Universidade Estadual de Campinas
Instituto de Computação

Ulysses Alessandro Couto Rocha

Heuristic Techniques for Large-Scale Instances of the Cable-Trench Problem

Técnicas Heurísticas para Instâncias de Grande Porte do Problema Cabo-Trincheira

CAMPINAS
2018

Ulysses Alessandro Couto Rocha

Heuristic Techniques for Large-Scale Instances of the Cable-Trench Problem

Técnicas Heurísticas para Instâncias de Grande Porte do Problema Cabo-Trincheira

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Flávio Keidi Miyazawa
Co-supervisor/Coorientador: Prof. Dr. Eduardo Candido Xavier

Este exemplar corresponde à versão final da Dissertação defendida por Ulysses Alessandro Couto Rocha e orientada pelo Prof. Dr. Flávio Keidi Miyazawa.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): CNPq, 133323/2018-8; CNPq, 131175/2017-3

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R582h

Rocha, Ulysses Alessandro Couto, 1992-
Heuristic techniques for large-scale instances of the cable-trench problem / Ulysses Alessandro Couto Rocha. – Campinas, SP : [s.n.], 2018.

Orientador: Flávio Keidi Miyazawa.
Coorientador: Eduardo Candido Xavier.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Programação heurística. 2. Otimização combinatória. 3. GRASP (Meta-heurística). I. Miyazawa, Flávio Keidi, 1970-. II. Xavier, Eduardo Candido, 1979-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Técnicas heurísticas para instâncias de grande porte do problema cabo-trincheira
Palavras-chave em inglês:
Heuristic programming
Combinatorial optimization
GRASP (Metaheuristic)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Flávio Keidi Miyazawa [Orientador]
Pedro Augusto Munari Junior
Fábio Luiz Usberti
Data de defesa: 10-12-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Ulysses Alessandro Couto Rocha

Heuristic Techniques for Large-Scale Instances of the Cable-Trench Problem

Técnicas Heurísticas para Instâncias de Grande Porte do Problema Cabo-Trincheira

Banca Examinadora:

- Prof. Dr. Flávio Keidi Miyazawa
  IC/UNICAMP
- Prof. Dra. Pedro Augusto Munari Junior
  UFSCar
- Prof Dr. Fábio Luiz Usberti
  IC/UNICAMP

Ata da defesa com as respectivas assinaturas dos membros encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade

Campinas, 10 de dezembro de 2018

Agradecimentos

Agradeço aos meus pais e à minha irmã, que foram sempre meus grandes apoiadores e incentivadores durante toda minha vida.

Aos contribuintes brasileiros, que através do CNPq, CAPES, Instituto Federal do Ceará, UNICAMP e outros órgãos de fomento e instituições de ensino, financiaram direta e indiretamente meus estudos.

Aos professores e demais funcionários do Instituto de Computação da UNICAMP, em especial aos meus orientadores Flávio Miyazawa e Eduardo Xavier, pela paciência, apoio e orientação durante o mestrado.

Aos colegas do Laboratório de Otimização e Combinatória, pelo convívio, ideias e excelentes conversas durante todos esses anos de curso.

À minha esposa, Raphaela, pela incondicional paciência, apoio e revisões desse e de diversos outros trabalhos.

Aos amigos, que sempre apoiaram e incentivaram meus estudos.

Aos demais professores que tive ao longo da vida e a todos que direta ou indiretamente fizeram parte da minha formação, meu muito obrigado!

Resumo

O problema cabo trincheira foi apresentado em 2002 para modelar redes cabeadas. Esse problema pode ser visto como a união do problema de caminhos mínimos com o problema da árvore geradora mínima. Como entrada do problema temos um grafo $G = (V, E)$ com pesos nas arestas que indicam a distância entre os vértices incidentes na mesma. Há um vértice especial que representa uma instalação e demais vértices representam clientes. Uma solução para o problema é uma árvore geradora enraizada na instalação. O custo da solução é o custo da árvore geradora multiplicado por um fator de custo de trincheira mais os custos de cabos. Para cada cliente, o seu custo de cabo é dado pelo custo do caminho do cliente até a instalação multiplicado por um fator de custo de cabo. Esse problema modela cenários onde cada cliente deve ser conectado a uma instalação central através de um cabo dedicado. Cada cabo deve estar acomodado em uma trincheira e cada trincheira pode conter um número ilimitado de cabos. Sabendo que o custo dos cabos e trincheiras é proporcional a seu comprimento multiplicado por um fator de custo, o problema é encontrar uma rede com custo mínimo. Trabalhos anteriores utilizaram o problema cabo trincheira para modelar problemas em telecomunicações, distribuição de energia, redes ferroviárias e até para reconstrução de vasos sanguíneos em exames de tomografia computadorizada.

O trabalho foca na resolução do problema em instâncias de grande porte (superiores a 10 mil vértices). Foram desenvolvidas várias heurísticas para o problema. Na busca por simplificações de instâncias, foram demonstradas regras seguras, ou seja, que não comprometem nenhuma solução ótima, e heurísticas para a remoção de arestas eliminando aquelas que dificilmente estariam em "boas soluções" de uma instância. Foi apresentado um algoritmo rápido para busca local capaz de ser executado mesmo em instâncias de grande porte. Foram desenvolvidos também algoritmos baseados em Greedy Randomized Adaptive Search Procedure (GRASP) e formulada uma heurística que contrai vértices. Com a contração de vértices, foram criadas instâncias do problema Cabo Trincheira com Demandas nos Vértices (CTDV). Essa versão com demandas tem um número menor de vértices que o problema original, o que viabiliza o uso de algoritmos baseados em programação linear para resolvê-lo. Foi demonstrado como é possível, ao resolver essa versão reduzida com demandas, remontar uma solução viável para o problema cabo trincheira original.

Foram obtidos, com essas heurísticas, resultados melhores do que trabalhos anteriores encontrados na literatura do problema. Para além disso, foi demonstrado como essa técnica de contração de vértices tem o potencial para resolver instâncias de tamanhos ainda maior para o problema cabo trincheira.

Abstract

The Cable Trench Problem (CTP) was presented in 2002 to model wired networks. This problem can be seen as the combination of the shortest path problem with the minimum spanning tree problem. An instance of the problem is composed by a graph $G = (V, E)$ with weigths, representing the distance between a pair of vertices. A special vertex represents a facility, and all others are clients. A solution to the problem is a spanning tree rooted in the facility.

The solution's cost is given by the spanning tree cost multiplied by a trench cost factor, added by the cables cost reaching the root from each vertex in the graph. For each client, its cable cost is given by the path in the spanning tree, from the client to the root, multiplied by a cable cost factor. The CTP models a scenario where each client must be connected through a dedicated cable to a central facility. Each cable must be laying on a trench and a trench may hold an unlimited number of cables. Knowing that the cost of cables and trenches are proportional to its lengths multiplied by a cost factor, the problem is to find a network of minimum cost. Previous works in the literature used the CTP to model telecommunication problems, power distribution, rail networks, and even a blood vessel networks for computed tomography exams.

In this research, we focused on large-scale instances of the problem (above 10 thousand vertices), achieving better results than previous works found in the literature. We developed a series of heuristics for the problem. Searching for a simplification for those instances, we present safe reductions, that do not affect any optimal solution, and heuristic reduction rules that are capable of removing edges unlikely to be part of "good" solutions in an instance. We present a fast local search algorithm, capable of improving even solutions for large-scale instances. We developed an algorithm based on a Greedy Randomized Adaptive Search Procedure (GRASP) and formulated a heuristic to cluster vertices. By clustering vertices, we represent a CTP instance as an instance of the Cable Trench Problem with Demands (CTPD). We represent the large-scale CTP instance into a vertex-wise smaller one adding demands to its vertices. Dealing with smaller instances, we enable a new range of techniques such as linear programming based algorithms to solve it. We demonstrate how this instances with demands can be used to build a feasible solution for the original CTP instance. We also demonstrate how this vertex clustering technique has the potential to solve even larger scale instances for the CTP.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

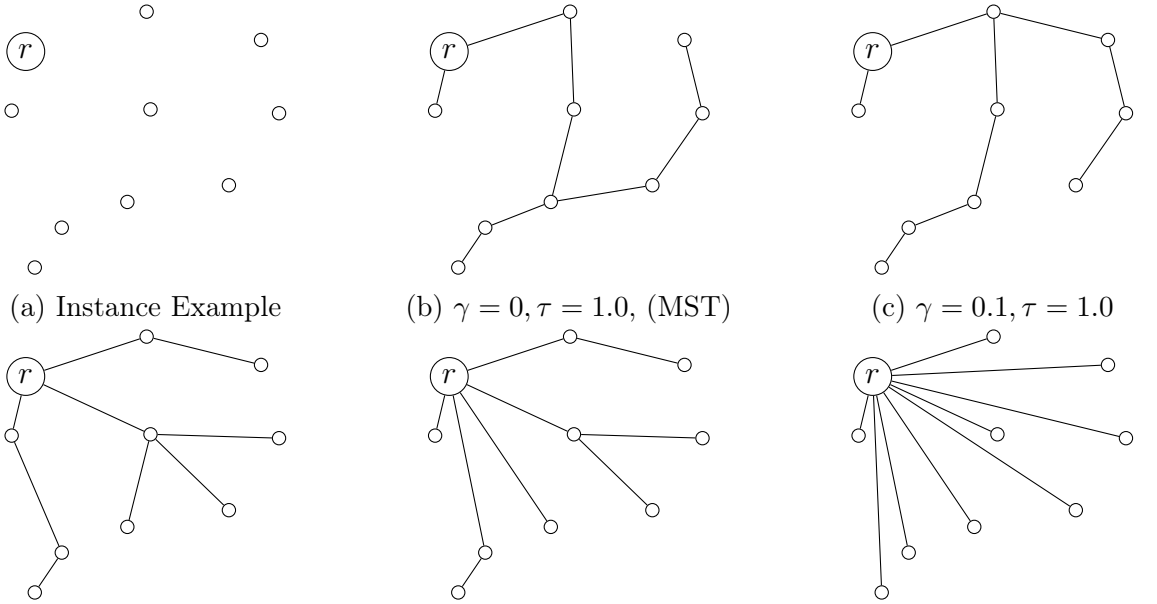
Introduction

In large wired-network systems, such as telecommunications and power distribution, building or upgrading the infrastructure demands high investment and a huge load of work considering several different aspects. Planning the network and building the infrastructure is a major cost component of those systems. Given the project requirements, it is desirable to find the best network arrangement to minimize its cost. To achieve this goal, computational models can be used to assist in the planning process and find the best possible configuration, reducing the final price of the infrastructure and the time required to build the system.

The Cable-Trench Problem (CTP) can model and be used to solve different optimization scenarios arising in the design of wired networks. It is an NP-Hard problem, first presented by Vasko et al. [2002], combining the Minimum Spanning Tree (MST) problem and the Single Source Shortest Path (SSSP) problem. The CTP can be described as a problem that aims at minimizing the costs of digging trenches and laying cables to connect nodes to a central hub through a network.

The following scenario can be used to illustrate its application. Consider that in a hypothetical university, with many buildings, each building must be connected to a central computer through a dedicated cable, where each cable must be laying on a trench, and each trench can carry an unlimited number of cables at once. Cables and trenches have different costs: a cost factor of γ for the cable, and τ for the trench, proportional to their length. The problem is to find a network connecting every building to the server with minimum cost.

In Figure 1.1, considering the previous example scenario, the vertex r would represent the central computer, and all the other vertices would represent the buildings from the university. Note that different pairs of values of τ and γ may influence the optimal solution layout obtained. Observing the special cases, if $\gamma = 0$ and $\tau > 0$, as presented in Figure 1.1b, to minimize the network cost, as the cables have no cost, the goal becomes to "dig" as little trenches as possible, this special case reduces the CTP to a Minimum Spanning Tree (MST) problem. In the opposite case, if $\gamma > 0, \tau = 0$, as each vertex need to get a cable from the root, the problem turns into minimizing the value spend in cables, as shown in Figure 1.1f, we reduce to a Single Source Shortest Path (SSSP) problem.



(a) Instance Example (b) $\gamma = 0, \tau = 1.0$, (MST) (c) $\gamma = 0.1, \tau = 1.0$

(d) $\gamma = 1.0, \tau = 1.0$ (e) $\gamma = 1.0, \tau = 0.1$ (f) $\gamma = 1.0, \tau = 0$, (SSSP)

Figure 1.1: Instances of CTP.

In previous works found in the literature, Nielsen et al. [2008] showed the practical relevance of CTP for telecommunication problems, analyzing the winnings of solving the CTP for a scenario of communications access networks. They presented, for a large-scale data access network, an analysis of cost reduction in the order of 8% comparing to the most efficient previous approach of network design.

Jamili and Ramezankhani [2015] presented a Mixed Integer Linear Programming model (MILP) to find the best routes between power substations and buildings, applied to model power transmission on a metro depot scenario. They also used the proposed solution to a real case study at the metro depot in Iran and analyzed the results comparing to previous formulations.

Girard et al. [2011] used the CTP to find the optimal layout for cables and trenches connecting the 96 low-frequency radio astronomy antennas at the LOFAR Super Station in Nançay, France.

For medical applications of blood vessel networks, Vasko et al. [2016] presented a model to use in image correction scenarios, where trenches represent blood vessels, cables represent blood volume and cost coefficients come from physical properties. They presented a variant to the CTP called, the Generalized Cable-Trench Problem, giving a MILP formulation and very fast heuristics for large-scale instances, been tested for graphs with up to 25.001 vertices.

