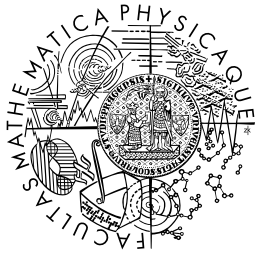

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Pavel Madaj

Heuristics for Length Bounded Cuts

Department of Applied Mathematics

Supervisor of the master thesis: doc. Mgr. Petr Kolman, Ph.D.

Study programme: Computer Science - Discrete Models and Algorithms

Study branch: IDMP

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

This thesis is dedicated to my family, friends and my thesis supervisor doc. Mgr. Petr Kolman, Ph.D for their support and encouragement. Without them this thesis would not have been possible.

Title: Heuristics for Length Bounded Cuts

Author: Bc. Pavel Madaj

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Petr Kolman, Ph.D., Department of Applied Mathematics

Abstract: This thesis deals with the problem of finding a minimum length-bounded cut in a graph. We first provide a brief overview of the problem and its applications. We then discuss the known theoretical results and approximation algorithms. We look at the existing linear programming formulations and propose a new one. A concise discussion on potential hard instances, utilized for testing our formulations, is also incorporated. The focus of our analysis is on the performance and behavior of our proposed linear programming family, contrasting it with the established natural formulation. We also compare the performance of various heuristics and approximation algorithms in practice by examining their behaviour on a large set of small instances.

Keywords: graph theory, approximation algorithms, length bounded cuts, linear programming, heuristics

Contents

Introduction

The problem of finding the minimum s-t-cut in a graph is one of the classic problems in computer science. Given a graph G and vertices s, t, we are interested in finding the least number of edges that, when removed, interrupt all paths from s to t. There are many applications for the known efficient algorithms solving this problem. One straightforward application, for example, is checking whether a network (road network, cable network, etc.) is robust to local failures. A small s-t-cut means that a few local issues could completely disconnect parts of the network. However, in practice knowing that there is no small cut does not necessarily mean the network is well-designed. One could easily imagine a situation where a car accident on a segment of a road does not interrupt the flow of traffic entirely, but the detours created are prohibitively long. Thus, one may want to study cuts which might not disconnect the two vertices entirely but that increase the distance between them above some constant L. For this purpose the concept of an L-bounded cut has been introduced. However, unlike the tractable problem of a minimal s-t-cut the problem of an L-bounded s-t-cut is not efficiently solvable unless $\mathbf{P} = \mathbf{NP}$. Moreover, current approximation algorithms for this problem have poor approximation ratios. In this thesis we explore a possible new approach to the approximation algorithms of the L-bounded cut problem and also compare various heuristics and approximations in practice.

Preliminaries

In this section, we provide definitions for essential concepts used throughout this thesis.

Definition 1 (Graph)**.** A *graph* G is an ordered pair $G := (V, E)$ comprising a set V of vertices or nodes together with a set E of edges. Each edge is a 2-element subset of V. We will commonly denote the number of vertices of a graph by n and the number of edges by m.

An *edge capacity function* is $c\colon G(E) \to \mathbb{Q}_0^+$. Unless stated otherwise a graph will have all edges with unit capacities ($\forall e \in E(G) : c(e) = 1$).

Definition 2 (Length Bounded Cut Instance)**.** An *instance of the length bounded cut problem* is (G, s, t, L) where G is a graph, $s, t \in V(G)$ and $L \in \mathbb{N}$. The vertices s and t are commonly referred as the *source* and the *sink*.

Definition 3 (s-t-paths)**.** A *path* in graph G is a sequence of distinct vertices $v_1, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $i \in \{1, \ldots, k-1\}$.

A *walk* in graph G is a sequence of vertices $v_1, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $i \in \{1, \ldots, k-1\}$, the vertices nor the edges need to be distinct.

An *s-t-path* in graph G where $s, t \in V(G)$ is a path in G that starts in s and ends in t.

The *length* of a path is the number of edges in the path. The *distance* between s and t is the length of the shortest s-t-path.

We will denote the set of all s-t-paths in G by $\mathcal{P}_{s,t}$ and the set of all s-t-paths of length at most L by $\mathcal{P}_{s,t}^{L}$.

Definition 4 (Cut)**.** Given an instance (G, s, t, L) of the length bounded cut problem :

A *cut* is a set $C \subseteq E(G)$ such that every path between s and t contains an edge of C. $(\forall P \in \mathcal{P}_{s,t} : P \cap C \neq \emptyset)$

An *L-bounded cut* is a set $C \subseteq E(G)$ such that every path between s and t *of length at most* L contains an edge of C. $(\forall P \in \mathcal{P}^L_{s,t} : P \cap C \neq \emptyset)$

The *value* of $C \subset E(G)$ is $\sum_{e \in C} c(e)$. If the graph has unit capacities we will also refer to this quantity as the *size* of the cut.

Definition 5 (Optimization Problem)**.** An *optimization problem* is given by a quadruple (I, f, m, g) where:

- I is the set of instances
- $f : I \to 2^S$ is the solution function that maps an instance to the set of possible solutions for that instance
- $m : I \times S \to \mathbb{R} \cup \{\pm\infty\}$ is the measure function that associates a real value to each instance-solution pair, where S denotes the set of all possible solutions
- $g \in \{\min, \max\}$ is a goal function that indicates whether the problem is one of minimization ($g = \min$) or maximization ($g = \max$)

A solution $s \in f(i)$ is *optimal* for instance $i \in I$ if $m(i, s) = g\{m(i, s') | s' \in f(i)\}$. We will often denote the optimal solution by s^*.

Definition 6 (Approximation Algorithm)**.** An *approximation algorithm* for an optimization problem (I, f, m, g) is an algorithm that, for each instance $i \in I$, produces a solution $s \in f(i)$.

The *approximation ratio* of an approximation algorithm A on an instance i is defined as

$$\rho_A(i) = \frac{m(i, A(i))}{m(i, s^*)}$$

where s^* is an optimal solution to i, and $A(i)$ is the solution produced by algorithm A for instance i.

For a maximization problem, the approximation ratio is defined as

$$\rho_A(i) = \frac{m(i, s^*)}{m(i, A(i))}$$

An algorithm is said to have an approximation ratio of ρ if, for all instances i, $\rho_A(i) \leq \rho$ for minimization problems, or $\rho_A(i) \geq \rho$ for maximization problems.

We will also mention heuristics, which are not defined formally. A heuristic is an algorithm that, like an approximation algorithm, produces a not necessarily optimal solution to an optimization problem. However, unlike an approximation algorithm, a heuristic does not have a guarantee on the quality of the solution it produces. We can therefore think of approximation algorithms as a special case of heuristics for which we can prove a (good) approximation ratio.

Definition 7 (Length Bounded Cut Problem)**.** Given an instance (G, s, t, L) of the length bounded cut problem the *length bounded cut problem* on that instance is an optimization problem of finding the L-bounded cut C with the minimum value.

The *length bounded cut problem* is defined as an optimization problem where the instances are those described above. Feasible solutions for an instance are all L-bounded cuts and the measure of a feasible solution is the size of the cut. The goal is to minimize the size.

Definition 8 (Linear Program)**.** A *linear program (LP)* is an optimization problem in which the objective function to be minimized or maximized, as well as all the constraints, are linear in terms of the decision variables. Formally, a linear program can be represented as:

- Minimization problem: $\min c^T x : Ax \geq b, x \geq 0$.
- Maximization problem: $\max c^T x : Ax \leq b, x \geq 0$.

Here:

- $x = (x_1, \ldots, x_n)^T$ is a vector of decision variables;
- $c = (c_1, \ldots, c_n)^T$ is a vector of coefficients for the objective function;
- A is a $m \times n$ matrix of coefficients for the constraint inequalities;
- $b = (b_1, \ldots, b_m)^T$ is a vector of constants for the constraint inequalities.

If we restrict the decision variables to be integers, we obtain an *integer linear program (ILP)*.

When we express a problem as an ILP we often refer to the LP with the integer constraints removed as the *relaxation* of the ILP. The solutions of a relaxation are called *fractional solutions*.

Definition 9 (Feasible Solution and Optimal Solution)**.** A *feasible solution* to a linear program is a vector x that satisfies all the constraints of the linear program. The set of all feasible solutions is known as the *feasible region* of the linear program.

An *optimal solution* is a feasible solution that minimizes (or maximizes) the objective function.

Definition 10 (Integrality Gap)**.** The *integrality gap* of an optimization problem is the worst-case ratio between the optimal solution of the linear program relaxation and the optimal solution of the original problem. We often express this as a function of the size of the input.

We also sometimes refer to the integrality gap on a specific instance of the problem, which is the ratio between the optimal solution of the linear program relaxation and the optimal solution of the original problem on that instance.

Linear programs are of fundamental importance in optimization theory because they can be solved in polynomial time in the real domain using methods such as the simplex method or interior-point methods.

However, when the decision variables are required to be integers, the problem becomes an *integer linear program (ILP)*, which is a type of problem known to be **NP**-hard. (This means that, unless **P** = **NP**, there is no known algorithm that can solve all instances of integer linear programming in polynomial time.)

Although the asymptotic worst-case complexity of solving ILPs is currently not known to be better than polynomial, in practice there are many effective heuristics and exact methods for solving ILPs efficiently at least for reasonably small instances.

1. Research to Date

Non-length bounded s-t-cuts were studied in a classical article by Menger [12] where he proved the now famous Menger's Theorem: the maximum number of edge-disjoint s-t-paths is equal to the minimum number of edges in an s-t-cut. Later Ford and Fulkerson [5] generalized this result to graphs with capacities and flows. They also provided algorithms for finding the maximum flow and minimum cut in a graph in polynomial time.

The problem of length-bounded flows seems to have been first studied by Adámek and Koubek [1]. They observe that the direct generlization of the min-cut max-flow equivalence does not hold for length-bounded cuts and flows. Independently of that result Lovász et al. proved in [10] studied the behaviour of length-bounded node-disjoint s-t-paths problem. For length bounds up to 4 they proved a result analogous to Menger's theorem.

Later in the year 1982 Itai et al. [9] gave algorithms for finding the maximum number of edge-disjoin s-t-paths with at most two or three edges. They also showed that the same problem is **NP**-hard for length bounds greater than 4.

The main results related to the topic come from [2] where Baier et al. proved that the problem of finding a minimum length-bounded cut for $L \geq 4$ is inapproximable within a factor of 1.377 by a reduction of the vertex cover problem. They also showed an approximation algorithm with approximation ratio of $\mathcal{O}(\min\{L, n^2/L^2, \sqrt{m}\})$, which is $\mathcal{O}(n^{2/3})$ if we want to express it as just a function of n.

