2c_rlfddd	3 956	3 955	77 328	435 586
	vsp_finan512_scagr7-2c_rlfddd	3 936	3 935	139 752	552 020
	vsp_sctap1-2b_and_seymourl	3 288	3 287	40 174	140 831
	finan512	3 073	3 072	74 752	261 120
	delaunay_n18	2 929	2 928	262 144	786 396
	vsp_south31_slptsk	2 710	2 709	39 668	189 914
	vsp_model1_crew1_cr42_south31	2 561	2 560	45 101	189 976
	vsp_c-30_data_data	1 768	1 767	11 023	62 184
	power	1 612	1 611	4 941	6 594
	delaunay_n17	1 484	1 483	131 072	393 176
	af_shell9	1 276	1 275	504 855	8 542 010
	vsp_bump2_e18_aa01_model1_crew1	1 212	1 211	56 438	300 801
	t60k	1 136	1 332	60 005	89 440
	vsp_p0291_seymourl_iiasa	942	941	10 498	53 868
	ldoor	904	903	952 203	22 785 136
	latin_square_10	901	900	900	307 350
	delaunay_n16	790	789	65 536	196 575
	add32	681	680	4 960	9 462
	vibrobox	625	624	12 328	165 250
	add20	367	366	2 395	7 462
	delaunay_n15	359	358	32 768	98 274
	NLR	336	335	4 163 763	12 487 976
	G3_circuit	317	316	1 585 478	3 037 674
	vsp_befref_fxm_2_4_air02	215	214	14 109	98 224
	delaunay_n14	181	180	16 384	49 122
	audikw1	163	162	943 695	38 354 076
	email	156	155	1 133	5 451
	uk	136	135	4 824	6 837
	M6	132	131	3 501 776	10 501 936
	cage15	129	128	5 154 859	47 022 346
	memplus	110	109	17 758	54 196
Generated Cycles	cycle-5000	5 000	5 000		
	cycle-1000	1 000	1 000		
	cycle-500	500	500		
	cycle-300	300	300		
	cycle-200	200	200		
	cycle-100	100	100		
	cycle-50	50	50		
Generated Stars	star-5000	5 000	4 999		
	star-1000	1 000	999		
	star-500	500	499		
	star-300	300	300		
	star-200	200	199		
	star-100	100	99		
	star-50	50	49		
Generated Cacti	cactus{16-20}	1000	1199		
	cactus{11-15}	200	279		
	cactus{06-10}	100	119		
	cactus{01-05}	100	109		