Other variants of the CTP were also presented. One of them is the p-Cable Trench Problem, introduced by Marianov et al. [2012]. They proposed scenarios to optimize the construction of logging roads and sawmills and for building canals and wells for irrigation. An Integer Linear Programming formulation was provided and a heuristic algorithm based on Lagrangian relaxation was used for instances up to 300 nodes.

Schwarze [2015] introduced the Multi-Commodity Cable-Trench Problem (MC-CTP) with the possibility of modeling scenarios where different network operators can share

the infrastructural costs of the system. This work highlights applications not limited to telecommunications or power distribution networks. They described how one can use the MC-CTP in transportation scenarios, like in the railroad design, modeling rail tracks as trenches and considering the cables commodity as train lines, looking for minimizing the time of the travels and the cost of the infrastructure.

Calik et al. [2017] presented a variant called the Capacitated p-Cable Trench Problem with Covering, which can be used to model problems in wireless networks, and also can be applied in most previous variants of the CTP. They proposed an algorithmic framework based on Benders decomposition, solving instances up to 900 vertices.

Another variant, the p-Cable-Trench Problem with Facility Location, presented by Rocha et al. [2017] model scenarios where one has to decide the placement of cable-sources to serve clients in a cable-trench network by paying an opening cost. They used two heuristics, one based on *Relax-and-Fix* (Wolsey [1998]) and another based on *BRKGA* (Gonçalves and Resende [2011]), with numerical experiments on instances up to 614 vertices.

In this work, we developed a range of techniques to solve large-scale instances of the Cable Trench Problem. As results, we defined a set of edge reduction rules capable of removing a significant number of edges on large-scale instances. In our main instance set, with 310 instances, containing graphs with up to 33.708 vertices, we achieved an average of 98.65% of the edges being removed. Two of this reduction rules are safe, and therefore can also be used in exact algorithms. We also presented a series of heuristics and a local search algorithm that achieved an average gap to the lower bound of 6.149%. The largest gap found for our best algorithm was of 37.898%. All algorithms were executed in our experiments with a time limit of 10 minutes.

This research is organized as follows, Section 2 presents the CTP. Section 2.2 present lower bounds for the problem. Section 2.3 shows how reductions can be applied to remove safely and heuristically edges of an instance. In Section 2.4, we present a fast Local Search algorithm that can significantly improve the quality of heuristic solutions. Heuristics are presented in Section 2.5. In Section 3 we present a new variant of the CTP adding demands on the vertices. We use this new variant in Section 3.3 to introduce a new heuristic framework to solve large-scale instances of the CTP. Computational experiments are presented in Section 4.2, and in Section 5 we present our conclusions and summarise future work.

Chapter 2

The Cable Trench Problem

The CTP was introduced by Vasko et al. [2002], with the potential of finding the cost-optimal layout of telecommunication networks. Proven to be NP-Hard in general, but polynomially solvable in special cases, CTP is a combination of the Minimum Spanning Tree Problem and the Single Source Shortest Path Problem.

Given a set of instances $\mathcal{I}_{CTP}$, a function $\ell_T(r, v)$ that gives the distance from a vertex r to a vertex v in a tree T, and a function $l : E \to \mathbb{R}^+$ which gives the length of each edge on a given graph $G = (V, E)$, the CTP can be formally described as:

> **Cable-Trench Problem:** An instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$ consists of a graph $G = (V, E)$, a root vertex $r \in V$, a function $l : E \to \mathbb{R}^+$, and cost factors τ and γ for trench and cables. The problem is to find a spanning tree $T = (V, E_T)$ of G in which $\tau \sum_{e \in E_T} l_e + \gamma \sum_{v \in V} \ell_T(r, v)$ is minimum.

Note that the cost function can be splitted in a trench cost function and a cable cost function. For the trenches, the trench cost factor τ is multiplied by the sum of the length of the trenches in the solution, $\sum_{e \in E_T} l_e$. For the cables, the cable cost factor γ is multiplied by the sum of the distance of every vertex to the root in T, $\sum_{v \in V} \ell_T(r, v)$.

2.1 CTP Formulations

To formulate the CTP in mathematical terms, we present Mixed Integer Linear Programming models (MILP). The following models consider graphs with directed arcs, where we denote the pair (i, j) or ij as the arc leaving from vertex i and reaching vertex j.

We denote by $A(E)$ the set of arcs obtained from the edges in E, i.e for each edge $\{i, j\} \in E$ we include both arcs (i, j) and (j, i) in $A(E)$. Let us also denote $\delta(i) = \{e \in E : e \text{ is incident to } i\ \}$, $\delta^+(i) = \{ij \in A(E)\}$ and $\delta^-(i) = \{ji \in A(E)\}$.

2.1.1 Vasko Formulation

This model was firstly defined in Vasko et al. [2002]. Let $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$ be an instance of the problem, where l_{ij} gives the length of arc ij. We use integer variables x_{ij} for the number of cables in the arc $ij \in A(E)$, and binary variable y_{ij} to indicates the existence of a trench in arc $ij \in A(E)$.

The formulation is presented below.

$$\text{Minimize} \quad \tau[\sum_{a \in A(E)} l_a y_a] + \gamma[\sum_{a \in A(E)} l_a x_a], \tag{2.1}$$

subject to:

$$\sum_{a \in \delta^+(r)} x_a = n - 1, \tag{2.2}$$

$$\sum_{a \in \delta^+(i)} x_a - \sum_{a \in \delta^-(i)} x_a = -1 \qquad \forall i \in (V \setminus r), \tag{2.3}$$

$$\sum_{a \in A(E)} y_a = n - 1, \tag{2.4}$$

$$x_a \leq (n-1) y_a \qquad \forall a \in A(E), \tag{2.5}$$

$$x_a \geq 0 \qquad \forall a \in A(E), \tag{2.6}$$

$$y_a \in \{0, 1\} \qquad \forall a \in A(E). \tag{2.7}$$

The objective function in (2.1) minimizes the overall cost of the spanning tree. Constraint (2.2) ensures that $n-1$ cables leaves the root. Constraint (2.3) ensures that each one of the nodes is connected by one cable. Note that the solution is a spanning tree, so constraint (2.4) ensures that $n-1$ trenches are in the solution. This constraint is not required, but it strengthens the formulation by reducing the solution space to be searched. Constraint (2.5) ensures a cable is not laid in an arc unless a trench is dug on it.

2.1.2 Multi-Commodity Formulation

Marianov et al. [2012] presented a multi-commodity flow formulation for a variant of the CTP, called the p-Cable-Trench problem. In this variant, an instance is similar to the CTP, a tuple $(G, r, l, p, \tau, \gamma)$, where p is an integer that defines the number of roots in the solution network. The p-Cable-Trench problem aims to find a minimum cost network with at most p cable sources. Each cable source will be a root in the spanning-forest that connects every node in the Graph, with the same rules of the CTP.

We adapted this model for the simple version of CTP. In the following formulation, variable f_{ij}^k indicates if the cable that reach vertex k passes in arc $ij \in A(E)$. The binary variable y_{ij} indicates the existence of a trench on arc $ij \in A(E)$.

$$\text{Minimize} \quad \gamma[\sum_{k \in V} \sum_{a \in A(E)} l_a f_a^k] + \tau[\sum_{a \in A(E)} l_a y_a], \tag{2.8}$$

subject to:

$$\sum_{a \in \delta^+(r)} f_a^k = 1 \qquad \forall k \in V, \tag{2.9}$$

$$\sum_{a \in \delta^-(k)} f_a^k = 1 \qquad \forall k \in V, \tag{2.10}$$

$$\sum_{a \in \delta^+(i)} f_a^k - \sum_{a \in \delta^-(i)} f_a^k = 0 \qquad \forall k \in V, i \in V \setminus k, \tag{2.11}$$

$$\sum_{a \in \delta^-(i)} y_a = 1 \qquad \forall i \in V, \tag{2.12}$$

$$f_a^k \leq y_a \qquad \forall a \in A(E) \setminus \delta^-(k), \tag{2.13}$$

$$f_a^k \geq 0 \qquad \forall a \in A(E), k \in V, \tag{2.14}$$

$$y_a \in \{0, 1\} \qquad \forall a \in A(E). \tag{2.15}$$

The objective function (2.8) aims to minimize the overall cost of the spanning tree. Constraints (2.9), (2.10) and (2.11) are related to flow conservation, stating that for each vertex, a unit of flow should origin from the root, that each node should receive its respective unit of flow, and that every unit of flow received that is destined to others vertices should exit it. Constraint (2.12) states that every vertex should be reached by one trench. Constraint (2.13) sets that if a cable uses some arc, then a trench must exist in that arc.

2.2 Lower Bound

For a minimization problem such as the CTP, lower bounds can be used to estimate, how far from an optimal network each heuristic solution is. Especially for instances where it is intractable to find an optimal solution, the use of lower bounds makes it possible to evaluate the quality of our heuristics.

We present a simple lower bound method for the problem using solutions for the Single Source Shortest-Path (SSSP) and the Minimum Spanning Tree (MST) problems. We also briefly introduce other lower bound methods found in previous works on the literature, but not suitable for our scale requirements.

2.2.1 SSSP + MST Lower Bound

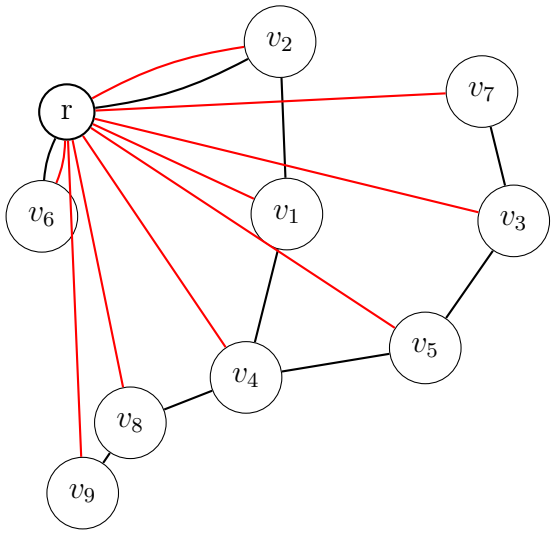


Figure 2.1: SSSP (red) + MST (black) lower bound on instance

This basic lower bound was introduced by Vasko et al. [2002] when presenting the problem. In the CTP we have two cost factors: cables and trenches. Individually calculating the optimal cost for each of those cost factors result in a valid lower bound for an instance.

As illustrated by Figure 2.1, for the cables cost, as our goal is to connect each vertex to the root. The cheapest connection is given by the shortest path from the source to each vertex in the graph. That can be calculated using the Dijkstra's Algorithm for the Single Source Shortest-Path (SSSP) problem (Dijkstra [1959]).

For the trench costs, the Minimum Spanning Tree (MST) will result in the lowest network cost design that connects every vertex in the graph, resulting in the lowest possible trench cost design. To calculate the MST, we can use Prim's Algorithm (Prim [1957]).

Therefore, we have by the SSSP a cable cost lower bound, and by the MST the trench cost lower bound. Adding those factors results in a lower bound to the CTP problem. Both algorithms are polynomial and suitable to be executed in large-scale instances, as demonstrated in our experiments.

2.2.2 Alternative Lower Bounds

Alternative lower bounds were present in previous works about the CTP. We will not dive in depth about them, mostly because they are not suitable for large-scale instances due to memory and computational power constraints.

Relaxed Linear Programming Solving an Integer Linear Programming model, like the ones in Section 2.1 provide an optimal solution. But when their variables have the integrality constraint relaxed, we have a new, and an easier problem that can be solved in polynomial time and that provides a valid lower bound for the problem.

Lagrange Relaxation Marianov et al. [2012] presented a Lagrangian relaxation algorithm derived from the multi-commodity flow formulation for the CTP. Following the idea of relaxing constraints, the Lagrangian relaxation takes a subset of constraints and moves them to the objective function, what makes to problem easier to solve and also provides a valid lower bound for an instance of the problem. More information about the Lagrangian relaxation can be found on (Wolsey [1998]) and on the paper of Marianov et al. [2012].

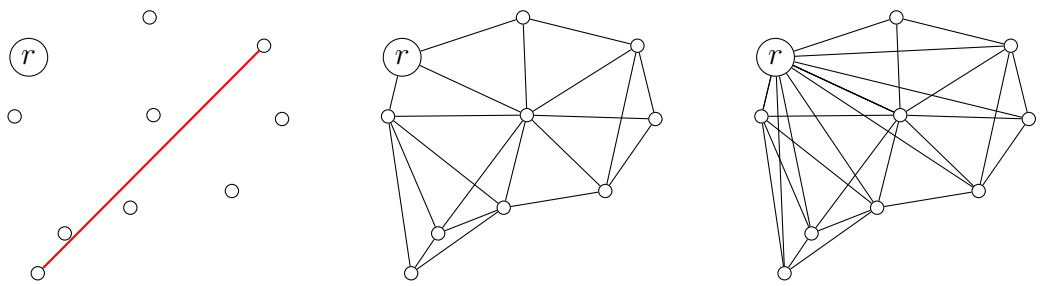
2.3 Edge Reduction Rules

We propose safe and heuristic edge reduction rules for metric instances, i.e instances where the distance function l satisfies the triangle inequality. By safe reduction rules we mean reduction that do not remove edges used in any optimal solution, and by heuristic reduction rules, one that are capable of removing edges that will unlikely be a part of "good" solutions, but may eventually remove edges used in an optimal solution.