Morever the following natural linear program is a direct translation of the problem into a linear program and has been also studied in [2].

$$\begin{aligned} &\min \sum_{e \in E} x(e) \\ &\sum_{e \in p} x(e) \geq 1 \quad \forall p \in \mathcal{P}^L_{s,t} \\ &x(e) \in \{0,1\} \quad \forall e \in E \end{aligned} \tag{1.1}$$

The relaxation of the natural linear program is then the following:

$$\begin{aligned} &\min \sum_{e \in E} x(e) \\ &\sum_{e \in p} x(e) \geq 1 \quad \forall p \in \mathcal{P}^L_{s,t} \\ &x(e) \geq 0 \quad \forall e \in E \end{aligned} \tag{NATLP}$$

2. Explored Instances

In this thesis we wish to examine the behaviour of heuristics, approximation algorithms and linear programs for the length-bounded cut problem on actual graphs. To get data which is complete, we would like to evaluate all graphs on a given number of vertices. However, the number of such graphs rises faster than exponentially, which makes this approach infeasible even for graphs with relatively low number of vertices. For this reason, we choose two approaches. For graphs with small number of vertices we will explore all instances. To also cover the behaviour on larger graphs we will explore a smaller set of instances which we consider to likely be difficult to solve. In this chapter we will discuss the instances we will explore.

The choice of these hard instances is not entirely objective, but we will try to justify our choices and explore alternatives. We will also discuss some general remarks about the instances which we will use in the evaluation.

2.1 General Remarks and Exhaustive Evaluation

There are some general statements we can make to exclude instances which are always unsuitable. One obvious category are graphs which are disconnected. If s and t are in different connected components the problem is solved already. If they are not, then the problem is equivalent to the problem on the connected component containing s and t.

This can be generalized further. Given an instance with a parameter L, if there is an edge which lies on no path of length at most L between s and t, then we can omit this edge from the graph without changing the solution. We shall call such edges *useless*. One approach would be to take the input instance and remove all useless edges and then solve the problem on the resulting graph. However, in the exhaustive evaluation part of our analysis we can just check if an instance has any useless edges and if it does, we can skip it. This is because the instance with the useless edges removed will be evaluated anyway due to the exhaustive nature of the evaluation.

Another obvious category of graphs to skip are those which have no path of length at most L between s and t. In this case the problem is already trivially solved with the empty cut. In the exhaustive evaluation portion of our analysis we cover this by taking a graph G, selecting vertices s and t and then generating instances of the length-bounded cut problem by starting with L equal to the distance between s and t and incrementing it.

Where do we stop this process? We could stop at $L = n - 1$ as there are no paths of length n or more in a graph with n vertices. However, in the vast majority of cases this is unnecessarily wasteful. Note that for $L = n - 1$, the problem is equivalent to the minimum s-t-cut problem as we are asking for a cut which intersects *all* paths between s and t. This problem is solvable on polynomial time for example by the Ford-Fulkerson algorithm[5]. Such value of L can thus be considered trivial, and we could stop at $n - 2$. However, we can

do better.

Observation 11. *Given a graph G, vertices s and t and $L \in \mathbb{N}$ such that for length-bounded cut instance (G, s, t, L) the optimal solution of length-bounded cut has the same size as the optimal solution of the minimum s-t-cut instance (G, s, t) then for all $L' \geq L$, the optimal solution of (G, s, t, L') has the same size as the optimal solution of (G, s, t).*

Proof. Every s-t-cut is also an L-bounded s-t-cut for all L by definition. Therefore, the size of the minimal s-t-cut is an upper bound on the size of the minimal L-bounded s-t-cut. At the same time the minimal L-bounded s-t-cut is a lower bound for the size of the minimal L'-bounded s-t-cut because $\mathcal{P}_{s,t}^{L} \subseteq \mathcal{P}_{s,t}^{L'}$. But since by assumption the size of the minimal s-t-cut is equal to the size of the minimal L-bounded s-t-cut, the size of the minimal L'-bounded cut must also be equal to this value. □

The corollary is that as we increment L once we reach an instance where the optimal solutions of the two problems coincide, we can conclude that for larger L the length-bounded cut would be trivial, and we can stop the process. This is the approach we will use in the exhaustive evaluation. One could make the argument that studying the behaviour of heuristics and linear programs, even on these seemingly trivial instances, could provide additional useful data. However, in the interest of reducing the computational effort required we decided to omit these instances.

In the exhaustive evaluation approach we use Brendan McKay's database of small connected unlabelled graphs [11] that covers graphs up to 11 vertices. This seemingly allows us not to worry about isomorphic graphs because the database already contains only one representative of each isomorphism class. However, note that for each graph G we create $\binom{n}{2}$ triples (G, s, t) and if the graph is not rigid (i.e. has non-trivial automorphisms) then some of these triples will be isomorphic. Thus, in order to avoid duplicates in preprocessing, we use NetworkX's [7] implementation of the vf2 [4] algorithm to filter out isomorphic instances.

When we add up all of this together, we arrive at Algorithm 1, which generates the list of instances on n vertices.

2.2 Random Graphs

One direct approach seems to be to sample among graphs with n vertices at random. In the Erdős–Rényi random graph model we select a probability p and each edge is present with that probability. However, during early experimentation, no matter the choice of p and L, these graphs were rarely interesting enough to be worth exploring. This is further confirmed later in the penultimate chapter where we look at how the ratio of "interesting" instances in the exhaustive evaluation portion is very small. For this reason we decided to omit the random graph approach from the evaluation.

It seems that for an instance to be hard, it needs to have more structure than random graphs are likely to provide. One possible approach, which we did not explore, is to take a random graph and amend it in a way that adds the expected

Algorithm 1 Generate exhaustive list of instances on n vertices

```
instances ← ∅
for all G simple graph on n vertices do
    networks ← ∅
    for all s, t ∈ V(G), s < t do
        if ∃N ∈ networks : (G, s, t) ≅ N then
            continue
        end if
        networks ← networks ∪ {(G, s, t)}
        L ← distance(s, t)
        while |optimal L-bounded cut in G| < |minimum s-t-cut in G| do
            instances ← instances ∪ {(G, s, t, L)}
            L ← L + 1
        end while
    end for
end for
```

structure. For example, one could replace each edge with a small copy of a *camel* graph (explored in the next section) or to attach s and t as new vertices to disjoint subsets of the graph that are some distance apart. However, we did not explore this approach in this thesis.

2.3 Camel Graphs

There are several known sets of instances on which the natural linear program NATLP has a large integrality gap. For example, in [2] we encounter a graph constructed as follows: Given an integer k, we take an s-t-path with $2k+1$ edges. The edges of this path are called the *ground edges*. Parallel to edge ground edge we add $k+$ paths of length 2 to form the final graph. Taken together with parameter $L = 3k + 1$, this forms an example of an instance family for which the integrality gap of the natural representation of the length-bounded edge cut problem and the length-bounded node cut problem is $\Theta(\sqrt{n})$.

However, in practice we saw that (for length-bounedd edge cuts which are the main focus of this thesis) we get clearer results from our approaches by eliminating the $+1$ from all the parameters. The main effect is that some of our linear programs in a later chapter have relaxed optima which are integers or follow ref easily observable arithmetic progressions. These patters are likely still present in the unmodified graph family but are slightly obfuscated by the $+1$ constants. This modification can be easily verified not to affect the asymptotic behaviour of the integrality gap. Let us then formally define this family:

Definition 12 (Camel Graph)**.** For $k \in \mathbb{N}$ a *k-camel graph* is a graph constructed as follows: Given an integer k, we take an s-t-path of length $2k$ and for every edge we add k parallel paths of length 2 to form the final graph. Together with parameter $L = 3k$ this forms the length-bounded cut problem instance.

We argue that due to this family having a large integrality gap on the natural linear program representation of the problem it is a good candidate for exploring

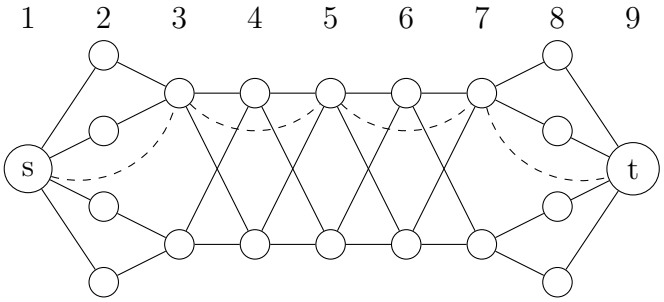

Figure 2.1: A fence graph with $k = 2$. Layer numbers are shown above the graph. The dashed edges are the shortcut edges.

the behaviour of heuristics and approximation algorithms. Moreover, the family is easy to describe, generate and parametrize. We will use this family in the evaluation of the heuristics and linear programs.

It is easy to show that give one "hump" of the camel graph there is little point in its edges being treated differently. For this reason we will also use camel graphs where instead of adding k paths of length two to each ground edge we add just one such path (for each ground edge) and set its weight to k. This is mostly done to reduce the graph size where LP computations get too intensive.

2.4 Fence Graphs

The careful reader might note that while the aforementioned instance family has a large integrality gap for the natural linear program representation, it is not the largest possible. Indeed, as mentioned in the previous chapter, this linear program has an integrality gap of $\Omega\left(n^{\frac{2}{3}}\right)$. This is achieved by the following graph family used in [2] to prove this integrality gap:

Definition 13 (Fence Graph)**.** Given an integer n that satisfies the condition $n^{\frac{1}{3}} \in \mathbb{Z}$, we define $k = n^{\frac{2}{3}}$.

First we create an auxiliary graph G' composed of $4k + 1$ layers.

The first and the last layers of G' each consist of a single vertex, denoted as s and t respectively. The second layer and the penultimate layer consist of $n^{\frac{2}{3}}$ vertices each. All remaining layers contain $n^{\frac{1}{3}}$ vertices each.

Consecutive layers of G' form a complete bipartite graph with the exception of layer pairs $2, 3$ and $4k - 1, 4k$. The vertices of the second layer are divided into $n^{\frac{1}{3}}$ groups of size $n^{\frac{1}{3}}$. Each vertex in a group is linked to a unique vertex from the third layer, distinct from the vertices linked to other groups. The connection pattern between layers $4k - 1$ and $4k$ follows a similar scheme.

To form the fence graph $G = (V, E)$, we select a single vertex v_i from each odd layer i of G'. The vertices v_i and v_{i+2} are linked by an edge for each odd i. These newly added edges are termed as "shortcut" edges.

We shall call this graph G a *fence graph*.

Note that we are sticking to the notation used in **??**, this means that n in this case does not correspond to the exact number of vertices of the fence graph. But it still holds that $|V(G)| = O(n)$.

We shall evaluate our techniques on these fence graphs as they form an obvious hard case due to them (asymptotically) having the largest possible integrality gap for the natural linear program. However, note that the construction here is more complex compared to the camel graphs, and as such the behaviour of methods on these graphs might be less easy to analyse. Also note that as described the size of these graphs grows rapidly since we require that $n^{\frac{1}{3}}$ is an integer. It could be interesting to explore the behaviour of methods on a modified version of this family where we relax this requirement and instead allow any integral n and round k up or down. However, we did not explore this in this thesis.

2.5 Recursive Camels

One last family of graphs we explore are an attempt to augment the camel graphs with additional structure. Given parameters $k, r \in \mathbb{N}$ we set G_0 to be an s-t-path with k edges and $K_0 = E(G_0)$. Then for $i = 1, 2, \ldots, r$, we create G_i from G_{i-1} by taking every edge in K_{i-1} and adding one path of length 2 parallel to it. We form K_i as the set of all edges added in this step. The final graph is the *r-depth recursive k-camel graph*. For simplicity, in our analysis we restrict this class of graphs to just those where $r = k$ and set $L = \lceil \frac{3}{2}k \rceil + 1$, as that is where the most complexity seemed to arise. figure

2.6 Reduction of Vertex Cover

Finally, there is one family of graphs that could look enticing at a glance but quickly proves not to be useful. In [2] Theorem 3.9 the **NP**-hardness of the length-bounded cut problem and **NP**-hardness of its approximation within a factor of 1.1377 is proven by a reduction from the Vertex Cover problem.

In the Vertex Cover problem we are given a graph G and an integer k and we are asked if there exists a set of at most k vertices $S \subseteq V(G)$ such that every edge of G has at least one endpoint in S. It is well known that this problem is **NP**-complete.

This problem has a natural linear program representation which has an integrality gap of 2 [14]. It is simple to show that the natural length-bounded cut problem linear program's fractional solutions correspond directly to the fractional solutions of the Vertex Cover linear program. Thus, on this graph family the integrality gap of the natural length-bounded cut problem linear program is also 2. This is far from the general $\Theta(n^{\frac{2}{3}})$ integrality gap. For this reason, we do not consider this instance family as difficult enough to be worth exploring. proof? figure?

3. Linear Programming

3.1 Natural LP

The natural linear program NATLP for the length-bounded cut problem has known integrality gap of $\Theta(n^{\frac{2}{3}})$ as mentioned earlier. This means that, even as a basis for an approximation algorithm, it is not very good. However, we still want to analyse its performance to compare it to other approaches and to also run it in an ILP solver to get the optimal integral solution. There is, however, an issue because in its standard form the natural LP has an exponential number of constraints. We will now describe a way to reduce the number of constraints to a polynomial amount.

In order to do this, we do not constraint each L-bounded s-t-path on its own, but instead we introduce new auxiliary variables. The x variables stay as they are in the natural LP — for each edge they indicate whether it is a part of the cut or not. On top of those we add y variables constrained in a way that, for a vertex v and $i \in \{1, 2, \ldots, L\}$, make $y(v, i) = \sum_{e \in \mathcal{P}^i_{s,v}} x(e)$. In order to do this, we need to define when such a (v, i) pair (representing vertex v at distance i from s) is valid to be a part of an actual s-t-path of length at most L.

Definition 14. Let G be a graph, $s, t \in V(G)$ and $L \in \mathbb{N}$. We say that given $v \in V(G)$ and $i \in \mathbb{N}$ a pair (v, i) is *valid* if $\text{dist}(s, v) \leq i$ and $\text{dist}(s, t) \leq L - i$.

Then we can write the following linear program:

$$\min \sum_{e \in E} x(e) \qquad \text{(NATLP-LAYERED)}$$

$$y(v, i) \leq y(u, i-1) + x(uv) \quad \forall uv \in E(G) \text{ such that } (v, i), (u, i-1) \text{ are valid}$$

$$y(t, i) \geq 1 \quad \forall i \in \{1, 2, \ldots, L\} \text{ such that } (t, i) \text{ is valid}$$

$$y(s, 0) = 0$$

$$x(e) \geq 0 \quad \forall e \in E$$

$$y(v, i) \geq 0 \quad \forall (v, i) \text{ valid}$$

With the notational caveat that in the first constraint, we consider both uv and vu to be edges.

Theorem 15. *Linear program NATLP-LAYERED is equivalent to the natural linear program NATLP. Specifically, for any feasible solution to NATLP-LAYERED we can construct a feasible solution to NATLP with the same objective value and for any feasible solution to NATLP we can construct a feasible solution to NATLP-LAYERED with the same objective value.*

Proof. Given a feasible solution x, y to NATLP-LAYERED we can construct a feasible solution x' to NATLP by setting $x'(e) = x(e)$ for all $e \in E$.

The basic constraint $x'(e) \geq 0$ for all $e \in E$ is satisfied because the same condition holds for x.

Now let us be given an s-t path $P = \{s = v_1, v_2, \ldots, v_k = t\} \in \mathcal{P}^L_{s,t}$. By the constraints in NATLP-LAYERED we have that $y(v_k, k) \geq 1$. But then we have:

$$
\begin{aligned}
1 &\leq y(v_k, k) \\
&\leq y(v_{k-1}, k-1) + x(v_{k-1}v_k) \\
&\leq y(v_{k-2}, k-2) + x(v_{k-2}v_{k-1}) + x(v_{k-1}v_k) \\
&\dots \\
&\leq \sum_{i=1}^{k-1} x(v_i v_{i+1}) \\
&= \sum_{e \in P} x(e)
\end{aligned}
$$

Therefore, x' is a feasible solution to NATLP. Objective values in both linear programs are the same, because x and x' are the same and both programs define the objective value as the sum of $x(e)$ over all $e \in E$.

For the other direction let us be given a feasible solution x' to NATLP. We can construct a feasible solution x, y to NATLP-LAYERED by setting $x(e) = x'(e)$ for all $e \in E$ and $y(v, i) = \min\{\sum_{e \in P} x(e) \colon P \in \mathcal{P}^i_{s,v}\}$ for all (v, i) valid.

The constraints $y(s, 0) = 0$, $x(e) \geq 0$ for all $e \in E$ and $y(v, i) \geq 0$ are satisfied by the construction trivially. Since in the non-layered linear program we know that every s-t-path that has length at most L has the sum of x over its edges at least 1, we can also conclude that the $y(t, i) \geq 1$ condition is satisfied.

In order to see why the last constraint is upheld, let us have $y(v, i)$ and an edge uv. We need to show that $y(v, i) \leq y(u, i-1) + x(uv)$. We know that there exists a path $P' \in \mathcal{P}^{i-1}_{s,u}$ such that $\sum_{e \in P} x(e) = y(u, i-1)$. Then we can construct a path $P \in \mathcal{P}^i_{s,v}$ by adding uv to P'. Therefore, $\sum_{e \in P} x(e) = y(u, i-1) + x(uv)$. Since $y(v, i)$ is defined as the minimum of such sums over all paths $P \in \mathcal{P}^i_{s,v}$ we have that $y(v, i) \leq y(u, i-1) + x(uv)$.

Finally, the objective values are the same because x and x' are the same and both programs define the objective value as the sum of $x(e)$ over all $e \in E$. □

Note that unlike the original linear program, this one has a polynomial number of constraints $\Theta(nL + m)$. So it is more useful for practical purposes.

3.2 A New Approach

The natural linear program and its layered variant are inspired by having some variables indicate which edges are in the cut C and then directly constrain the distance of s and t in the graph $G \setminus C$. However, the integrality gap of this approach is quite large. One could argue that this is because the variables are not constrained well enough to represent cuts. In order to remedy this, we can try to capture not only the distances of s and t in $G \setminus C$ but also the distances of all other pairs of vertices. This is the approach taken by the following linear program and its variants. The variable $x(u, v, i)$ is meant to indicate that the distance of u and v in $G \setminus C$ is exactly i with the exception of value $L + 1$ which is meant to indicate that the distance is at least $L + 1$. This is because longer distances are not relevant to the problem. We provide the linear program in its relaxed form, the ILP is obtained naturally by constraining all variables to be integral (or specifically in the $\{0, 1\}$ domain).

$$\min \sum_{e \in E} y(e)$$

$$y(uv) = \sum_{i=2}^{L} x(u,v,i) \quad \forall uv \in E \tag{COST}$$

$$x(s,t,L+1) = 1 \tag{STDIST}$$

$$\sum_{i=1}^{L+1} x(u,v,i) = 1 \quad \forall uv \in E \tag{UNIQUEDIST}$$

$$\sum_{i=b+c+1}^{L+1} x(u,w,i) \leq \sum_{i=b+1}^{L+1} x(u,v,i) + \sum_{i=c+1}^{L+1} x(v,w,i) \tag{CUTDIST}$$

$$\forall u,v,w \in V \text{ pairwise distinct}; b,c \in \{1,\ldots,L+1\}; b+c < L+1$$

$$\sum_{i=1}^{L+1} ix(u,w,i) \leq \sum_{i=1}^{L+1} ix(u,v,i) + \sum_{i=1}^{L+1} ix(v,w,i) \tag{LPDIST}$$

$$\forall u,v,w \in V \text{ pairwise distinct}$$

$$x(u,v,i) = 0 \quad \forall u,v \in V, i \in \{1,\ldots,\text{dist}(u,v)-1\} \tag{ZERO}$$

$$y(e) \geq 0 \quad \forall e \in E$$

$$x(u,v,i) \geq 0 \quad \forall u,v \in V, u < v, i \in \{1,2,\ldots,L+1\}$$

This linear program tries to capture the metric space nature of the distances in the graph $G \setminus C$. First note that the variables y exist mostly just for convenience. The COST rule just defines them as shortcuts for how much weight is placed on the edge distance in $G \setminus C$ being longer than 1. The STDIST rule is the one that actually captures the distance of s and t in $G \setminus C$. The UNIQUEDIST rule is there to ensure that each pair of vertices has a unique distance in $G \setminus C$ in the integral case. The rules mentioned so far are essential and will always be present. The remaining named rules will be explored in a variety of combinations later on. The CUTDIST and LPDIST rules are the ones that actually capture the distances.