As a result, we can reduce an instance size, substantially simplifying it. Therefore, these reductions enable heuristics, local search procedures and algorithmic techniques otherwise infeasible due to memory and computational-power limitations.

Due to some of our heuristic's requirements, we state that for every reduction below, we must keep the arc between the root and each vertex in the instance. More information about how these rules are used in our experiments, and the results achieved can be found in Section 4.2.

2.3.1 Reduction rule 1



(a) Longest Edge (LE) (b) Edges $\leq 50\%$ of LE (c) Reduction 1, $\alpha = 0.5$

Figure 2.2: Reduction Rule 1

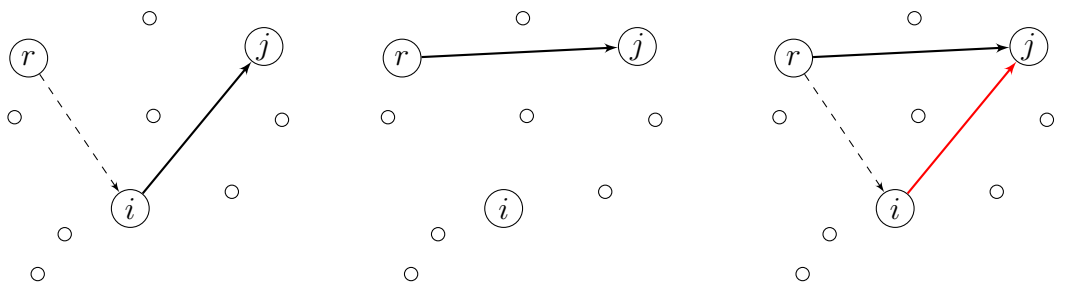
The first reduction ever proposed for the CTP, to the best of our knowledge, was presented by Vasko et al. [2016] in their computational experiments using complete graphs of size up to 25.001 vertices. They only consider edges with a length smaller than 10% of the diameter of the input graph. They argue that, in practice, this reduction was consistent with their vascular dataset, reducing their complete graphs into a more memory-wise manageable instance. Notice that this reduction rule is not safe, that is, it may remove edges that would be used in an optimal solution. We can apply this rule for any factor of the instance diameter, removing edges that are greater than a factor λ value from the

longest edge length. We slightly modify this reduction rule for our experiments, keeping the root directly connected to every node in the graph.

We can formalize it as:

> **Reduction Rule 1:** Let an instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$, where $G = (V, A)$, and reduction parameter $\lambda \in [0, 1]$. The reduced graph is $G^* = (V, A^*)$, where $A^* = \{a \in A : l_a \leq \lambda \cdot l_{max}\} \cup \{(r, i) : i \in V\}$, where $l_{max} = max\{l_a : a \in A\}$

2.3.2 Reduction rule 2



(a) $cost_{min}(i, j)$ (b) $cost(r, j)$ (c) $cost(r, j) \leq cost_{min}(i, j)$

Figure 2.3: Reduction Rule 2 decision step

We present a safe reduction rule that can remove edges without compromising any optimal solution. Let us first calculate the lowest possible cost that a vertex i can reach a vertex j by using edge (i, j). As illustrated in Figure 2.3a, the lowest possible cost that i can reach j by (i, j) is by paying the cable and trench cost between i and j, plus the cable of the shortest path from r to i,

$$cost_{min}(i, j) = \gamma(sp(r, i) + l_{ij}) + \tau(l_{ij}) \tag{2.16}$$

Let us now consider that we connect the root vertex r to a vertex j by their shortest path on the graph, paying the cable and trench prices for the whole length of this connection. The shortest path of r to j gives an upper bound to the connection cost of j in the solution. Figure 2.3b show an example of the shortest-path-connection of the root r to a vertex j, $sp(r, j)$, where $cost(r, j) = sp(r, j)(\tau + \gamma)$.

By knowing an upper bound for connecting j, for any $i \in \delta^-(j)$, we can remove all edges (i, j) that cannot lead to a lower connection cost.

Note that by connecting r to j by their shortest path, we have a upper bound for the cost of reaching j of minimal cable distance, that is, we have a feasible path from r to j that cannot reach the root by a lower distance. It implies that in a solution, replacing (i, j) by $sp(r, j)$ would not increase the cable distance of any vertex in the sub-tree rooted by j, and therefore, would not increase the cost for any other vertex.

Let us formally define the reduction rule 2 as:

> **Reduction Rule 2:** Let an instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$, where $G = (V, A)$. The reduced graph is $G^* = (V, A^*)$, where $A^* = \{(i, j) \in A \ : cost(r, j) \geq cost_{min}(i, j)\} \cup \{(r, i) : i \in V\}$.

Lemma 2.3.1 *Reduction Rule 2 is safe.*

Proof. To prove the Reduction Rule 2, let us assume, with the aim of obtaining a contradiction, that there exists an optimal solution tree T^* where $(i, j) \in T^*$, and $cost(r, j) < cost_{min}(i, j)$. By replacing (i, j) by $sp(r, j)$ we would obtain a new solution with lower cost, since we would not increase the cost of connecting any vertex in the sub-tree rooted by j, as the cable distance to every node on its sub-tree would be at most the same as in the current solution, given the fact that the graph is metric and $sp(r, j)$ will provide the shortest cable-distance to j. We can conclude that the cost of T^* could be reduced by replacing (i, j) by the path $sp(r, j)$, which contradicts the fact that T^* is an optimal solution.

□

2.3.3 Reduction rule 3

(a) Heuristic solution T (b) $cost_T(j) < cost_{min}(i, j)$

Figure 2.4: Reduction 3

This is a heuristic reduction that, in our experiments, was capable of removing a large number of edges, and did not compromise the quality of the solutions found. The idea of this heuristic reduction rule is to remove edges that will not locally produce better results than a given start solution.

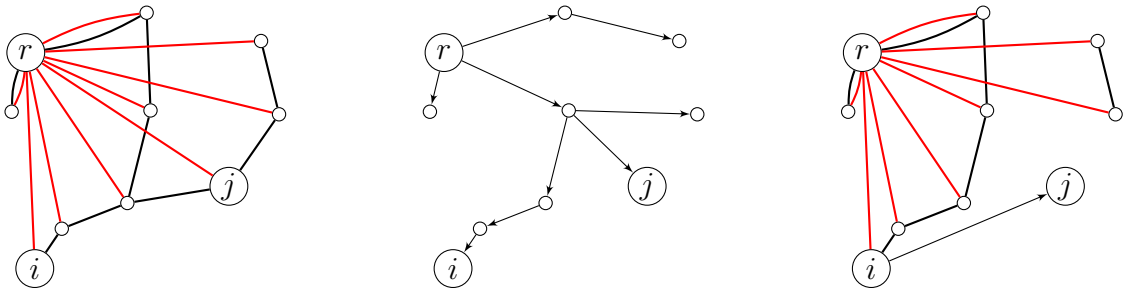
In Figure 2.4a, we have a solution tree T. Figure 2.4b, shows the lowest possible cost that vertex i can reach j, with the same calculation as in (2.16). Knowing the cost that j is connected in the solution T, we decide if the edge ij should be kept.

Let $cost_T(r, j)$ be the cost of connecting j in T. By connection cost, we mean, the cable from r to j in T, which its distance is given by $\ell_T(r, j)$, plus the trench of the arc $\delta^-(j) \in T$, $cost_T(r, j) = (\gamma \cdot \ell_T(r, j)) + (\tau \cdot l_{\delta^-(j)})$. The reduction rule removes every (i, j) where $cost_{min}(i, j) > \beta \cdot cost_T(r, j)$.

Formally, we can define rule 3 as:

> **Reduction Rule 3:** Let an instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$, where $G = (V, A)$, a reduction parameter $\beta \in [0, \infty]$, and a solution tree T. The reduced graph is $G^* = (V, A^*)$, where $A^* = \{(i, j) \in A : cost_{min}(i, j) \leq \beta \cdot cost_T(r, j)\} \cup \{(r, i) : i \in V\}$.

2.3.4 Reduction rule 4



(a) Lower Bound (b) Heuristic solution T (c) LB + edge $\{i, j\}$

Figure 2.5: Reduction 4

We present a new safe reduction rule where, given a feasible solution, we remove all edges that cannot be part of a solution with a lower cost than it. To decide if an edge can be part of a better solution, we calculate the lower bound of the instance, forcing the existence of this edge on the solution. If this lower bound cost, added with the edge have a higher cost than the given solution, we can prove that this edge cannot not be part of any optimal solution, and therefore, can be safely removed.

Let an arc $a \in A$, a function $LB(a)$ that calculates a lower bound where a belongs to the solution, and a feasible solution T for the problem. If $LB(a) > cost(T)$, we can safely remove arc a from the instance as it cannot be part of any solution with a lower value than the one found in T.

Note that for a graph $G = (V, A)$, where $a \in A$, given a function $LB(G)$ that calculate a lower bound of a graph, we can create a subgraph G' of G where arc a is contracted, and calculate $LB(a)$ as $LB(a) = LB(G') + (\tau + \gamma) \cdot l_a$.

Formally, we can define rule 4 as:

Reduction Rule 4: Let an instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$, where $G = (V, A)$, and a solution tree T. The reduced graph is $G^* = (V, A^*)$, where $A^* = \{(i, j) \in A : cost(T) \geq LB(i, j)\} \cup \{(r, i) : i \in V\}$.

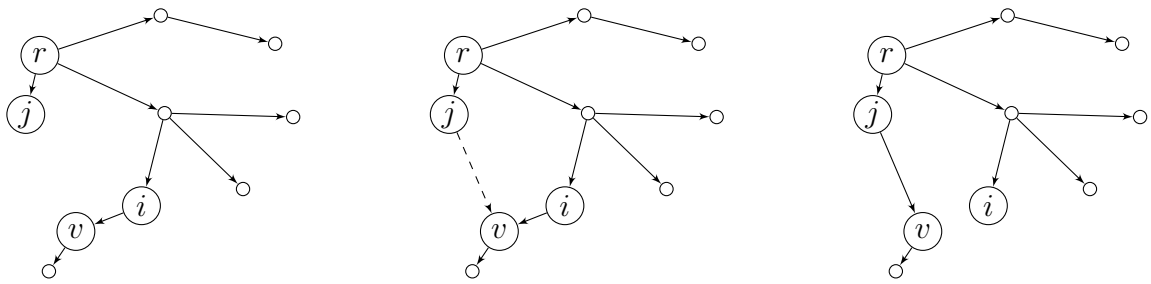
Lema 2.3.1 *Reduction Rule 4 is safe.*

Proof. Let us assume, with the aim of obtaining a contradiction, a feasible solution tree T, and an optimal solution tree T^*, where for an edge $a \in T^*$, $cost(T) < LB(a)$. We have that, since LB(a) gives a valid lower bound, $LB(a) \leq cost(T^*)$, which implies $cost(T) < cost(T^*)$, which is a contradiction to the fact that T^* is optimal. ∎

2.4 Local Search

Working with heuristics, we can explore the local neighbourhood of a solution looking for improvements. We present a fast local search procedure capable of searching for local improvements on any solution.

2.4.1 Single Edge Neighborhood



(a) Heuristic solution T (b) Single edge exchange (c) Solution T'

Figure 2.6: Single Edge Neighborhood

Marianov et al. [2012] discussed on their paper about exploring the local neighbourhood of a heuristic solution, searching for improvements for the p-Cable Trench Problem. Based on it, we focused on designing an algorithm able to quickly calculate local improvements for large scale instances of the CTP.

We are looking for a fast algorithm, able to calculate the cost of replacing an edge in a solution tree T. For every vertex $v \in T$, the algorithm checks if changing v's parent would improve the solution, as illustrated in Figure 2.6.

Given an instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$ and a solution tree T, let us assume, that a vertex $v \in T$ has a vertex $i \in T$ as its parent. Our goal is to calculate the cost of replacing the current parent of v to a vertex $j \in T$ by removing the edge $(i, v) \in T$ and adding the edge (j, v). For the moment, let us assume that it results in a feasible solution, that is, it will not lead to cycles.

To calculate the cost of disconnecting v from T, we need the distance of v to the *root* in T, let us denote this distance by $\ell_T(r, v)$. The size of the sub-tree where v is the root gives how many cables passes through v, let us call the number of vertices on v's sub-tree (with v included) as $|T_v|$.

Removing edge (i, v) from T, reduce its cost by,

$$\text{rem}(i, v) = (\gamma \cdot \ell_T(r, v) \cdot |T_v|) + \tau \cdot l_{iv} \tag{2.17}$$

Let $cost(T)$ be the function that calculates the cost of a solution. Removing edge (i, v) from T would result in a spanning forest F, the cost of F is given by, $\text{cost}(F) = \text{cost}(T) - \text{rem}(i, v)$.

In F, the cost of adding (j, v) have a similar structure, we need the distance of j to the root, $l_T(j)$. The increase in cost is given by,

$$\text{add}(j, v) = (\gamma \cdot (\ell_T(r, j) + l_{jv}) \cdot |T_v|) + \tau \cdot l_{jv}, \tag{2.18}$$

resulting in a spanning tree T' where $\text{cost}(T') = \text{cost}(F) + \text{add}(j, v)$. Therefore, $\Delta(T, T') = \text{add}(j, v) - \text{rem}(i, v)$.