In the integral interpretation the CUTDIST rule says that if the distance of u and w is greater than $b+c$ then it must be the case that the distance of u and v is greater than b or the distance of v and w is greater than c. This can be interpreted as an "implication" based form of a triangle inequality. The LPDIST rule also captures a triangle inequality but this time it uses sums of x variables to get the distances between u, v and w directly, hence the name "LP dist", as it uses the distances that the linear program tells us about. Finally the ZERO rule restricts us from any vertex distances shortening below what they are initially in G.

However, this linear program can be tweaked further and we explored multiple combinations of constraints. One variation is one in which we interpret the x variables differently. Instead of having them indicate the exact distance between two vertices, we assume that vertices are split into layers based on the distances from s. Then we have $x(u,v,i)$ indicate that u and v are in layers which are i apart. However, in order to do this the LP needs to be augmented by removing the ZERO rule, because now vertices can go closer together as we no longer interpret

$x(u, v, i)$ as the actual distance. Moreover, since multiple vertices could be in each layer we need to extend the range of i to be $\{0, 1, \ldots, L\}$ instead of $\{1, 2, \ldots, L+1\}$. This modification propagates straightforwardly to the other rules as well, for the sake of brevity we omit the modified rules. However, approaches inspired by this interpretation seem less useful than those where x captures actual distances, because the approach of placing vertices in layers based on distance from s is already captured by the NATLP linear program. In the following sections we stick to the convention that if the ZERO rule is present then indices i are in the domain $\{1, 2, \ldots, L+1\}$ and if it is not then they are in the domain $\{0, 1, \ldots, L\}$. With this layer-based interpretation one could also introduce additional variables $z(u, i)$ which would indicate that u is in layer i. We briefly explored this approach but found it not to be of much use. For the sake of completeness, we include the rules for the z variables as well, but they will not be explored in the rest of the thesis:

$$\sum_{i=0}^{L+1} z(u,i) = 1 \quad \forall u \in V$$

$$\sum_{i=0}^{L+1} iz(u,i) - \sum_{i=0}^{L+1} iz(v,i) \leq \sum_{i=0}^{L+1} ix(u,v,i) \quad \forall uv \in E$$

$$\sum_{i=0}^{L+1} iz(v,i) - \sum_{i=0}^{L+1} iz(u,i) \leq \sum_{i=0}^{L+1} ix(u,v,i) \quad \forall uv \in E$$

$$z(s,0) = 1$$

$$z(t, L+1) = 1$$

$$z(u,0) = 0 \quad \forall u \in V \setminus \{s\}$$

$$z(u,i) \geq 0 \quad \forall u \in V, i \in \{0, 1, \ldots, L+1\}$$

If we stick with the ZERO rule being present, we see that it narrows down the solution space by disabling the distance indices which are too low. During the exploration of the various LPs we arrived at situations where there seemed to be "too much space" in the upper part of this index domain which led to harder to interpret results, as there were many solutions with the same objective value. In order to remedy this we tried to add constraints which would restrict the solution space in the upper part of the index domain. To achieve this, let us quickly look at the interpretation of x. In theory, we could have $x(u, v, i)$ to be an indicator for the distance of u and v being i exactly for any i. Due to the nature of the problem, we do not need to do this as distances longer than L are all the same to us. Therefore, we essentially squash indices i greater than L into a single value. This is a global operation based purely on the value of L, but we can do better.

Given a pair of vertices u, v let us define $g(u, v) = L - \text{dist}(s, u) + \text{dist}(t, v) + 1$. Let us now assume that cut C increases the distance of u, v to at least $g(u, v)$. If there was a short path $\{s, \ldots, u, \ldots, v, \ldots, t\}$ in G then after the removal of the cut the three segments of the path have distances at least $\text{dist}(s, u) + g(u, v) + \text{dist}(v, t) = L + 1$, which makes this path no longer short. Therefore, we could do the "index squashing", not just for indices larger than L, but for indices larger than $g(u, v)$. To do this correctly we need to also account for paths where u and v are met in the opposite order, so this rule is actually:

$$x(u,v,i) = 0 \quad \forall u,v \in V, i > \max\{g(u,v), g(v,u), \text{dist}(u,v)\} \quad \text{(ZEROUPPER)}$$

The final addition of $\text{dist}(u,v)$ is to prevent the combination of ZERO and ZEROUPPER from eliminating the full domain. This tends to only happen when the vertex pair is not particularly useful in the first place, so it is more of a technical detail. With this rule included, the index value $\max\{g(u,v), g(v,u), \text{dist}(u,v)\}$ stands for distances greater than this value too.

Finally, we can look at the LPDIST rule and again consider the previously mentioned concept of "index squashing". Instead of applying this rule just once for each vertex pair, we can apply it multiple times, each time squashing a longer range of upper indices. This is the approach taken by the following rule:

$$\begin{aligned} &\sum_{i=1}^{k-1} ix(u,w,i) + k\sum_{i=k}^{L+1} x(u,w,i) \\ &\leq \sum_{i=1}^{k-1} ix(u,v,i) + k\sum_{i=k}^{L+1} x(u,v,i) \\ &+ \sum_{i=1}^{k-1} ix(v,w,i) + k\sum_{i=k}^{L+1} x(v,w,i) \\ &\forall u,v,w \in V \text{ pairwise distinct}, k \in \{1,\ldots,L+1\} \end{aligned} \quad \text{(EXTDIST)}$$

Note that while the number of constraints for a given instance of the LP is polynomial, this polynomial can grow relatively large. Specifically for CUTDIST the number of constraints is $\Theta(n^3L^2)$. This poses no theoretical issue, however, it can be a practical problem as LP / ILP solvers can struggle with large number of constraints when it comes to computational time or even memory.

Given the integral linear program (i.e. one in which we constrain all variables to be integral, or more specifically in the $\{0,1\}$ domain), if the CUTDIST rule is present, it is easy to prove that under the distance interpretation of x any feasible solution to the ILP forms a metric space. From this it follows that the set of edges for which y is 1 forms an L-bounded cut. However, for comparison we will also briefly explore the behaviour of linear programs which do not have the CUTDIST rule. In those cases such a property need not hold and as such we must be careful not to, for example, take the optimum of the relaxation and round it up expecting to get an L-bounded cut.

4. Heuristics

In addition to exploring linear programs we also look at the behaviour of various heuristic approaches to the length bounded cut problem.

4.1 Delete Shortest

The first heuristic we consider is the *delete shortest* heuristic. This one is very simple: we repeatedly delete the shortest path between s and t until the shortest path is longer than L. Brief pseudocode is given in Algorithm 2.

Algorithm 2 Delete Shortest

```
1: procedure DeleteShortest(G, s, t, L)
2:     G' ← G
3:     deleted ← ∅
4:     while dist(s, t) ≤ L do
5:         P ← shortestPath(G', s, t)
6:         G' ← G' \ P
7:         deleted ← deleted ∪ P
8:     end while
9:     return deleted
10: end procedure
```

This algorithm runs in time $O(nm)$ since finding the shortest path can be done in linear time and every iteration removes at least one edge from the graph. Despite its simple nature this heuristic can be shown to have approximation ratio of L. Each of the paths we remove is of length at most L and therefore the optimal solution must remove at least on edge from each of them. We remove the whole path which is of length at most L and therefore the size of the solution we find is at most L times the size of the optimal solution.

4.2 Naive Cut

The second heuristic we consider is the *naive cut* heuristic. The idea is to use a polynomial algorithm for the minimum cut problem. Every s-t-cut is also an L-bounded cut, albeit likely not a minimum one. It is easy to see that the gap between these two is generally unbounded. For example if s and t are already further than L apart, then the minimum L-bounded cut is empty, but the minimum s-t-cut can easily be on the order of n. Despite that the heuristic can be surprisingly useful in some cases. Theorem 3.7 in [2] states that the difference between the size of the minimum L-bounded cut an the size of a minimum s-t-cut is at most $\mathcal{O}\left(\frac{n^2}{L^2}\right)$ and if $L \geq \sqrt{m}$ then it is at most $\mathcal{O}(sqrtm)$. Therefore if L is comparatively large the heuristic can be expected to perform well.

In our analysis we use implementation of the Goldberg-Tarjan algorithm for the minimum cut problem from the NetworkX library[7]. That algorithm runs in time $O(n^2\sqrt{m})$.

4.3 Shortpath Cut

A direct refinement of the previous approach to avoid the obvious pitfalls is to take the input graph G' and remove all its useless edged. (As defined previously an edge is useless if it is not on any s-t-path of length at most L.) Then we run the minimum cut algorithm on the resulting graph. We call this heuristic *shortpath cut*. The running time is equivalent to the previous heuristic as pre-calculating distances and filtering the graph can be done in linear time.

4.4 Shortestpath Cut

Another variant on the application of general s-t-cut algorithms is to use the network of the shortest paths. We take the graph G and create G' by only keeping edges that are on some shortest path between s and t. Then we run the minimum cut algorithm on the resulting graph. This increases the distance between s and t by at least one in every iteration. Once this distance is greater than L, we stop. We call this heuristic *shortestpath cut*. Since the number of these iterations is bounded by L, the running time is $O(n^2\sqrt{m}L)$. Brief pseudocode is given in Algorithm 3.

Algorithm 3 Shortestpath Cut

```
1: procedure MakeShortestpathNetwork(G, s, t)
2:     G' ← {V(G), ∅}
3:     for all uv ∈ E(G) do        ▷ Note that distances can be precomputed in
   linear time.
4:         if dist(s, u) + dist(v, t) + 1 = dist(s, t) then
5:             G' ← G' ∪ uv
6:         end if
7:     end for
8:     return G'
9: end procedure
10: procedure ShortestpathCut(G, s, t, L)
11:     deleted ← ∅
12:     while dist(s, t) ≤ L do
13:         G' ← MakeShortestpathNetwork(G', s, t)
14:         C ← minCut(G', s, t)
15:         G ← G \ C
16:         deleted ← deleted ∪ C
17:     end while
18:     return deleted
19: end procedure
```

4.5 $\frac{L}{2}$ Combination

The next heuristic we consider is the $\frac{L}{2}$ *combination* heuristic. The idea is to combine the previous two approaches. We first run the shortestpath cut heuristic

until the distance between s and t is at least $\frac{L}{2}$. Then we run the shortpath cut heuristic on the graph that is left. We run the union of the edges that both approaches removed. We call this heuristic $\frac{L}{2}$ *combination*. The running time is $O(n^2\sqrt{m})$. We expect this approach of combining the two approaches to partially remedy the weaknesses of both as the first one is good for small L and the second one is good for large L.