In every scenario where Δ is negative, we have a local improvement. Notice that after pre-processing and storing those initial pieces of information regarding the distance of

every node to the root, and the size of each vertex's sub-tree, we can calculate in constant time each parent change, and in linear time on the number of edges of the original graph G all local improvements.

Algorithm 1 Local Search - Single Edge Improvement Search Algorithm

1: **procedure** SINGLEEDGEIMPROVEMENTSEARCH$(G, T, \ell_T, T_{Size}, \gamma, \tau)$
2: $\quad localImprovements \leftarrow []$ ▷ Solution Vector
3: $\quad$ **for** each $v \in T$ **do**
4: $\quad\quad vParent \leftarrow T[v]$ ▷ $T[v]$ gives v's parent
5: $\quad\quad subTreeSize \leftarrow T_{Size}[v]$ ▷ $T_{Size}[v]$ gives v's sub-tree size
6: $\quad\quad currCableDist \leftarrow \ell_T[v]$ ▷ $\ell_T[v]$ gives v's distance to Root in T
7: $\quad\quad remCost \leftarrow \gamma(currCableDist * subTreeSize) + \tau(edge(vParent, v))$
8: $\quad\quad$ **for** each j where $edge(j, v) \in G \setminus T_v$ **do** ▷ j is not in v's sub-tree
9: $\quad\quad\quad newCableDist \leftarrow \ell_T[j] + edge(j, v)$
10: $\quad\quad\quad addCost \leftarrow \gamma(newCableDist * subTreeSize) + \tau(edge(j, v))$
11: $\quad\quad\quad \Delta \leftarrow addCost - remCost$
12: $\quad\quad\quad$ **if** $\Delta < 0$ **then** ▷ This change improve the solution cost
13: $\quad\quad\quad\quad localImprovements(\Delta, j, v)$
14: $\quad$ **return** $localImprovements$

In the loop of line 3 of the Algorithm 1, it iterates over each vertex of the solution tree T, it stores in $remCost$ the cost of disconnecting its current parent. In line 8 it looks for a new parent for the vertex, searching in every other vertex that have a connection to v and is not in v's sub-tree. For each edge replacement candidate, it calculates the cost of this new connection and store in $addCost$. If it achieves a local improvement by finding a negative value of Δ, that is, if the cost of adding this new edge is lower than the cost of removing the current one, it appends it in the $localImprovements$ list.

Considering that the algorithm know the size of the sub-tree of each vertex in T, and its distance to the root vertex, it can calculate each edge replacement in constant time. Representing the graph as an adjacency list, we can achieve a time complexity of $O(|E|)$.

Parallelization

In the Algorithm 1, notice that lines 3 and 8 are data independent loops, therefore, we can perform them in parallel. The only critical point is when inserting an item on the $localImprovements$' list. Aware of this, a set of vertices can be simultaneously verified, speeding up the local search procedure.

2.4.2 Local Search Procedure

We use the Single Edge Neighborhood in a local search procedure, described bellow.

Algorithm 2 Local Search Procedure Algorithm

```
procedure LocalSearch(G, T, r, γ, τ, timeLimit)
    while changes.empty ≠ true ∧ didReachTimeLimit(timeLimit) ≠ true do
        d_T ← calculateDistanceToRoot(T)
        T_size ← getSizeOfSubtrees(T)
        changes ← SingleEdgeImprovementSearch(G, T, d_T, T_size, γ, τ)
        changes ← Sort(changes)
        ApplyToTree(changes.getFirst, T, G, γ, τ)
    return T
```

The Algorithm 2 iterates over our solution tree T until it reaches the time limit or it could not find more local improvements. An auxiliary function $calculateDistanceToRoot(T)$ returns a vector with the distance of every vertex in T to the root. Another function $getSizeOfSubtrees(T)$ returns the number of vertices on each vertex sub-tree. The algorithm calculates the number of local improvements, sort all the possible changes by its Δ value, and apply the most significant change to the solution tree T, repeating the process.

2.5 Heuristics

2.5.1 Mod-Prim

Presented by Vasko et al. [2016], Mod-Prim is a greedy heuristic, based on the Prim's Algorithm for Minimum Spanning-Tree (Prim [1957]). Mod-Prim can quickly generate a feasible solution even for large-scale instances of the CTP.

Starting with a CTP instance $(G, r, l, \tau, \gamma) \in \mathcal{I}_{CTP}$, the algorithm initializes a vector of the distances to every node in G to the root r as infinite. It starts a solution tree T with the root. Them, calculates the cost of reaching each node from the root, accounting for the cable and trench costs. It them connects the "cheapest" node to the solution tree. For each node inserted in T, it updates if needed, the lowest connection cost known to every other node that is not in T. The algorithm keeps inserting the node with the lowest connection cost that is not in T until every node of G is connected to T.

The pseudocode is presented as follows in Algorithm 3,

Algorithm 3 Mod Prim Algorithm

1: **procedure** MODPRIM(G, r, γ, τ)
2: distanceToRoot $\leftarrow \{\infty_0, \infty_1, \ldots, \infty_{|V|}\}$
3: bestCost $\leftarrow \{\infty_0, \infty_1, \ldots \infty_{|V|}\}$
4: parent $\leftarrow \{\}$
5: T $\leftarrow \{\}$
6: bestCost[r] $\leftarrow 0$
7: distanceToRoot[r] $\leftarrow 0$
8: **while** $|T| \neq |V|$ **do**
9: nextNode $\leftarrow argmin(bestCost[i])$ $\triangleright\ i \in (G \setminus T)$
10: **for** $v \in neighborhood(nextNode)$ **do**
11: edgeLength $\leftarrow l(nextNode, v)$
12: cost $\leftarrow \tau \cdot edgeLength + \gamma \cdot (distanceToRoot[nextNode] + edgeLength)$
13: **if** $cost < bestCost[v]$ **then**
14: parent[v] $\leftarrow nextNode$
15: distanceToRoot[v] $\leftarrow$ distanceToRoot[nextNode] + edgeLength
16: bestCost[v] $\leftarrow$ cost
17: $T \leftarrow \{nextNode\} \cup T$
18: **return** parent

In lines 2 to 7, we initialize variables that store the distance to the root, and the best-known connection cost to the solution tree. We also set the connection cost and cable distance of the root as 0. The loop in line 8 iterates until every vertex in the graph is in the solution tree. In line 9 it searches the vertex with the lowest connection cost that is not in T yet. Line 10, from this nextNode, we update the distance to all vertex not in T, changing in line 13 if finding a better connection cost.

Using a heap to make the search for the vertex with lowest connection cost, we can achieve time complexity of $O(|E| \log(|V|))$ for the Mod Prim algorithm.

2.5.2 GRASP and Path Re-link

In this section, we present a new heuristic based on Greedy Randomized Adaptive Search Procedure (GRASP). It consists of an iterative meta-heuristic, that is based on a constructive phase and local search phase (Resende and Ribeiro [2003]).

The constructive phase generates greedily randomized solutions, looking for a set of diverse and feasible solutions to the instance. The local search phase looks for local improvements over the given solutions. This process is repeated for many iterations and

is demonstrated by previous works on the literature to be a good heuristic approach for many hard problems.

Constructive Phase

For the constructive phase, the algorithm builds a feasible solution by using a greedly-randomized function. The algorithm is based on the Mod Prim algorithm, presented in Section (2.5.1). When building a solution tree, instead of greedily selecting the vertex of minimum cost that is not yet on the solution tree, it build a list of candidate choices that can be incorporated in the solution, which is called by GRASP's literature as the Restricted Candidate List(RCL), and randomly selects one of them to be incorporated to the solution.

We will set an α factor that, based on the best "greedy choice", will limit the candidates that could be selected. From the RCL the algorithm will randomly pick a vertex that will be added to the solution, and repeat this process until it reaches a feasible solution tree.

This randomized algorithm is presented in Algorithm 4.

Algorithm 4 Semi-Greedy ModPrim

1: **procedure** RANDOMIZEDMODPRIM(G, α, γ, τ)
2: distanceToRoot $\leftarrow \{\infty_0, \infty_1, \ldots \infty_{|V|}\}$
3: bestCost $\leftarrow \{\infty_0, \infty_1, \ldots \infty_{|V|}\}$
4: bestCost[root] $\leftarrow 0$
5: distanceToRoot[root] $\leftarrow 0$
6: parent $\leftarrow \{\}$
7: T $\leftarrow \{\}$
8: **while** $|T| \neq |V|$) **do**
9: $RCL \leftarrow \{\}$
10: **repeat**
11: RCL $\leftarrow$ RCL $\cup argmin(bestCost[v])$ $\triangleright$ $v \in G \setminus (T \cup RCL)$
12: maxCostToRCL $\leftarrow \alpha \cdot (max(bestCost) - cost(RCL.first))$
13: costOfNextRCLCandidate $\leftarrow (cost(RCL.last) - cost(RCL.first)$
14: **until** $|V| = |T \cup RCL| \vee (costOfNextRCLCandidate > maxCostToRCL)$
15: nextNode $\leftarrow$ randomElementAt(RCL) $\triangleright$ Randomly selects from RCL
16: **for** $v \in$ neighborhood(nextNode) **do**
17: edgeLength $\leftarrow$ l(nextNode,v)
18: cost $\leftarrow \tau \cdot edgeLength + \gamma \cdot (distanceToRoot[nextNode] + edgeLength)$
19: **if** cost $<$ bestCost[v] **then**
20: parent[v] $\leftarrow$ nextNode
21: distanceToRoot[v] $\leftarrow$ distanceToRoot[nextNode] + edgeLength
22: bestCost[v] $\leftarrow$ cost
23: T $\leftarrow \{nextNode\} \cup T$
24: **return** parent

The algorithm randomly select the next vertex to be added to the solution tree from the RCL set. In the loops from line 10 to 14, it will add to the RCL the cheapest vertex that is not in the RCL and not in the solution tree until the condition on line 14 is reached. In line 15, it randomly pick an item from the RCL, and connects it to the tree in a similar manner as in the Mod Prim algorithm.

Local-Search Phase

The Local-Search phase looks for improvements on the greedy solutions found in the constructive phase. Our GRASP based heuristic uses the Local Search procedure presented in Section 2.4.

Path-Relinking

Path-Relinking is another meta-heuristic used to find improved solutions in the neighbourhood of "high-quality solutions". Given two solution trees, a base solution T_b and a target solution T_t, we calculate the "distance" of the base solution to the target solution. By distance, we mean the number of vertices with a different parent in the base solution, when compared to the target solution.

The goal of our path relinking algorithm is to search for the best solution in the neighbourhood of both given solutions, by moving the base solution towards the target solution, changing its parents, until it closes its distance.

Given the set of best solutions found, it considers every pair of solutions as a base and target solutions. For each pair it does iterative changes until it converts the base solution into the target solution. In each iteration, it evaluates the solution cost, retaining the best one found in the process.

If the resulting solution tree of an iteration is infeasible, we consider the solution with an infinite cost and keep changing the vertices' parents until we reach a feasible solution again (considering the target solution is feasible, we know that this will eventually happen).

Algorithm 5 CTP PathRelink Algorithm

```
procedure PathRelink (G, T_b, T_t, γ, τ)
    T_Best ← T_b
    T_Local ← T_b
    for each v ∈ T_b do
        if T_b[v] ≠ T_t[v] then
            changeParents(T_Local, v, T_t[v])
            if calcCost(T_Local, γ, τ) < calcCost(T_Best, γ, τ) then
                T_Best ← T_Local
    return T_Best
```

We present the pseudocode of this path relinking in Algorithm 5. Path relinking algorithm receives two solution trees: T_b and T_t. It starts with T_b as a base solution, and change, in sequence, each different parent vertex until it reaches the same solution as T_t. Note that this changes can lead to an invalid solution (e.g with loops). In this case, the cost of the infeasible solution is infinity. We repeat this process until there is no more vertices to change.

GRASP + Path-Relinking Applied to the CTP

Combining these algorithms, we have our final heuristic. It runs for a number of iterations the greedy randomized algorithm, apply a local search on it, and at the end, in the best solutions found, it uses a path relinking to search for improvements, returning the best solution found in the process. The final GRASP heuristic is presented in Algorithm 6.

Algorithm 6 GRASP based heuristic

1: **procedure** GRASP($G, iterations, eliteSize, \alpha, \gamma, \tau$)
2: solutions $\leftarrow \{\}$
3: **while** $i <$ iterations **do**
4: $T_{solution} \leftarrow RandomizedModPrim(G, \alpha, \gamma, \tau)$
5: $\{solutionValue, T_{localSearchSolution}\} \leftarrow LocalSearch(G, T_{solution}, \gamma, \tau, false)$
6: $solutions \leftarrow solutions \cup \{solutionValue, T_{localSearchSolution}\}$
7: elite $\leftarrow$ getEliteFromSolutions(solutions, eliteSize)
8: $T_{best} \leftarrow$ PathRelink (elite)
9: **return** T_{best}

The loop in line 3 defines the number of iterations that will be executed. For each iteration we generate a solution, apply the local search and append it to the solutions set. After generating those solutions, we get the elite set with the best results found. In line 8 we apply the path-relinking in the elite set and return the best solution found.

Parallel GRASP

It is worth mentioning that the constructive, local search and path-relinking phases are data independent. So, each of those steps could run in parallel. We could generate multiple solutions at once, using a local search with multiple cores, and path-relinking multiple pairs at the same time.

Chapter 3

The Cable Trench Problem with Demands

We present a variant for the Cable Trench Problem, that we call the Cable Trench Problem with Demands (CTPD), which add a cable demand to the set of vertices. This variant is used in a new technique to solve large-scale instances of the CTP, presented in Section 3.3.

In the Cable-Trench Problem with Demands each vertex in the graph has a positive demand, as if each client is requesting a certain number of cables from the root.

We can formally define the Cable-Trench Problem with Demands as:

> **Cable-Trench Problem with Demands:** An instance $(G, r, l, D, \tau, \gamma) \in \mathcal{I}_{CTPD}$ consists of a graph $G = (V, E)$, a root vertex $r \in V$, a function $l : E \to \mathbb{R}^+$ which gives the length of each edge, cost factors τ and γ for trench and cables, and a positive demand factor $d_i \in D$ for each $i \in V$. The problem is to find a spanning tree $T = (V, E_T)$ of G in which $\tau \sum_{e \in E_T} l_e + \gamma \sum_{v \in V} \ell_T(r, v) \cdot d_v$ is minimum.

3.1 CTPD Formulations

We can adapt the CTP linear programming models presented in Section 2.1 to the CTPD problem. We denote by D the vector of demands of every vertex in G.

3.1.1 Single-Commodity Formulation

We present below, a MILP formulation for the CTPD. Given an instance $(G, r, l, D, \tau, \gamma) \in \mathcal{I}_{CTPD}$, where $G = (V, A)$, for each arc $ij \in A$, we create an integer variable x_{ij}, that gives the number of cables on arc ij, and a binary variable y_{ij}, that indicates the existence of a trench on edge ij.

$$\text{Minimize} \quad \gamma[\sum_{a \in A(E)} l_a x_a] + \tau[\sum_{a \in A(E)} l_a y_a], \tag{3.1}$$

subject to:

$$\sum_{a \in \delta^+(r)} x_a = \sum_{i \in (v \setminus r)} d_i \tag{3.2}$$

$$\sum_{a \in \delta^+(i)} x_a - \sum_{a \in \delta^-(i)} x_a = -d_i \qquad \forall i \in (V \setminus r), \tag{3.3}$$

$$\sum_{a \in A(E)} y_a = n - 1, \tag{3.4}$$

$$x_a \leq (\sum_{(i \in v \setminus r)} d_i) y_a \qquad \forall a \in A(E), \tag{3.5}$$

$$x_a \geq 0 \qquad \forall a \in A(E), \tag{3.6}$$

$$y_a \in \{0, 1\} \qquad \forall a \in A(E). \tag{3.7}$$

Constraint (3.2) sets the sum of all cables demands in the instance as the amount of cables that should be provided by the root. In constraints (3.3), it sets that node $i \in V$ should receive d_i units of cable. Constraints (3.5) sets that if a trench is dug, it can hold all cable demanded in the graph. All other constraints remains the same as the ones presented for the CTP, in Section 2.1.1

3.1.2 Multi-Commodity Formulation

We present a CTPD formulation, based on the one in Section 2.1.2. Given an instance $(G, r, l, D, \tau, \gamma) \in \mathcal{I}_{CTPD}$, where $G = (V, A)$, for each arc $ij \in A$, we create a variable f_{ij}^k that indicates if the cable that reach vertex k passes in arc ij, and a boolean variable y_{ij} that indicates the existence of a trench on arc ij.

$$\text{Minimize} \quad \gamma[\sum_{k \in V} \sum_{a \in A(E)} l_a f_a^k] + \tau[\sum_{a \in A(E)} l_a y_a], \tag{3.8}$$

subject to:

$$\sum_{a \in \delta^+(r)} f_a^k = d_k \qquad \forall k \in V, \tag{3.9}$$

$$\sum_{a \in \delta^-(k)} f_a^k = d_k \qquad \forall k \in V, \tag{3.10}$$

$$\sum_{a \in \delta^+(i)} f_a^k - \sum_{a \in \delta^-(i)} f_a^k = 0 \qquad \forall k \in V, i \in V \setminus k, \tag{3.11}$$

$$\sum_{a \in \delta^-(i)} y_a = 1 \qquad \forall i \in V, \tag{3.12}$$

$$f_a^k \leq y_a \cdot \sum_{(i \in v \setminus r)} d_i \qquad \forall a \in A(E) \setminus \delta^-(k), \tag{3.13}$$

$$f_a^k \geq 0 \qquad \forall a \in A(E), k \in V, \tag{3.14}$$

$$y_a \in \{0, 1\} \qquad \forall a \in A(E). \tag{3.15}$$

Constraint (3.9) sets the number of cables leaving the root equals to the vertex demand. Constraint (3.10) states the number of cable units to be received for each vertex.

In constraint (3.13), it adds the sum of demands in D, in order to allow a trench to hold, at most, all cables on it. All other constraints remains the same as the one presented for the CTP, in Section 2.1.2

Thanks to a comment made by Professor Usberti, we later notice that considering the cable cost factor of a vertex k as $\gamma \cdot d_k$, it is possible to model its demand as the cable price that should reach k, that is, if a vertex have a demand of 10 cables, it is equivalent to state its cable price factor as 10γ.

Therefore, we could take advantage of the Multi-Commodity Formulation and model the CTPD as a Multi-Commodity Cable Trench Problem, setting a different cable cost factor for each cable commodity based on its demand. As consequence, the model could be much similar to the one presented in Section 2.1.2, requiring only the addition of an specific cable cost factor for each vertex. Notice that this new formulation would not even require modification in the constraints. Even so, this was not explored in this research and is left as future work discussed in Section 5.

3.2 Mod-Prim with Demands

We can modify the Mod Prim algorithm presented in Section 3 to solve the version of the problem with demands.

In the Mod Prim algorithm, to calculate the connection cost from $nextNode$ to v, we have:

$$cost \leftarrow \tau \cdot edgeLength + \gamma \cdot (distanceToRoot[nextNode] + edgeLength) \tag{3.16}$$

To solve the version with demands, we have to account for the number of cables that should reach a vertex v. So, to calculate the cost delivering d_v cables to v, we modify the cost equation to:

$$cost \leftarrow \tau \cdot edgeLength + \gamma \cdot d_v \cdot (distanceToRoot[nextNode] + edgeLength) \quad (3.17)$$

Keeping the same pseudocode presented in Section 3, and only modifying the cost formula, we have a version of Mod Prim algorithm for the Cable Trench Problem with Demands.

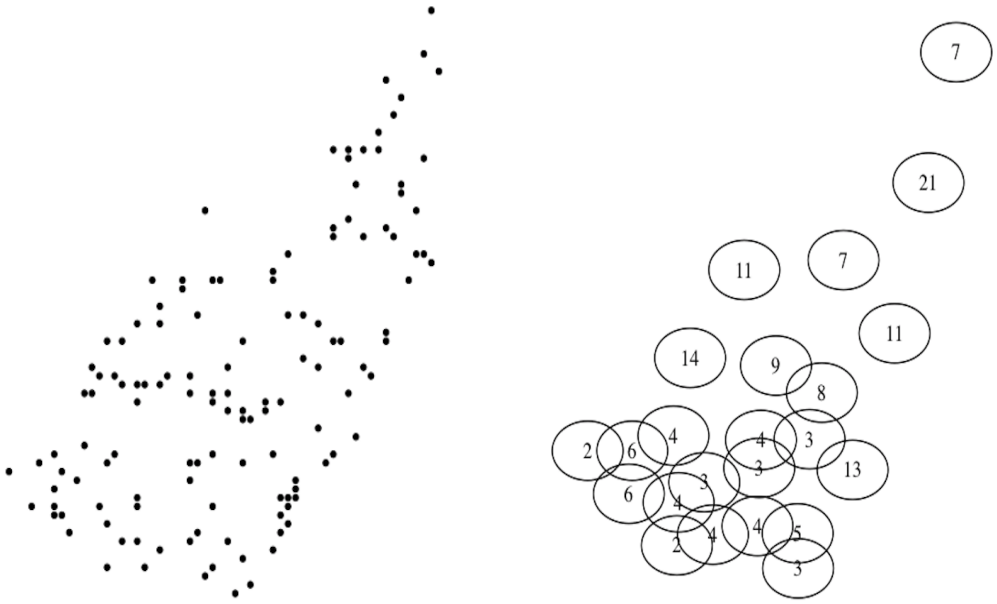
3.3 CTPD applied to large-scale instances of the CTP

We present a new technique to solve large-scale instances of the Cable Trench Problem by reducing them into vertex-wise "smaller" instances of the CTPD. We start by partitioning the set of vertices of the CTP instance into clusters. For each of those clusters, we create a vertex in a CTPD instance. We define the demand of each vertex equal to the size of its correspondent cluster. We demonstrate that a solution of this CTPD instance can be used to build a feasible network design for the original CTP instance.

Dealing with those "smaller" CTPD instances enables the use of a new range of techniques, such as linear programming based heuristics, and we also demonstrate that it can even improve results on greedy algorithms such as Mod-Prim.

We present in Section 3.3.1 details on how to reduce a CTP instance into a CTPD instance. Section 3.3.2 show details about how the algorithmic framework works.

3.3.1 Representing Large-Scale CTP instances as CTPD instances



(a) CTP - 151 vertices (b) CTPD - 23 vertices

Figure 3.1: Example of CTP instance into CTPD instance

To solve large-scale instances, we aim to represent, as best as possible, a large scale CTP instance as a vertex-wise "smaller" instance. Figure 3.1 shows an example where we represent an instance with 151 vertices as an instance with demands of 23 vertices.

Given a graph $G = (V, E)$, we start by choosing k partitions, $\{C_0 \subset V, C_1 \subset V, \ldots, C_k \subset V\}$, where $\bigcup_{i=0}^{k} C_i = V$. We call each of these partitions, a cluster of vertices.

We create a CTPD instance where for each cluster C_i we create a vertex i'. To better represent the original CTP instance, we set a demand to each vertex equal to the size of its cluster, $|C_i|$. The position of each vertex is given by the position of the vertex closest to the correspondent geometric centroid of its cluster. We add an edge between any pair of vertices, if exist a pair of vertices in the correspondent clusters that also have an edge between them. The root of this new instance is defined by the cluster that have the root vertex of the CTP instance. Cable and trench prices remains the same as the original instance.

It is worth mentioning that alternative representations may use different rules or metrics to represent a CTP instance as a CTPD instance, but due to time limitations we did not explored it on this research.

Details on the algorithm used in this research to compress the graph into clusters is presented in the following section.

K-Clustering

Looking for a representative way to segment our graph into clusters, we use an iterative process similar to the one used in the k-means clustering algorithm (MacQueen [1967]).

Figure 3.2: CTP Instance

Figure 3.3: Step 1

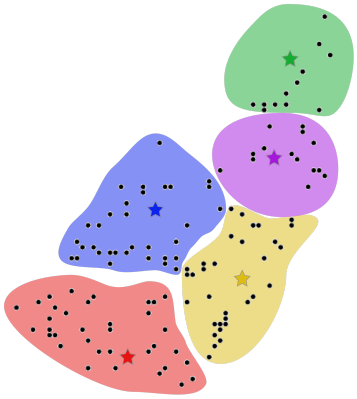


Figure 3.4: Step 2

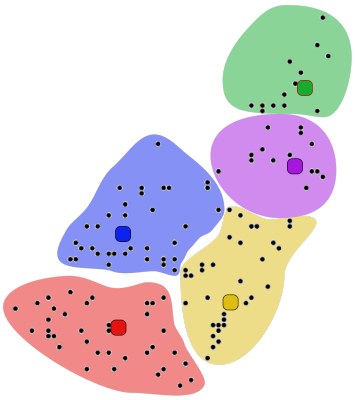


Figure 3.5: Step 3

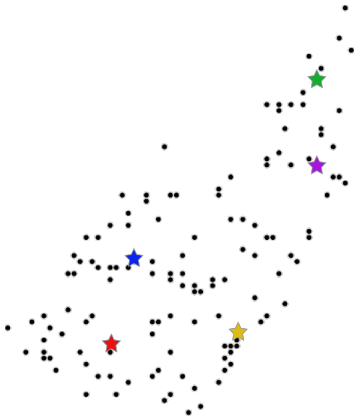


Figure 3.6: Step 4

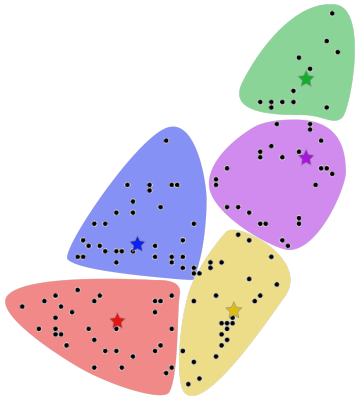


Figure 3.7: Step 5

Figure 3.8: K-Clustering algorithm adapted to CTP clustering

Let us denote by center-vertex, a vertex that represent the center of a cluster. The K-Clustering algorithm adapted to CTP instance clustering can be summarised by the following steps, illustrated in Figure 3.8:

1. Randomly choose a set of vertices of size k which are the initial center-vertices.
2. Add each vertex to the cluster correspondent to its closest center-vertex.
3. Calculate the geometric centroid of each cluster
4. For each cluster, update the center-vertex as the one closest to its centroid.
5. Repeat steps 2, 3 and 4 until it converges, or reach a limit number of iterations.