4.6 Most Shortest Paths Greedy

A class of greedy approaches to the problem is to repeatedly remove edges one by one based on some criterion until the distance between s and t is greater than L. Based on the statement of the problem we could imagine the criterion to be "always pick the edge with the most paths in $\mathcal{P}_{s,t}^L$ going through it". However, this runs into issues as even enumerating all s-t-paths is known to be **#P**-complete [13]. Therefore we need to find a different criterion. We will consider two such criteria.

The first one is to in each step again calculate the network of the shortest paths just like in the shortestpath cut heuristic. In this network we can calculate the number of the shortest paths going through each edge quite easily proceeding in the direction of increasing distance from s. Then we remove the edge with the most of these paths going through it and repeat the process. We call this heuristic *most shortest paths greedy*. The running time is $O((n+m)n)$. However, one could possibly improve this by reusing the shortest path network from the previous iteration. Brief pseudocode is given in Algorithm 4.

4.7 Most Walks Greedy

The second edge-removing greedy approach we consider is to in each step remove the edge with the most walks of length at most L going through it. We call this heuristic *most walks greedy*. Unlike counting paths this can be done in polynomial time by considering an auxiliary directed graph G' in which we replace each vertex v by $L+1$ copies $v_0, \ldots, v_L$ representing the vertices v at distance $0, 1, \ldots, L$ from s on a walk. This is similar to the approach taken in defining the NATLP-LAYERED linear program. For every edge uv and each layer $i \in \{0, 1, \ldots, L\}$ we add directed edges $u_i v_{i+1}$ and $v_i u_{i+1}$ except if the target vertex is t (as there is little point in also counting walks which go through t multiple times). We also add directed edges t_0t_1, t_1t_2, $\ldots$, $t_{L-1}t_L$ to allow for walks of length shorter than L. It is easy to see that every s-t-path of length L in this auxiliary graph corresponds to an s-t-walk of length at most L in the original graph so running the previously defined CalculateShortestPathsGoingThrough procedure and then summing up the path counts for all copies of the edge gives us the number of walks of length at most L going through the edge. The running time is $O((n+m)Ln)$. Brief pseudocode is given in Algorithm 5.

Algorithm 4 Most Shortest Paths Greedy

```
1: procedure MostShortestPathsGreedy(G, s, t, L)
2:     deleted ← ∅
3:     while dist(s, t) ≤ L do
4:         pathCounts ← CalculateShortestPathsGoingThrough(G)
5:         e ← argmax_{e∈E(G')} pathCounts[e]
6:         G ← G \ {e}
7:         deleted ← deleted ∪ {e}
8:     end while
9:     return deleted
10: end procedure
11: procedure CalculateShortestPathsGoingThrough(G)
12:     G ← MakeShortestpathNetwork(G, s, t)
13:     layers ← []
14:     pathCounts ← []
15:     for all v ∈ V(G) do
16:         layers[dist(s, v)] ← layers[dist(s, v)] ∪ {v}
17:         pathCounts[v] ← 0
18:     end for
19:     pathCounts[s] ← 1
20:     for i ← 0 to dist(s, t) − 1 do
21:         for all v ∈ layers[i] do
22:             for all u ∈ neighbours(v) do
23:                 pathCounts[u] ← pathCounts[u] + pathCounts[v]
24:             end for
25:         end for
26:     end for
27:     return paths
28: end procedure
```

Algorithm 5 Most Walks Greedy

```
1: procedure MakeAuxillaryGraph(G)
2:     V' ← ∅
3:     for all v ∈ V(G) do
4:         for i ← 0 to L do
5:             V' ← V' ∪ {v_i}
6:         end for
7:     end for
8:     G' ← {V', ∅}
9:     for i ← 0 to L − 1 do
10:        G' ← G' ∪ t_i t_{i+1}
11:    end for
12:    for all uv ∈ E(G) do
13:        for i ← 0 to L − 1 do
14:            G' ← G' ∪ u_i v_{i+1}
15:            G' ← G' ∪ v_i u_{i+1}
16:        end for
17:    end for
18:    return G'
19: end procedure
20: procedure CalculateWalksGoingThrough(G)
21:    G' ← MakeAuxillaryGraph(G)
22:    pathCounts ← CalculateShortestPathsGoingThrough(G')
23:    walkCounts ← []
24:    for all uv ∈ E(G) do
25:        walkCounts[uv] ← Σ_{i=0}^{L−1} pathCounts[u_i v_{i+1}] + pathCounts[v_i u_{i+1}]
26:    end for
27:    return walkCounts
28: end procedure
29: procedure MostWalksGreedy(G, s, t, L)
30:    deleted ← ∅
31:    while dist(s, t) ≤ L do
32:        walkCounts ← CalculateWalksGoingThrough(G)
33:        e ← argmax_{e∈E(G)} walkCounts[e]
34:        G ← G \ {e}
35:        deleted ← deleted ∪ {e}
36:    end while
37:    return deleted
38: end procedure
```

4.8 Linear Programming Based Heuristics

One final class of heuristics which we can look at are those which are based on linear programs. We can use the linear programs we have defined to find the optimal fractional solution and then find some way to round it to an integral solution. We will consider three such approaches. Following the note at the end of the previous chapter there is little to no reason to use

4.8.1 Round Up

In this approach we take the optimal fractional solution and simply replace all non-zero values of the "is this edge cut" indicator variables with 1 (perhaps a bit confusingly in retrospect these are the x variables in the case of NATLP and NATLP-LAYERED and y values for the other class of linear programs). We call this heuristic *round up*. This is a very simple approach and there is no direct bound on its approximation ratio.

4.8.2 "Smart" Round Up

A direct refinement of the previous approach is not to pick > 0 as the round up condition but instead pick the largest still valid threshold. If we assume that the relaxed linear program assigned variables as weights of how important it is to remove an edge then this approach is equivalent to removing the edges in order of decreasing importance. We call this heuristic *"smart" round up*. We just take the optimal fractional solution, sort edges in descending order of their variable's value and then remove them one by one until the distance between s and t is greater than L. This is equivalent to picking the largest threshold α such that the solution to the linear program is still feasible if we set all variables with value $\geq \alpha$ to 1 and all others to 0. Note that if we can prove that some α always produces a feasible solution then this implies that the approximation ratio here is at most $\frac{1}{\alpha}$ because what we are doing is essentially multiplying the fractional solution by α and then rounding down.

Specifically when it comes to NATLP (and its equivalent layered variant) we can make the following observation:

Observation 16. *Let (G, s, t, L) be an instance of the length bounded cut problem and let x be an optimal fractional solution to the NATLP linear program. Then for any $P \in \mathcal{P}^L_{s,t}$ there exists an edge $e \in P$ such that $x_e \geq \frac{1}{L}$.*

Proof. By the constraints of the LP we have that $\sum_{e \in P} x_e \geq 1$. By the definition of P it is at most L edges long. If all of them had $x_e < \frac{1}{L}$ then $\sum_{e \in P} x_e < 1$ which is a contradiction. □

The above observation implies that $\alpha = \frac{1}{L}$ is a valid threshold for the NATLP linear program. Therefore the approximation ratio of the *"smart" round up* heuristic for NATLP is L.

4.8.3 Largest Weighted Distance

This approach applies only to the alternative class of linear programs we proposed in the previous chapter. We apply similar logic to the previous heuristic but instead of interpreting the fractional y variables as indications of how "important" the edge is for a cut we now have access to x variables which tell us about the distance. It is reasonable to expect that if uv was an edge in the input graph, but the fractional solution tells us that these two vertices are supposed to be distant then it is likely to be a good candidate for a cut.

Therefore, for each edge uv we assign it $f(uv) = \sum_{i=1}^{L+1} i x_{uv}^i$ which and proceed as before by removing edges in order of decreasing f value. We call this heuristic *largest weighted distance*. Contrast this with the previous approach which is equivalent to assigning $f(uv) = y(uv) = \sum_{i=2}^{L+1} x_{uv}^i$. One could imagine a spectrum of in-between approaches which use index weights different from $\{0, 1, 1, 1, \ldots, 1\}$ and $\{1, 2, 3, \ldots, L+1\}$ for these sums.

5. Results

In this chapter we look at the results of our analysis of the heuristics and linear programs on real data.

5.1 Camel Graphs

We evaluated the linear programs on the non-weighted variant of the camel graphs up to $k = 3$. For larger values of k the linear programs were too slow to solve in a reasonable amount of time. However, the results for the weighted and non-weighted variants are equal for these small k values just like one would expect based on the construction and the properties of the graph. Let us then focus just on the weighted variant of these graphs.

Results of the linear program analysis are shown in Table 5.1. Each column corresponds to a weighted camel graph with k being equal to that column's header. Row L corresponds to the value of the parameter L of the instance. Rows n and m correspond to the number of vertices and edges of the instance respectively. Row OPT corresponds to the optimal value of the instance (calculated by running an ILP solver on the integral version of **??**). The rest of the rows correspond to the fractional optima of the linear programs. Each row is named after the constraints we include in addition to the ones we described as always present.

We are looking for programs which have the smallest integrality gap as that is a good sign that the linear program describes the problem in question well. Programs with small integrality gaps are often used to then create approximation algorithms with approximation ratios which are derived from the integrality gap. Therefore, generally the larger the values in a row the better the program is.

The results paint a fairly clear picture when it comes to some programs. If the CUTDIST rule is omitted, we get results we get programs that have integrality gaps worse than even the natural linear program LPNAT. This is unsurprising as without the CUTDIST rule we have no reason to believe that the integral solutions of the program even correspond to L-bounded cuts. The CUTDIST rule is also the one composed of the most constraints, so it makes sense that leaving it out would have a large impact on the quality of the program.

However, interestingly enough despite the large number of constraints in the CUTDIST rule it seems that the rule on its own has very little strength too. Specifically CUTDIST, CUTDIST ZERO and CUTDIST ZERO ZEROUPPER all seem to have the exact same fractional optima as LPNAT(-LAYERED). We conjecture that this pattern holds in general for any instance of the problem.

During a previous stage of this analysis devised the ZEROUPPER rule in order to constrain the behaviour of x variables a bit more in the CUTDIST ZERO scenario. It seemed that in the CUTDIST ZERO program the x variables of the optimum form a nice generalizable pattern. However, in some cases this pattern was somewhat broken likely because of there being multiple optimal solutions. The ZEROUPPER rule was supposed to fix this, and we did not expect it to change the objective value. And indeed for the program in question (CUTDIST ZERO ZEROUPPER) this was the case. A surprising side effect was that adding

the ZEROUPPER rule to CUTDIST LPDIST ZERO program actually improved the fractional optima to the level of integral optima resulting in integrality gap of 1 for these specific instances. This was an unexpected result as we did not expect the ZEROUPPER rule to have any effect on the LPDIST part of the program.

We conjecture that this pattern of CUTDIST LPDIST ZERO ZEROUPPER achieving integrality gap of 1 holds for all instances of the weighted camel graphs. However, we were unable to prove this conjecture. As we will see in the next section, there are certainly other instances in which this is not the case.

The EXTDIST rule seems to help less than we would have expected. When only CUTDIST LPDIST and ZERO are present it helps a small bit but in the CUTDIST LPDIST ZERO ZEROUPPER case the optimal integrality gap 1 is achieved even without it. Thus, it seems that either this approach of extending LPDIST into EXTDIST is not very useful or that it needs further refinement.

5.2 Fence Graphs

As mentioned in the definition of fence graphs, we are sticking to the original definition in [2] where it is required that $n^{\frac{1}{3}}$ is an integer. This means that the number of vertices of successive fence graph instances grows quickly. Compounded by the fact that the CUTDIST rule is composed of $O(n^3L^2)$ constraints this meant that with computational resources available to us, we were only able to examine the behaviour of the linear programs for fence graphs with $n \in \{1, 8\}$. Of these the $n = 1$ case is trivial as it is composed of just five vertices. So let us focus on the $n = 8$ case. The results re shown in Table 5.2.

Results here paint a less clear picture. None of the linear program variants achieve approximation ratio of 1. It is what we would expect on difficult instances as the problem is known not to be approximable within a factor of 1.1377, so in a way this result partially validates the choice of fence graphs as hard instances. As we would expect the smallest integrality gap is reached when all LP rules are enabled. This results in integrality gap of approximately 1.25737. The other notable result is that CUTDIST without LPDIST still reaches the same fractional optimum as LPNAT-LAYERED.

Here we see the EXTDIST rule help even in the CUTDIST LPDIST ZERO ZEROUPPER case. However, the difference is still not very large.

5.3 Recursive Camel Graphs

Recursive camel graphs as we defined them also struggle from quickly growing number of vertices. Here the growth is exponential. However, despite that fact we were able to examine the behaviour of the linear programs for $k \in \{1, 2, 3\}$. The results are shown in Table 5.3.

Results here support the previous observations. Lacking LPDIST the linear programs still behave as poorly as LPNAT(-LAYERED) and the smallest integrality gap is now approximately 1.31250. However, unlike the fence graph instances here we once again see that the addition of the EXTDIST rule does not improve the integrality gap when the other rules are present.

Table 5.1: Weighted Camel Data

Parameter	**Weighted Camel**						
	1	**2**	**3**	**4**	**5**	**6**	**7**
L	4	7	10	13	16	19	22
n	7	11	15	19	23	27	31
m	9	15	21	27	33	39	45
OPT	2	3	4	5	6	7	8
LPNAT-LAYERED	1.5	1.66667	1.75	1.8	1.83333	1.85714	1.875
CUTDIST	1.5	1.66667	1.75	1.8	1.83333	1.85714	1.875
CUTDIST ZERO	1.5	1.66667	1.75	1.8	1.83333	1.85714	1.875
CUTDIST ZERO ZEROUPPER	1.5	1.66667	1.75	1.8	1.83333	1.85714	1.875
LPDIST	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
LPDIST ZERO	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
LPDIST ZERO ZEROUPPER	0.5	0.666667	0.75	0.8	0.833333	0.857143	0.875
CUTDIST LPDIST	1.875	2.31034	2.69355	3.06	3.42308	3.77679	4.125
CUTDIST LPDIST ZERO	2	2.38462	2.77826	3.14279	3.4962	3.84366	4.18749
CUTDIST LPDIST ZERO ZEROUPPER	2	3	4	5	6	7	8
CUTDIST LPDIST EXTDIST ZERO	2	2.72727	3.26667	3.78947	4.30435	4.81481	5.32258
CUTDIST LPDIST EXTDIST ZERO ZEROUPPER	2	3	4	5	6	7	8

Table 5.2: Fence $n = 8$

Parameter	**Fence 2**
	Value
L	12
$\|V(G)\|$	36
m	72
OPT	5
LPNAT-LAYERED	2
CUTDIST	2
CUTDIST ZERO	2
CUTDIST ZERO ZEROUPPER	2
LPDIST	0.363636
LPDIST ZERO	0.363636
LPDIST ZERO ZEROUPPER	0.8
CUTDIST LPDIST	2.80919
CUTDIST LPDIST ZERO	2.92234
CUTDIST LPDIST ZERO ZEROUPPER	3.81876
CUTDIST LPDIST EXTDIST ZERO	3.47339
CUTDIST LPDIST EXTDIST ZERO ZEROUPPER	3.97654

Table 5.3: Recursive Camel Data

Parameter	**Recursive Camel**		
	1	**2**	**3**
l	2	4	6
n	3	9	25
m	3	14	45
OPT	2	3	4
LPNAT-LAYERED	2	2.5	2.5
CUTDIST	2	2.5	2.5
CUTDIST ZERO	2	2.5	2.5
CUTDIST ZERO ZEROUPPER	2	2.5	2.5
LPDIST	1	0.666667	0.6
LPDIST ZERO	1	0.666667	0.6
LPDIST ZERO ZEROUPPER	1	0.666667	0.75
CUTDIST LPDIST	2	2.5	2.75101
CUTDIST LPDIST ZERO	2	2.5	2.9
CUTDIST LPDIST ZERO ZEROUPPER	2	2.5	3.04762
CUTDIST LPDIST EXTDIST ZERO	2	2.5	2.93651
CUTDIST LPDIST EXTDIST ZERO ZEROUPPER	2	2.5	3.04762

5.4 Exhaustive Evaluation

We started the exhaustive evaluation portion of our analysis at 6 vertex graphs. It may seem odd that we ignore smaller graphs. However, our results showed that even among all 925 instances on 6 vertices, there were none for which even the natural linear program LPNAT-LAYERED gave a fractional optimum. For this reason we can say that even 6 vertex instances are too small to be interesting.

Summarized results of this evaluation is available in attachments A.1, A.2 and A.3. Each summarizes respectively the results for 6, 7 and 8 vertex graphs. Each attachment contains data where each linear program's largest integrality gap is shown and each heuristic's worst approximation ratio is shown. The results are also subdivided based on the value of L. At the end of each summary there is the total number of instances, the number for which the fractional NATLP-LAYERED program gave a non-integral optimum and the number of instances for which the CUTDIST LPDIST EXTDIST ZERO ZEROUPPER (i.e. the full alternative linear program) gave a non-integral optimum.

One result of this exhaustive evaluation is that the CUTDIST ZERO ZEROUPPER program always gave the same fractional optimum as that of the NATLP-LAYERED program. This is in line with the results we saw for the previous instances. Therefore, we are fairly certain in conjecturing that this relationship holds in general.

We also see that the fraction of sufficiently interesting graphs (i.e. ones for which at least the relaxed natural LP gives a non-integral optimum) is very small. Even for $n = 8$ this is only about 0.01% of all instances. This seems to confirm our previous belief that sampling instances randomly would not be a good approach.

Comparison of the linear program integrality gaps suggests that while the "full" linear program (CUTDIST LPDIST EXTDIST ZERO ZEROUPPER) has a smaller integrality gap than the natural linear program (LPNAT-LAYERED) it is not by much. For $n = 8$ the integrality gap of the natural linear program is $\frac{4}{3}$ in the worst case while the integrality gap of the full linear program is $\frac{6}{5}$ in the worst case. This is not a large difference. However, we expect that this is due to the fact that we are looking at very small instances here. It was our hope that we would be able to look at the data and conjecture the asymptotic behaviour of the integrality gaps. However, it seems that for instances which are small enough to be tractable for us fully the integrality gaps are too small to be useful for this purpose. For example for $n = 7$ there are still linear programs which achieve integrality gap of 1.

When it comes to heuristics we once again focus mostly on the $n = 8$ case. The best approximation ratio achieved for these instances is $\frac{4}{3}$ which is reached for the "smart" rounding approach applied to the linear program CUTDIST LPDIST EXTDIST ZERO ZEROUPPER. Interestingly we see that four linear programs share the integrality gaps here (1.2 for CUTDIST LPDIST ZERO and all other programs containing these three rules). However, their derived heuristics differ s only the full set of rules achieves the approximation ratio of $\frac{4}{3}$.

We also see that with the exception of the shortpath cut heuristic all non-LP based heuristic approaches fare poorly. This is somewhat surprising and might again just be an artifact of the small instance size as we expect the refinement of the $\frac{L}{2}$ combination heuristic to give better results on larger instances.

We can also examine the smallest graphs which are in some sense non-trivial. For example Figure 5.1 is one of the two 7 vertex graphs for which the fractional NATLP-LAYERED program gives a non-integral optimum. There is clear similarity with the camel graph instances which supports our assessment that the camel graphs are a good choice for hard instances. Similar "camel-like" structures repeat in the larger instances too.

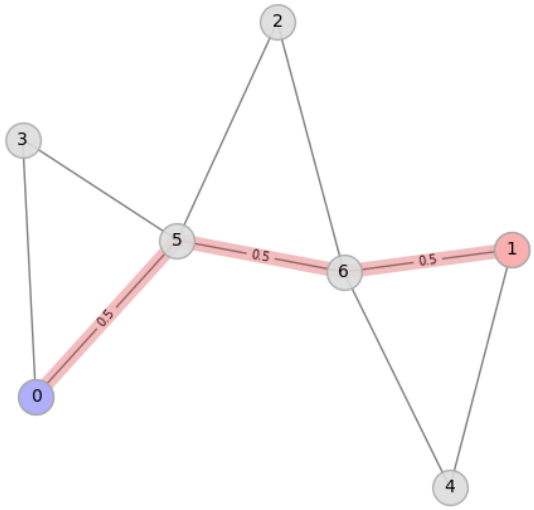

Figure 5.1: A 7 vertex graph for which the fractional NATLP-LAYERED program gives a non-integral optimum. The blue vertex is s and the red vertex is t. The edges on the direct path from s to t have x variables equal to 0.5.