At the end, it results in all vertices of a CTP instance divided into k clusters.

3.3.2 Reduce and Solve - CTPD based Heuristic

We present an algorithmic framework to solve large-scale CTP instances. It requires an algorithm capable of solving a CTPD instance, that we call *CTPD_Solver*, an algorithm to reduce a CTP instance into a CTPD instance, *CTP_Reducer*, and an integer k that is the number of vertices (or clusters) that will be created for the CTPD instance.

The algorithm, from a top-down perspective, solves the problem in "levels", diving into each cluster, and building a solution tree that reaches every vertex on it.

It first applies the reduction algorithm on the input instance, to create a CTPD instance of size at most k. It solves it, building a solution that we call T_0. For each vertex of T_0, starting from the root, it "opens" the correspondent cluster, getting the vertices that it contains, and solving it. By solve it, we mean that, based on the set of vertices, and on the solution tree T_0, it builds a new CTPD instance, adjust its demands, and recursively solve it.

Going in depth on each vertex, eventually, the algorithm will hit a unitary cluster. A unitary cluster is our base case, when the cluster represent only one vertex of the original CTP instance. At this point, a path in the solution tree to this vertex will already be created. At the end, we have a valid network that connects every vertex in the graph, and that is a valid solution tree for the CTP instance.

Algorithm 7 presents the pseudocode for the heuristic.

Algorithm 7 Reduce and Solve Heuristic Algorithm

```
1: procedure ReduceAndSolve(G, r, D, γ, τ, k, CTPDSolver, CTPReducer)
2:     if |V| = 1 then
3:         return []                                  ▷ The set of edges for one vertex is empty
4:     else
5:         CTPDInstance ← CTPReducer(G, r, τ, γ, k)
6:         CTPDInstance.D ← SumDemands(CTPDInstance, D)
7:     T_0 ← CTPDSolver(CTPDInstance)
8:     clusterToBeSolved ← {CTPDInstance.root}                ▷ queue of vertices
9:     SolutionTree ← []
10:    while clusterToBeSolved.isEmpty = false do             ▷ Building a solution
11:        currentCluster ← clusterToBeSolved.pop()
12:        clusterParentVertex ← GetClosestVertex(T_0[currentCluster], currentCluster)
13:        r' ← getRootFrom(currentCluster)
14:        SolutionTree.insert( {clusterParentVertex, r'})
15:        G' ← getGraphOfCluster(currentCluster)
16:        D' ← [1, 1, ..., 1]                    ▷ D' receives a vector of size |V'| of 1
17:        for i ∈ δ_{T_0}(currentCluster) do
18:            closestVertexToCluster ← GetClosestVertex(i, currentCluster)
19:            D'[closestVertexToCluster] ← getACMSizeOfCluster(i)
20:        ClusterSol ← ReduceAndSolve(G', r', D', γ, τ, k, CTPDSolver, CTPReducer)
21:        SolutionTree.insert(ClusterSol)
22:        for i ∈ δ_{T_0}(currentCluster) do
23:            clusterToBeSolved.insert(i)
24:    return SolutionTree
```

The input of Algorithm 7, is a CTPD instance. Note that we can trivially reduce a CTP instance into a CTPD instance where each vertex have its demand equal to one unit.

As a recursive algorithm, our base case, in line 2, is when the number of vertices of an instance is equal to one. At this point, the solution, that is composed by the edges in the solution tree, is trivially empty.

If the instance is vertex-wise bigger than k, the algorithm must reduce its size. The algorithm uses the *CTP_Reducer* algorithm to create, from the input graph G, an instance with size at most k.

As our input is also a CTPD instance, only representing this graph is not enough,

due to the fact that each vertex may have a previous demand. So the algorithm must also account from previous demands bigger than one unit. In line 6, it calls a function $SumDemands$ that will sum the demands of D, bigger than 1, to the respective vertices in CTPDInstance.

In line 7, it solves CTPDInstance. In line 8 it creates a clusterToBeSolved queue, that will be used to go through every cluster in T_0 to, sequentially, "open" and solve each cluster vertex.

The loop on line 10 goes until there is no other vertex to visit, starting from the root, it goes through all vertices in T_0.

Line 12 gets the current cluster parent. First, $T_0[currentCluster]$ returns the cluster that is the parent of currentCluster in T_0. The function GetClosestVertex gets the vertex that is the closest in the parent cluster, to the current one.

In line 13 it gets the current parent root, if this cluster is the root cluster of T_0, the original root is chosen, if not, it selects the closest vertex in the current cluster to the parent cluster as root.

In line 14 it adds the edge that connects the parent's closest vertex to the current root r', to the solution.

At this point on, the algorithm starts to create a new instance based on the current cluster, aiming to recursively build a feasible CTP solution for the currentCluster.

In Line 15 it gets the graph correspondent to the vertices in the currentCluster. Line 16 it creates a new demand vector, at first, with demand equal to one to each vertex.

In the loop of lines 17 to 19, it iterates over every cluster adjacent to the currentCluster in T_0, the goal is to add as demand to the currentCluster the size correspondent to all neighbours. First, in line 18, it calls the function GetClosestVertex, that returns the closest vertex in the current cluster to the cluster i, storing in the variable closestVertexToCluster. The idea is to add to this closest vertex, the demands correspondent to the subtree of clusters rooted by i in T_0.

That is what happens in line 19, where the demand vector, in the position of the vertex represented by the variable closestVertexToCluster, gets the accumulated demand of subtree rooted by i, given by the function getACMSizeOfCluster(i).

At this point, in line 20, it can recursively call itself with an instance correspondent to the current cluster. In line 21 we add the returning result to the current SolutionTree, and in lines 22 and 23 it adds the adjacent clusters to the clusterToBeSolved queue.

When every vertex of T_0 is solved, a valid CTP solution for this tree is stored in variable SolutionTree, and returned in line 24.

Chapter 4

Computational Experiments

We run our experiments on a 8 Cores-Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz, with 32GB of RAM, running Ubuntu 16.04.1 LTS. An instance for the Cable Trench Problem is composed of a graph, a root vertex and a cost factor for cables and trenches. We have 11 classes of Euclidean graphs based on 2D and 3D points, all graphs used in this research can be found on (Rocha [2018]). For each graph, we create an instance with a cable value $\gamma = 1.0$, and trench value equals to $\tau = \{0.01, 1.0, 5.0, 10.0, 100.0\}$, same values as used by Vasko et al. [2016].

Our main instance set, with 310 instances, contain graphs of size ranging from 8.000 to 33.708 vertices. All algorithms presented on this research were run with a time limit of 10 minutes, followed by a local search algorithm limited to 35 seconds.

4.1 Methodology

4.1.1 Graph Classes

Vasko We used the same set of graphs presented by Vasko et al. [2016], with vertices set size of 10.001, 15.001, 20.001, 25.001. This instance is based on points of a vascular image exam where each vertex is mapped as a point in a 3D-space.

Maps Based on the National Traveling Salesman Problems set, we used 6 country maps: Greece with 9.882 vertices, Morocco with 14.185 vertices, Italy with 16.862 vertices, Vietnam with 22.775 vertices, Sweden with 24.978 vertices, and Burma with 33.708 vertices. Each vertex in this class is mapped as a point in a 2D-space.

Spiral We generated 6 graphs with vertices set size of 10.000, 15.000, 15.000, 20.000, 25.000, 30.000, that form a spiral shape in a 2D-space.

Grid2D We generated 5 graphs with vertices set size of 10.000, 19.600, 19.600, 19,600, 28.900, that forms a equally spaced grid in a 2D-space.

01Cluster2D and SparseCluster2D We generated 6 graphs with vertices set size of 10.000, 15.000, 15.000, 20.000, 25.000, 30.000. Each vertex is mapped as a point in a 2D-space. Each of these graphs have a number of clusters between $[1, \sqrt{n}]$, where a cluster is a dense area of vertices. 01Cluster2D limit its coordinates between between x = $[0, 1]$ and y = $[0, 1]$, while SparseCluster2D limit it to x = $[0, n]$ and y = $[0, n]$.

01Random2D and SparseRandom2D We generated 6 graphs with vertices set size of 10.000, 15.000, 15.000, 20.000, 25.000, 30.000. Each vertex is a point in a 2D-space. All vertices follow a uniform random distribution in between the limit coordinates. 01Random2D limit its coordinates between x = $[0, 1]$ and y = $[0, 1]$, while SparseRandom2D limit it to x = $[0, n]$ and y = $[0, n]$.

01Cluster3D, 01Random3D and Grid3D We generated graphs where each vertex is a point in a 3D-space. Each of these classes is analog to the correspondent 2D-space classes, with the addition of a z-axis. With the exception of Grid3D, the vertices size are the same as in the 2D version. For Grid3D, to keep equally spaced vertices, we generated graphs with the vertices set size of 8.000, 8.000, 15.625, 27.000, 27.000.

4.1.2 Instances

For each graph we build a set of 5 instances with cable value equal to 1.0, and trench values equals to $\{0.01, 1.0, 5.0, 10.0, 100.0\}$. The root is chosen as the first vertex of each graph.

Due to the fact that we were not able to store the complete graph on memory, we build our instances using reduction rule 1 with an $\lambda = 0.1$. We discarded every edge longer than 10% of the longest edge, but keep the root directly connected to every node in the graph

We use two instance sets on this research, presented as follows.

Main Instance Set

The main instance set contains all 11 graph classes: Vasko, Maps, Spiral, Grid2D, 01Cluster2D, SparseCluster2D, 01Random2D, SparseRandom2D, 01Cluster3D, 01Random3D and Grid3D. The Main Instance Set have, in total, 310 instances.

Reduced Instance Set

For the reduced instance set, we focus on 3 graph classes **Vasko**, **Maps** and **Spiral**. We chose it because instances of the Spiral set presented the highest average gap to the lower bound, being the most difficult ones for our heuristics. Instances of the Vasko set were presented in a previous work on the literature. The instances of the Maps set are based on a practical scenario for the CTP. The Reduced Instance Set have, in total, 80 instances.

4.1.3 Edge Reduction Rules

We used in our research reduction rules 1, 2 and 3. Reduction rule 1 was used when building our instances, we used factor λ of 10%, the same used in (Vasko et al. [2016]). Reduction rule 2 has no parameters. For reduction rule 3 we chose factor $\beta = 1.0$, discarding edges that did not result in an improvement over our greedy solution given by the Mod Prim algorithm. We leave as future work experiments on reduction rule 4.

4.1.4 Lower Bound

The lower bound, discussed in Section 2.2.1 was taken using a Minimum Spanning Tree + Single Source Shortest Path from the root to every other vertex in the instance.

4.1.5 Time limits

Every algorithm runs with a time limit of 10 minutes, this time was chosen after we run our heuristics on the Reduced Instance Set and notice that after it, no considerable improvement was found. The same idea applies to the time limit of 35 seconds imposed on the local search algorithm.

4.1.6 Heuristics

GRASP and Parallel GRASP

We present the GRASP based heuristic in our experiments in two different modes.

GRASP, where the constructive phase used a single core, and the local search phase, for each solution, used all 8 available cores. This version generated a single solution at a time and parallelized the local search procedure.

In the **Parallel GRASP**, the constructive phase used all 8 available cores, and the local search phase, for each solution, uses a single core. The parallel GRASP generated 8 simultaneous solutions for each iteration.

In the constructive phase of GRASP the size of the RCL list depends on a parameter α, which dictates the cost of elements being considered. We randomly chose an α value for each iteration. It chooses with uniform probability an α in following list: [0.5, 0.25, 0.125, 0.075, 0.03, 0.03, 0.01, 0.01, 0.01, 0.005]. The values on the list were chosen due to previous experiments showing that lower α values presented better results for our instances, but we also added higher values with a lower probability to increase the diversity of the solutions found.

We divided the 10 minutes of the time limit in 6 minutes for the constructive and local search phases, and 4 minutes for path relinking phase (executed in parallel for both of them), where the elite size used was of 5 instances. For both, we also limited the time of the local search to 35 seconds.

Reduce and Solve Heuristic

For the Reduce and Solve heuristic, it uses the K-Clustering as the *CTP_Reducer*. As *CTPD_Solver* we use two algorithms, Mod Prim with Demands and a PLI algorithm based on the model of Section 3.1.2. To solve the PLI algorithm we used the *Gurobi Optimizer 8.0*.

For the parameter k, we used for the Mod Prim algorithm: $\{50, 100, 250, 400, 600\}$, and for the PLI algorithm, values in $\{50, 100\}$. In both cases, we account the time for running all sizes of k and get the best solution value found. In the experimental results, these algorithms are presented, respectively, as **(CMP) Mod Prim** and **(CMP) PLI**.

4.2 Experimental Results

In this Section we evaluate the results of our experiments.

The following image presents our results regarding, for our main instance set, the percentage of instances solved with a particular gap to the lower bound for each of the researched algorithms.