On the other hand if we look at the smallest instances for which the full alternative linear program gives a non-integral optimum we see that that new patterns emerge. Figure 5.2 shows one such instance. There is some vague similarity with the camel graphs in the sense that we have shorter s-t-paths with longer paths running in parallel to them. However, out of the instances we explored this perhaps most closely resembles the recursive camel graphs, but even then the resemblance is not very strong. Likely this means that there are other hard instances which we have not managed to capture with our instance families.

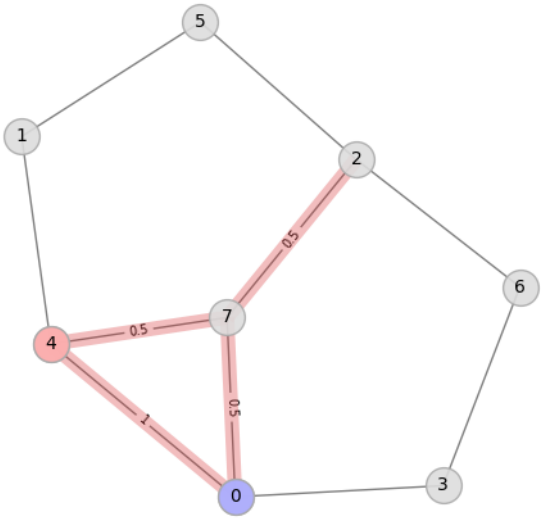

Figure 5.2: A 9 vertex graph for which the fractional CUTDIST LPDIST EXTDIST ZERO ZEROUPPER program gives a non-integral optimum. The blue vertex is s and the red vertex is t. Each edge is labelled with its y variable value.

6. Implementation Details

Let us briefly discuss some practical details, problems and their solutions that we have encountered during our implementation.

The core of our implementation is written in the programming language Python. For working with graphs we use the NetworkX library[7]. Originally we used the general purpose high-level library Pyomo [3] [8] with the LP solver Gurobi [6] version 9.1 under its academic licence. However, we have found that the overhead of Pyomo is too high for our purposes. As a general purpose LP library it provides a convenient high-level interface but, given the relatively large linear programs we worked with, this sometimes hid performance costs. Moreover, Pyomo itself is written in Python which is known not to generally be fast. After examining the performance profile of our implementation we have found that the majority of the time and memory was spent not in the LP solver but in Pyomo presolver. For this reason we opted to change our approach and bypass Pyomo by creating our own specialized Python module written in C++ which directly interfaces with Gurobi. This has resulted in a significant performance improvement.

One could also consider generating the linear program in one of the many formats that Gurobi accepts and then running Gurobi solver as a separate process. However, interfacing with Gurobi's C++ bindings directly proved to be more convenient and efficient.

As mentioned before for some instance families their vertex count grows rapidly. For that reason we resorted to substituting the camel graph with its weighted version as was discussed previously. This allowed us to continue evaluating this instance family up to $k = 7$ as opposed to $k = 3$ for the unweighted camel graph.

For all the calculations we used a university server with 32 physical cores working at frequency of 3.310 GHz and 251 GB of memory. Despite this fact we found the Pyomo running out of memory several times which was the main reason for us to switch to the custom C++ module. We did not focus on measuring the time of our implementation since it was not the main goal of this thesis. However, in general the running time of solving the largest linear programs was on the order of days.

In order to reduce the running time as much as possible we used a small camel instance and ran Gurobi's automatic parameter tuning on it. After a day's worth of iterations it produced a parameter set which we then used for all the calculations. This parameter set is included in the C++ module, and it contains the following parameters:

- `Method`: 1 = Use the dual simplex method
- `PreDual`: 1 = Force presolve to use the dual
- `PrePasses`: 1 = Perform exactly one presolve pass
- `Presolve`: 1 = Conservative presolve

These generally reduced the solving time of the linear programs by a factor of 2 to 3 compared to the default parameters.

The issues mentioned above mostly relate to the part of our analysis where we examined the behaviour of few large linear programs. However, we also did the "exhaustive evaluation" part in which the situation was different as we were running heuristics and LP solvers on comparatively small instances. A task like that is easily parallelizable, and we used the built-in Python library `multiprocessing` to run the calculations on all 64 available virtual cores. Initially we hit a memory bottleneck in some cases, but that was easily solved by having processes share a single memory mapped input file instead of each having its own copy of the input data.

In order to be able to iterate on our methods we split this part of the code into two stages. The preprocessing stage reads an input file where each line represents a connected graph in the compact graph6 format. As mentioned earlier for this input we used Brendan McKay's database of small connected unlabelled graphs [11] which alreaday ensures that we do not need to deal with multiple isomorphic graphs. The preprocessing stage then generates instances as described in the second chapter and writes their representations into a new file where each line is in the JSON format and represents a single instance. Had we made the whole file in the JSON format we would have issues when it comes to splitting the file into multiple parts for parallel processing. Notably this "JSON lines" format is then used as both the input and the output format for the actual solving stage. This allowed us to first run the solving stage with some linear programs. As we added more we could use the output of the last run as the input for the next one. This way we would not need to recompute LP or heuristic results which have already been computed.

Conclusion

We have presented a new linear programming formulation for the L-bounded cut problem. We have explored several variations of it. For the purpose of exploring its properties we discussed several instances which we expect to be sufficiently difficult to provide a good insight into the behaviour of the formulations. We have also discussed the known theoretical results and approximation algorithms for the problem. We have implemented several heuristics and approximation algorithms and compared their performance on a large set of small instances and also the difficult instances. We have also compared the performance of our formulations with the natural formulation.

We expect the alternative LP formulation to be useful for future results, but we were not able to prove any new theoretical results using it. However, we propose at least two conjectures:

Conjecture 17. *The linear program CUTDIST ZERO ZEROUPPER has the same fractional optimum as the linear program NATLP on all instances of the L-bounded cut problem.*

Conjecture 18. *On the camel graph instances the linear program CUTDIST LPDIST ZERO ZEROUPPER achieves fractional optima which turn out to be the integral optima.*

Further research could be done by trying to prove these conjectured or by exploring the behaviour of the formulations on other instance families. It would also be interesting to explore the behaviour of the formulations on larger instances than we were able to handle. One could also explore further refinement of the linear programs we proposed. The ideal goal would be eventually proving the integrality gap of the full alternative linear program and then developing an approximation algorithm with better approximation ratio than the current best known based on that result.

Bibliography

[1] J. Adámek and V. Koubek. Remarks on flows in network with short paths. *Commentationes Mathematicae Universitatis Carolinae*, 12:661–667, 1971.

[2] Georg Baier, Thomas Erlebach, Alexander Hall, Ekkehard Köhler, Heiko Schilling, and Martin Skutella. Length-bounded cuts and flows. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 679–690, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-35905-0.

[3] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021.

[4] Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. 01 2001.

[5] 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.

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

[7] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[8] William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[9] A. Itai, Y. Perl, and Y. Shiloach. The complexity of finding maximum disjoint paths with length constraints. *Networks*, 12(3):277–286, 1982. doi: https://doi.org/10.1002/net.3230120306. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230120306.

[10] Lovász László, V. Neumann-Lara, and M. Plummer. Mengerian theorems for paths of bounded length. *Period Math Hung*, 9:269–276, 12 1978. doi: 10.1007/BF02019432.

[11] Brendan McKay. Simple graphs. URL http://users.cecs.anu.edu.au/~bdm/data/graphs.html.

[12] Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927. URL http://eudml.org/doc/211191.

[13] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. doi: 10.1137/0208032. URL https://doi.org/10.1137/0208032.

[14] Vijay V. Vazirani. *Approximation Algorithms*. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-662-04565-7. URL https://doi.org/10.1007/978-3-662-04565-7.

List of Figures

List of Tables

A. Attachments

A.1 Exhaustive Evaluation Results for $n = 6$

```
Integrality gaps:
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1

Integrality gaps by L:
L=3
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=5
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=4
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
```

```
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1

Approximation ratios:
solve_del_shortest: 4
solve_naive_cut: 2
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 2.5
solve_smartround_NATLP-LAYERED: 1.5
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1.75
solve_largest_weighted_dist_lall_cutdist: 1.75
solve_roundup_lall_cutdist_zero: 2
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1.33333
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 3
solve_smartround_lall_cutdist_lpdist: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.5
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.33333
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.5
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 2
solve_greedy_walks: 4

Approximation ratios by L:
L=3
solve_del_shortest: 3
solve_naive_cut: 2
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 1
solve_NATLP-LAYERED_optimal: 1
```

```
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1.75
solve_largest_weighted_dist_lall_cutdist: 1.75
solve_roundup_lall_cutdist_zero: 2
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1.33333
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.5
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.33333
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.5
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 2
solve_greedy_walks: 4
L=5
solve_del_shortest: 4
solve_naive_cut: 1
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 2.5
solve_smartround_NATLP-LAYERED: 1.5
solve_roundup_lall_cutdist: 1
solve_smartround_lall_cutdist: 1
solve_largest_weighted_dist_lall_cutdist: 1
solve_roundup_lall_cutdist_zero: 2
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1
```

```
solve_largest_weighted_dist_lall_cutdist_lpdist: 1
solve_roundup_lall_cutdist_lpdist_zero: 1.5
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 1
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 1.5
solve_greedy_walks: 3
L=4
solve_del_shortest: 4
solve_naive_cut: 2
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1
solve_largest_weighted_dist_lall_cutdist: 1
solve_roundup_lall_cutdist_zero: 2
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 3
solve_smartround_lall_cutdist_lpdist: 1
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.5
solve_roundup_lall_cutdist_lpdist_zero: 2
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 2
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
```

```
solve_greedy: 2
solve_greedy_walks: 4

Total instances: 925
# fractional for NATLP-LAYERED: 0
# fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 0
% fractional for NATLP-LAYERED: 0.00%
% fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 0.00%
```

A.2 Exhaustive Evaluation Results for $n = 7$

```
Integrality gaps:
NATLP-LAYERED: 1.33333
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1.33333
LALL_CUTDIST_ZERO: 1.33333
LALL_CUTDIST_ZERO_ZEROUPPER: 1.33333
LALL_CUTDIST_LPDIST: 1.23077
LALL_CUTDIST_LPDIST_ZERO: 1.2
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1

Integrality gaps by L:
L=3
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=4
NATLP-LAYERED: 1.33333
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1.33333
LALL_CUTDIST_ZERO: 1.33333
LALL_CUTDIST_ZERO_ZEROUPPER: 1.33333
LALL_CUTDIST_LPDIST: 1.23077
LALL_CUTDIST_LPDIST_ZERO: 1.2
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=6
NATLP-LAYERED: 1
```

```
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=5
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1

Approximation ratios:
solve_del_shortest: 4
solve_naive_cut: 3
solve_shortpath_cut: 1.5
solve_shortestpath_cut: 2
solve_lhalf_combination: 3
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1.66667
solve_smartround_NATLP-LAYERED: 1.33333
solve_roundup_lall_cutdist: 3
solve_smartround_lall_cutdist: 1.66667
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.8
solve_largest_weighted_dist_lall_cutdist_zero: 2
solve_roundup_lall_cutdist_zero_upper: 1.5
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 3
solve_smartround_lall_cutdist_lpdist: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.8
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
```

```
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 3
solve_greedy_walks: 5

Approximation ratios by L:
L=3
solve_del_shortest: 3
solve_naive_cut: 3
solve_shortpath_cut: 1.5
solve_shortestpath_cut: 2
solve_lhalf_combination: 1.5
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 3
solve_smartround_lall_cutdist: 1.66667
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.8
solve_largest_weighted_dist_lall_cutdist_zero: 1.75
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 3
solve_smartround_lall_cutdist_lpdist: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.8
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 2
solve_greedy_walks: 5
L=4
solve_del_shortest: 4
solve_naive_cut: 2
```

```
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1.66667
solve_smartround_NATLP-LAYERED: 1.33333
solve_roundup_lall_cutdist: 2.5
solve_smartround_lall_cutdist: 1.66667
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.5
solve_largest_weighted_dist_lall_cutdist_zero: 2
solve_roundup_lall_cutdist_zero_upper: 1.5
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 3
solve_smartround_lall_cutdist_lpdist: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.66667
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 2
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.33333
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.5
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 2
solve_greedy_walks: 2
L=6
solve_del_shortest: 4
solve_naive_cut: 1
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 3
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1.66667
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 1
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
```

```
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.66667
solve_roundup_lall_cutdist_lpdist_zero: 1
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 1
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 3
solve_greedy_walks: 3.5
L=5
solve_del_shortest: 3.5
solve_naive_cut: 2
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1.5
solve_largest_weighted_dist_lall_cutdist: 1.5
solve_roundup_lall_cutdist_zero: 1
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.5
solve_roundup_lall_cutdist_lpdist_zero: 1
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 2
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
```

```
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 3
solve_greedy_walks: 2

Total instances: 10881
# fractional for NATLP-LAYERED: 2
# fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 0
% fractional for NATLP-LAYERED: 0.02%
% fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 0.00%
```

A.3 Exhaustive Evaluation Results for $n = 8$

```
Integrality gaps:
NATLP-LAYERED: 1.33333
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1.33333
LALL_CUTDIST_ZERO: 1.33333
LALL_CUTDIST_ZERO_ZEROUPPER: 1.33333
LALL_CUTDIST_LPDIST: 1.23077
LALL_CUTDIST_LPDIST_ZERO: 1.2
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1.2

Integrality gaps by L:
L=3
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=4
NATLP-LAYERED: 1.33333
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1.33333
LALL_CUTDIST_ZERO: 1.33333
LALL_CUTDIST_ZERO_ZEROUPPER: 1.33333
LALL_CUTDIST_LPDIST: 1.23077
LALL_CUTDIST_LPDIST_ZERO: 1.2
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1.09091
```

```
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1.09091
L=5
NATLP-LAYERED: 1.2
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1.2
LALL_CUTDIST_ZERO: 1.2
LALL_CUTDIST_ZERO_ZEROUPPER: 1.2
LALL_CUTDIST_LPDIST: 1.2
LALL_CUTDIST_LPDIST_ZERO: 1.2
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1.2
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1.2
L=7
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1
L=6
NATLP-LAYERED: 1
NATLP-LAYERED_BOOL: 1
LALL_CUTDIST: 1
LALL_CUTDIST_ZERO: 1
LALL_CUTDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST: 1
LALL_CUTDIST_LPDIST_ZERO: 1
LALL_CUTDIST_LPDIST_ZERO_ZEROUPPER: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO: 1
LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 1

Approximation ratios:
solve_del_shortest: 4.5
solve_naive_cut: 3
solve_shortpath_cut: 1.5
solve_shortestpath_cut: 3
solve_lhalf_combination: 3
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 2.33333
solve_smartround_NATLP-LAYERED: 1.66667
solve_roundup_lall_cutdist: 4
solve_smartround_lall_cutdist: 2.5
solve_largest_weighted_dist_lall_cutdist: 1.8
```

```
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.83333
solve_largest_weighted_dist_lall_cutdist_zero: 2
solve_roundup_lall_cutdist_zero_upper: 2.33333
solve_smartround_lall_cutdist_zero_upper: 1.66667
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1.66667
solve_roundup_lall_cutdist_lpdist: 4
solve_smartround_lall_cutdist_lpdist: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.8
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 2
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 2
solve_roundup_lall_cutdist_lpdist_zero_upper: 2.33333
solve_smartround_lall_cutdist_lpdist_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1.33333
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.8
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 2
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1.66667
solve_greedy: 3
solve_greedy_walks: 6

Approximation ratios by L:
L=3
solve_del_shortest: 3
solve_naive_cut: 3
solve_shortpath_cut: 1.5
solve_shortestpath_cut: 2
solve_lhalf_combination: 1.5
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 3
solve_smartround_lall_cutdist: 1.8
solve_largest_weighted_dist_lall_cutdist: 1.8
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.83333
solve_largest_weighted_dist_lall_cutdist_zero: 2
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2.5
solve_smartround_lall_cutdist_lpdist: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.8
solve_roundup_lall_cutdist_lpdist_zero: 3
solve_smartround_lall_cutdist_lpdist_zero: 2
```

```
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 2
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 3
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.8
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.8
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 2
solve_greedy_walks: 6
L=4
solve_del_shortest: 4
solve_naive_cut: 2
solve_shortpath_cut: 1.5
solve_shortestpath_cut: 3
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 2.33333
solve_smartround_NATLP-LAYERED: 1.66667
solve_roundup_lall_cutdist: 4
solve_smartround_lall_cutdist: 2.5
solve_largest_weighted_dist_lall_cutdist: 1.75
solve_roundup_lall_cutdist_zero: 3
solve_smartround_lall_cutdist_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_zero: 1.75
solve_roundup_lall_cutdist_zero_upper: 2
solve_smartround_lall_cutdist_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1.66667
solve_roundup_lall_cutdist_lpdist: 4
solve_smartround_lall_cutdist_lpdist: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.75
solve_roundup_lall_cutdist_lpdist_zero: 2.33333
solve_smartround_lall_cutdist_lpdist_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.75
solve_roundup_lall_cutdist_lpdist_zero_upper: 2.33333
solve_smartround_lall_cutdist_lpdist_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1.33333
solve_roundup_lall_cutdist_lpdist_extended_zero: 2.33333
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.75
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.75
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 2
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1.25
solve_greedy: 3
solve_greedy_walks: 3
L=5
```

```
solve_del_shortest: 4.5
solve_naive_cut: 2
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1.66667
solve_smartround_NATLP-LAYERED: 1.33333
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1.33333
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 1.33333
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 2.33333
solve_smartround_lall_cutdist_zero_upper: 1.66667
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1.66667
solve_roundup_lall_cutdist_lpdist: 2.66667
solve_smartround_lall_cutdist_lpdist: 1.66667
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.66667
solve_roundup_lall_cutdist_lpdist_zero: 2
solve_smartround_lall_cutdist_lpdist_zero: 1.33333
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_zero_upper: 1.33333
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 2.5
solve_smartround_lall_cutdist_lpdist_extended_zero: 1.33333
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.66667
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 2
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1.5
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1.66667
solve_greedy: 3
solve_greedy_walks: 2
L=7
solve_del_shortest: 3
solve_naive_cut: 1
solve_shortpath_cut: 1
solve_shortestpath_cut: 1.75
solve_lhalf_combination: 2
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 2
solve_smartround_lall_cutdist: 1
solve_largest_weighted_dist_lall_cutdist: 1.66667
solve_roundup_lall_cutdist_zero: 1
solve_smartround_lall_cutdist_zero: 1
```

```
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.66667
solve_roundup_lall_cutdist_lpdist_zero: 1
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist_extended_zero: 2
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1
solve_greedy: 1
solve_greedy_walks: 1.75
L=6
solve_del_shortest: 3.5
solve_naive_cut: 1
solve_shortpath_cut: 1
solve_shortestpath_cut: 2
solve_lhalf_combination: 3
solve_NATLP-LAYERED_optimal: 1
solve_roundup_NATLP-LAYERED: 1
solve_smartround_NATLP-LAYERED: 1
solve_roundup_lall_cutdist: 1.5
solve_smartround_lall_cutdist: 1
solve_largest_weighted_dist_lall_cutdist: 1
solve_roundup_lall_cutdist_zero: 1
solve_smartround_lall_cutdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_zero: 1
solve_roundup_lall_cutdist_zero_upper: 1
solve_smartround_lall_cutdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_zero_upper: 1
solve_roundup_lall_cutdist_lpdist: 2
solve_smartround_lall_cutdist_lpdist: 1
solve_largest_weighted_dist_lall_cutdist_lpdist: 1.33333
solve_roundup_lall_cutdist_lpdist_zero: 1
solve_smartround_lall_cutdist_lpdist_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero: 1
solve_roundup_lall_cutdist_lpdist_zero_upper: 1
solve_smartround_lall_cutdist_lpdist_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_zero_upper: 1
```

```
solve_roundup_lall_cutdist_lpdist_extended_zero: 1.66667
solve_smartround_lall_cutdist_lpdist_extended_zero: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero: 1.33333
solve_roundup_lall_cutdist_lpdist_extended_zero_upper: 1.66667
solve_smartround_lall_cutdist_lpdist_extended_zero_upper: 1
solve_largest_weighted_dist_lall_cutdist_lpdist_extended_zero_upper: 1.33333
solve_greedy: 2
solve_greedy_walks: 2

Total instances: 191930
# fractional for NATLP-LAYERED: 28
# fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 5
% fractional for NATLP-LAYERED: 0.01%
% fractional for LALL_CUTDIST_LPDIST_EXTENDED_ZERO_ZEROUPPER: 0.00%
```