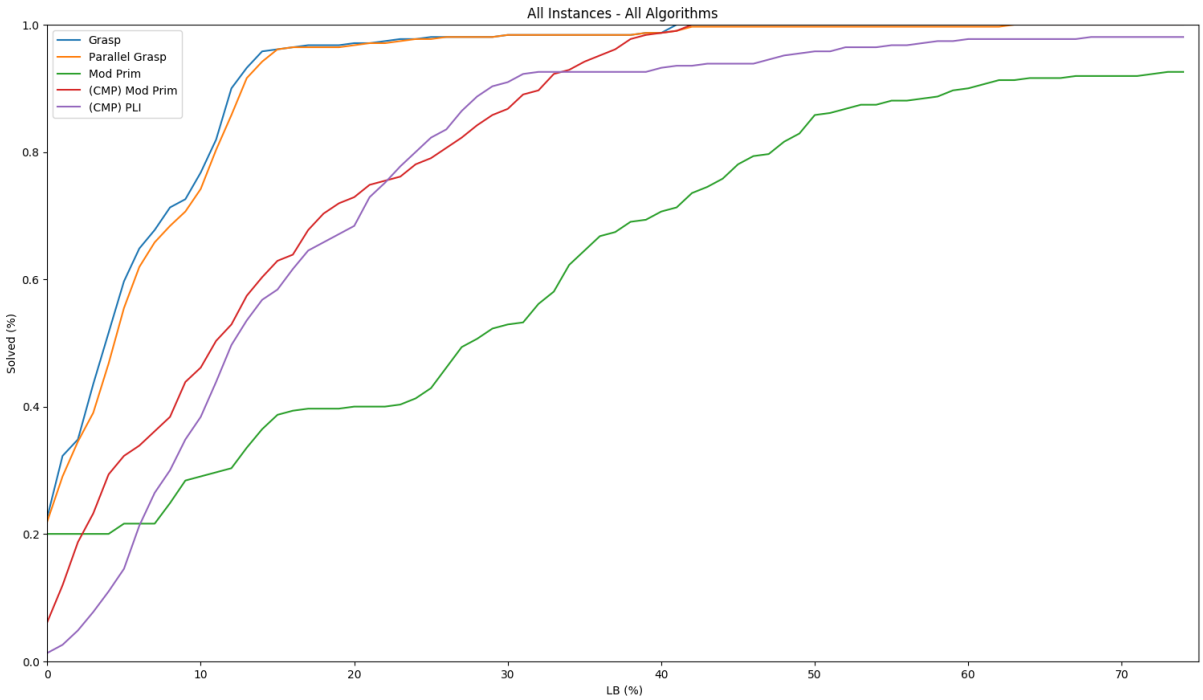


Figure 4.1: Algorithms for the complete instance set

In Figure 4.1, we have in the vertical axis, the percentage of solved instances [0,1]. In the horizontal axis, we have the gap of the solutions to the lower bound.

In Figure 4.1, we can state that GRASP and Parallel GRASP solved close to 70% of instances below the 10% gap and almost 95% of instances below 15% gap. It is also worth highlighting how (CMP) Mod Prim performed better than Mod Prim, while Mod Prim solved only around 40% of instances below the 20% gap, (CMP) Mod Prim solved almost 75% of instances below the same mark.

As stated in Section 4.1.5, we applied a 35 seconds local search after each heuristic. The figure below presents the result, for our main instance set, of the gap to the lower bound of each heuristic after the local search algorithm.

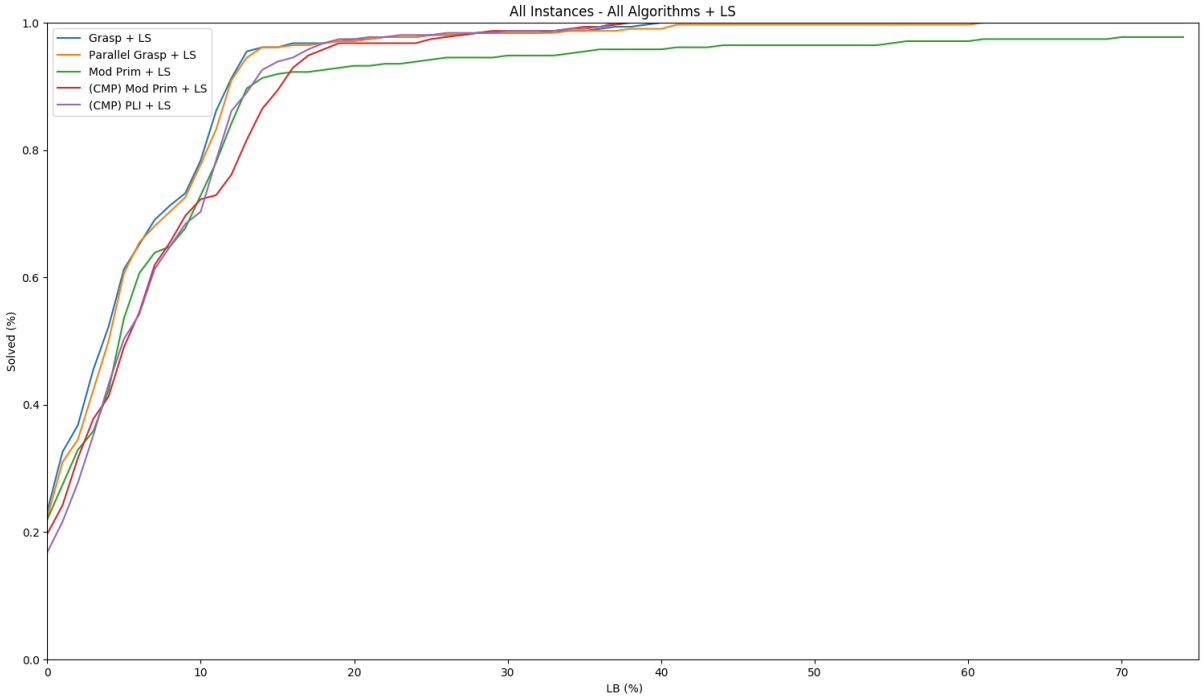


Figure 4.2: Algorithms + LS for the complete instance set

Notice how significant was the improvement after the local search algorithm, specially for Mod Prim. Mod Prim was solving less than 30% of the instances below the 10% gap mark, and after the local search, it jumps to over 70% of the instances solved below 10% gap to the lower bound.

We can summarize our heuristics' results in a table:

	avg	stdev	Q1	median	Q3	max
GRASP	**6.418**	**6.866**	**1.501**	**4.812**	**10.588**	**41.777**
Parallel GRASP	6.933	7.485	1.705	5.580	11.089	63.998
(CMP) PLI	17.742	16.202	7.889	13.219	22.653	111.460
(CMP) Mod Prim	14.525	11.593	4.299	11.992	22.199	42.703
Mod Prim	50.771	140.386	9.102	28.487	44.296	1946.668
GRASP + LS	**6.149**	6.446	**1.469**	**4.596**	**10.181**	40.544
Parallel GRASP + LS	6.424	7.059	1.561	5.008	10.409	61.384
(CMP) PLI + LS	7.369	**6.353**	2.651	5.986	11.542	**37.898**
(CMP) Mod Prim + LS	7.646	6.981	2.136	6.134	12.456	38.167
Mod Prim + LS	12.882	40.337	1.801	5.668	11.347	393.450

Table 4.1: Table of results, (%) to lower bound

(1) **avg** is the average, (2) **stdev** is the standard deviation, (3) **Q1** is the first quartile, (4) is the **median**, (5), **Q3** is the third quartile, and **max** is the maximum value found.

We can see that GRASP had a better performance among the algorithms without local search, although, is worth mentioning that GRASP itself already has a local search procedure. If we consider only algorithms that do not have any local search procedure, we have our heuristics based on the CTPD framework with Mod Prim and PLI, presenting a significantly better solution quality over Mod Prim.

Looking at Figure 4.2, after the local search, we have a considerable improvement in every algorithm besides GRASP, (CMP) Mod Prim jumps from 22.199% over the lower bound on its third quartile, to 12.456%. The same for (CMP) PLI, from 22.653% to 11.542%.

Comparing by the average result, we have the algorithm GRASP + LS, with the best results, and also as the one with the best solution for most of the instances. Comparing by the farthest solution from the lower bound, (CMP) PLI + LS presented at most 37.898% gap, compared to 40.544% of GRASP+LS.

We can also take into account the time required to run each heuristic:

	avg	stdev	Q1	median	Q3	max
GRASP + LS	579.9	53.0	531.5	560.6	635.3	635.9
Parallel GRASP + LS	605.4	38.8	578.6	635.1	635.4	635.7
(CMP) PLI + LS	493.4	183.4	340.8	635.0	635.5	635.9
(CMP) Mod Prim + LS	116.4	57.9	75.4	98.3	146.6	333.5
Mod Prim + LS	33.7	7.8	35.2	36.1	36.1	73.4

Table 4.2: Table of results, execution time in seconds

We see that Mod Prim after using the local search is a fast heuristic that has a competitive solution quality using only, on average, 33.7 seconds to solve each instance. It is also worth mentioning (CMP) Mod Prim + LS, that in our experiments solves 5 times the problem, each time with a different value of k, and still runs, on average, below the 2 minutes mark.

(CMP) PLI + LS also provides an interesting result, once infeasible for instances of this size, our heuristic framework enables a PLI algorithm to solve this set of very large scale instances. Solving at each time, instances of the CTPD of sizes of 50 and 100 vertices, we were capable of building a valid solution tree for the CTP, on average, in around 8 minutes.

The following sections present details about some results found on this research.

4.2.1 Heuristics Results by Graph Classes

Spiral

Starting with the graph set where every algorithm presented, on average, its higher gap to the lower bound, we present Figure 4.4 with the lower bound gap followed by a Table 4.3 summarizing its results.

As presented in Figure 4.3, this instance set is composed of instances where the vertices in the graph present a spiral form. Based on its greedy nature, it biases our heuristics to "follow" the spiral, which may lead to bad solutions.

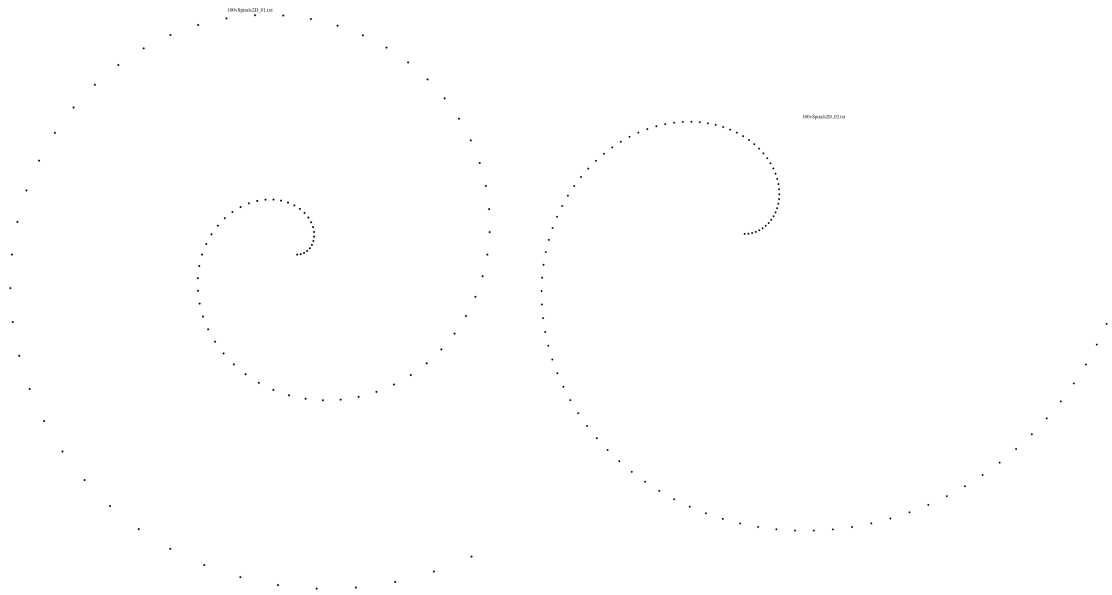


Figure 4.3: Spiral Instance Examples - 100 vertices

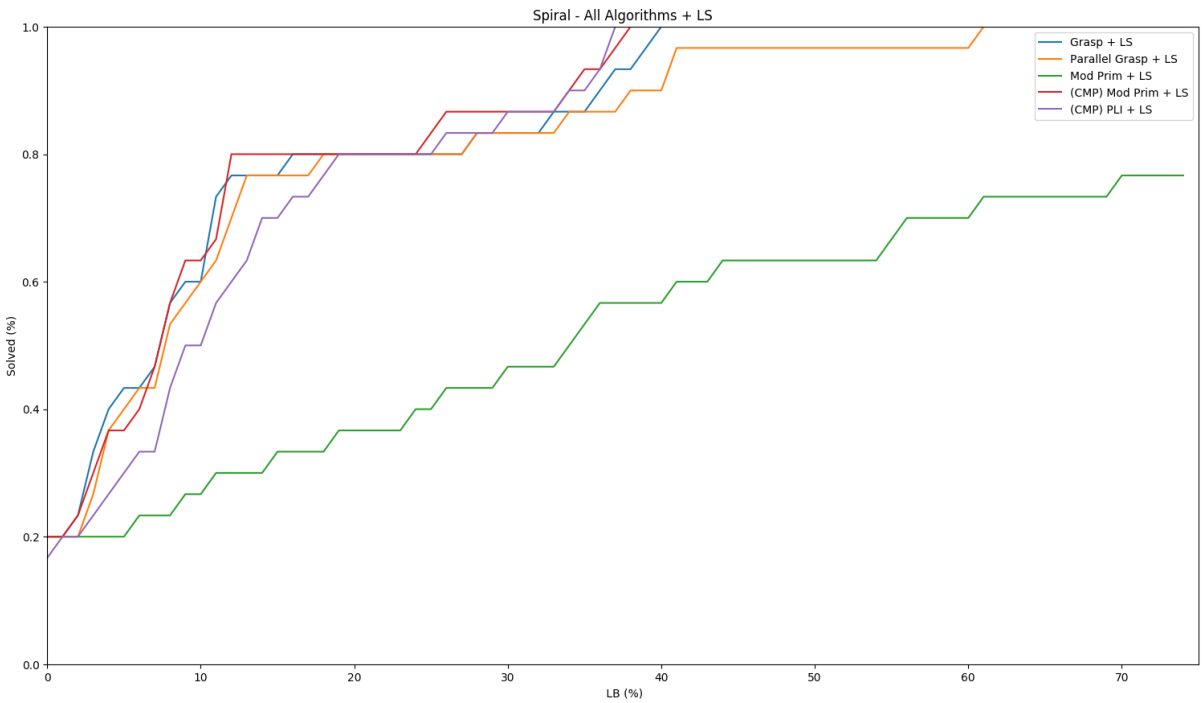


Figure 4.4: Spiral Instance Set - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	12.965	14.270	3.623	8.578	12.745	41.777
Parallel GRASP	14.416	16.262	3.931	9.088	14.419	63.998
(CMP) PLI	24.140	15.411	11.977	20.339	31.849	57.734
(CMP) Mod Prim	13.771	13.304	4.571	9.610	14.905	42.703
Mod Prim	278.662	382.689	50.266	225.644	289.677	1946.668
GRASP + LS	12.147	13.003	3.575	8.414	12.471	40.544
Parallel GRASP + LS	13.542	15.426	3.729	8.745	13.514	61.384
(CMP) PLI + LS	13.402	11.868	4.424	10.444	18.059	37.898
(CMP) Mod Prim + LS	11.554	11.833	3.750	8.552	12.156	38.167
Mod Prim + LS	76.553	111.598	9.917	34.795	68.085	393.450

Table 4.3: Spiral Instance Set - Table of results, (%) to lower bound

Notice that Mod Prim + LS presented in the main instance set an average of 12.882% gap to the lower bound. But viewing only Spiral instances, its average gap is over 76%.

Note how (CMP) Mod Prim in this instance set have the best solution among all heuristics. We speculate that, due to the fact of "compressing" the graph, it allows the greedy algorithm to have more information about the instance, assisting the heuristic to avoid bad greedy choices.

Vasko

Vasko set present a graph based on 3D-points of a vascular image exam. We present an image with the lower bound gap followed by a table summarizing its results.

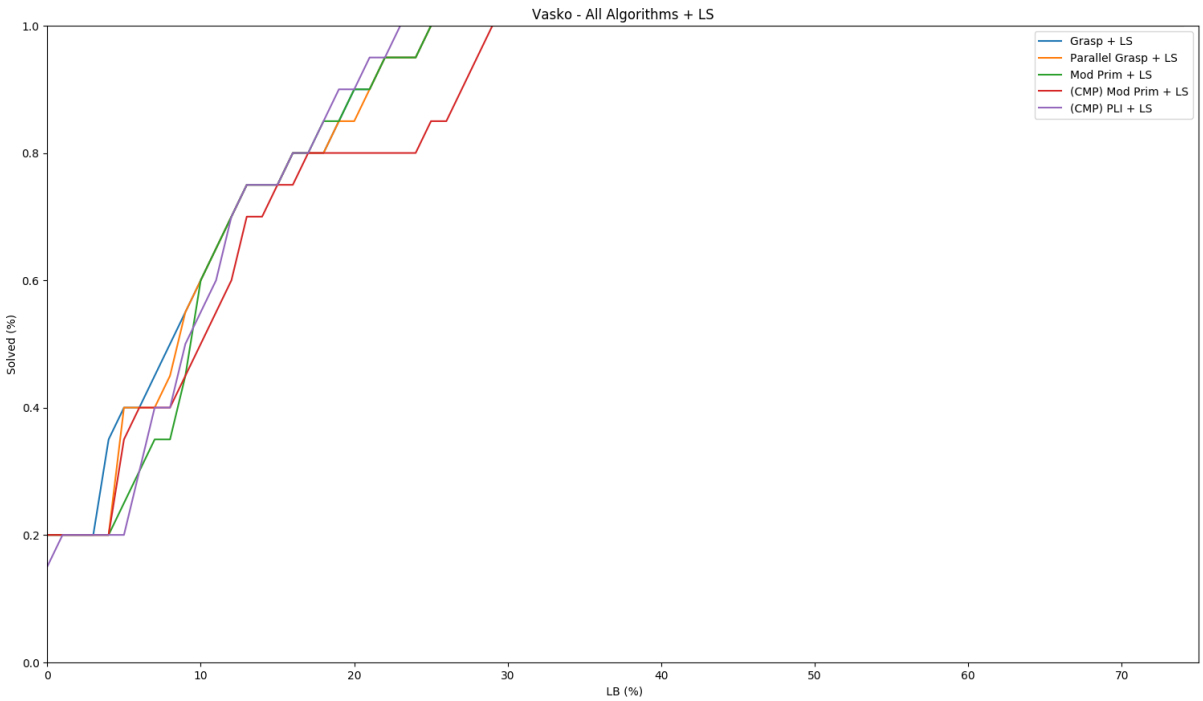


Figure 4.5: Vasko Instance Set - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	10.257	8.158	4.536	9.267	14.388	25.520
Parallel GRASP	10.800	8.238	5.432	10.346	14.669	26.148
(CMP) PLI	20.170	5.655	15.811	20.586	25.408	28.360
(CMP) Mod Prim	16.656	10.081	7.671	14.756	21.520	33.863
Mod Prim	43.857	29.184	23.862	51.879	63.810	92.728
GRASP + LS	9.889	7.838	4.330	9.055	13.946	25.516
Parallel GRASP + LS	10.131	7.759	5.056	9.515	14.110	25.301
(CMP) PLI + LS	10.464	6.946	6.640	9.994	14.226	23.280
(CMP) Mod Prim + LS	11.991	9.610	5.263	10.668	15.957	29.256
Mod Prim + LS	10.591	7.314	6.018	10.596	14.223	25.072

Table 4.4: Vasko Instance Set - Table of results, (%) to lower bound

Notice that for this set, after running the local search, all heuristics presented results, on average, around the 10% gap to the lower bound.

Also worth mentioning that even for a 3D-based instance set, our (CMP) heuristics did not lose performance, still competitive among the other presented heuristics.

Maps

Maps set represents maps of countries, where each vertex represents a city. We present a figure with the lower bound gap followed by a table summarizing its results.

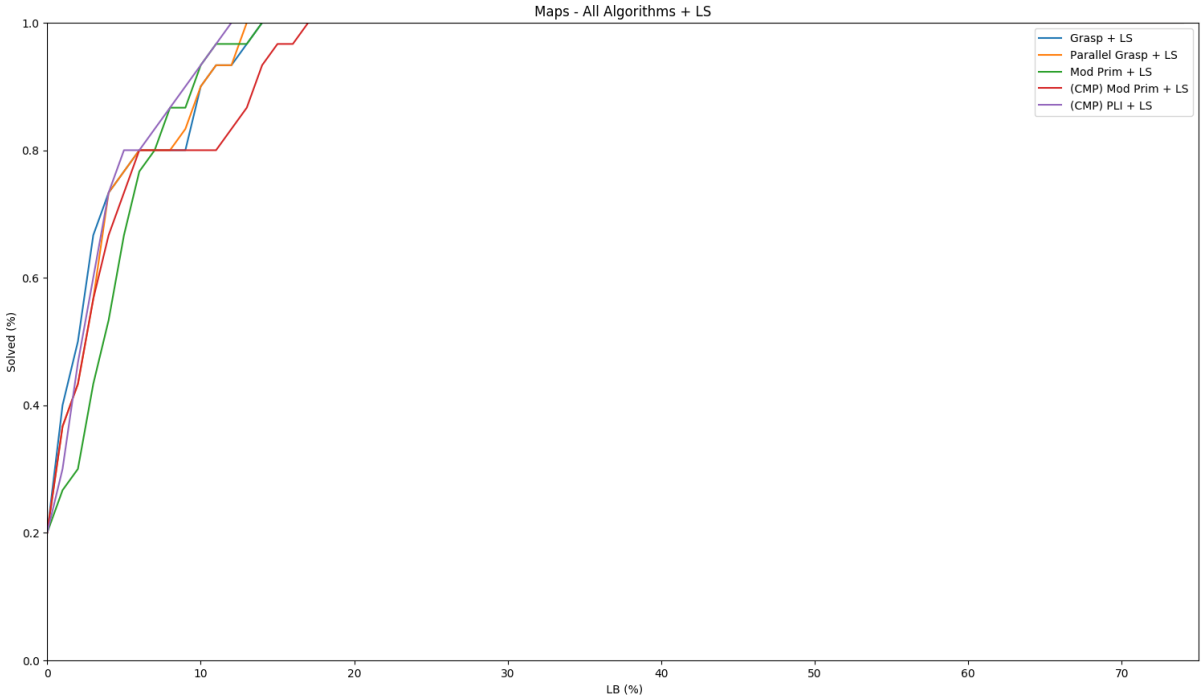


Figure 4.6: Maps Instance Set - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	4.474	4.633	1.336	3.319	5.390	15.867
Parallel GRASP	4.833	4.546	1.705	4.027	5.947	15.734
(CMP) PLI	7.192	5.194	3.587	5.140	9.007	21.921
(CMP) Mod Prim	8.294	9.573	1.835	4.512	9.018	32.467
Mod Prim	32.161	25.017	10.575	31.626	49.915	86.769
GRASP + LS	4.155	4.185	1.248	3.165	5.232	14.059
Parallel GRASP + LS	4.384	4.145	1.561	3.746	5.520	13.881
(CMP) PLI + LS	4.048	3.486	1.780	3.074	5.293	12.155
(CMP) Mod Prim + LS	5.056	5.258	1.266	3.364	6.074	17.647
Mod Prim + LS	4.830	3.865	1.875	4.679	6.634	14.949

Table 4.5: Maps Instance Set - Table of results, (%) to lower bound

Our heuristics found on average, a gap below 5% to the lower bound, highlighting how these heuristics are suitable for real map-related applications.

Another interesting aspect to notice is how significant was the improvement of the local search procedure for Mod Prim. The average result goes from 32.161% to 4.830%, lowering over 8 times the gap to the lower bound.

Other Graph classes

As the other graph classes have not presented a high individual variability in the results, we are presenting them combined.

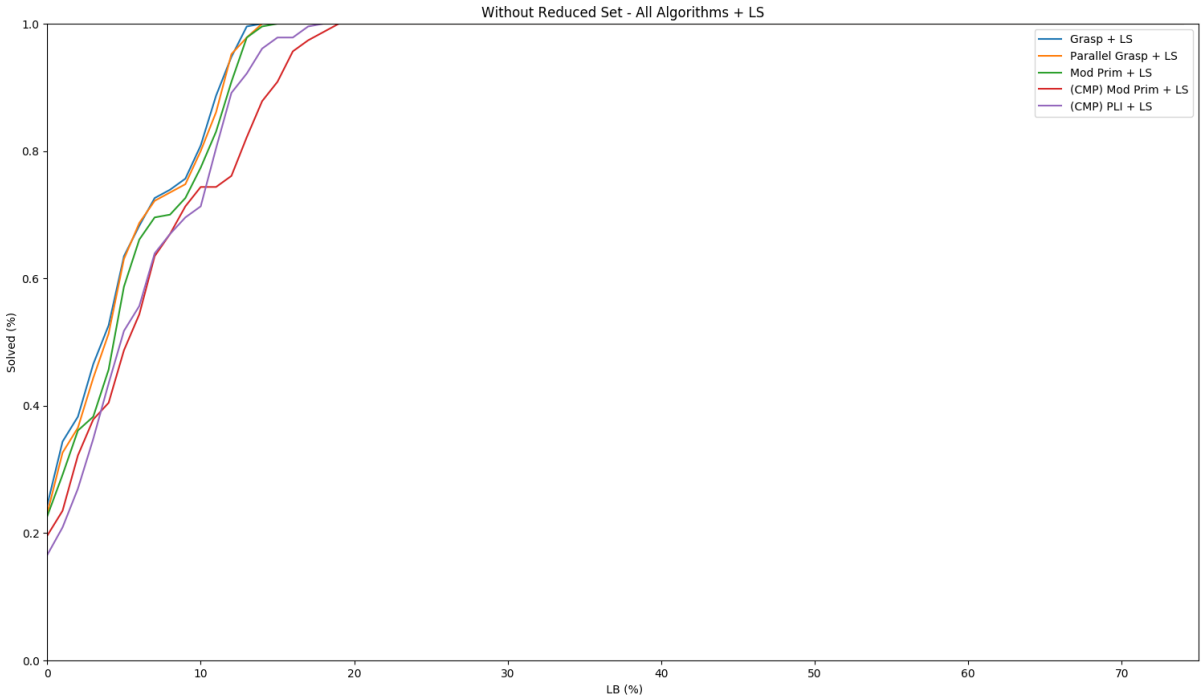


Figure 4.7: All Instances besides Reduced Set - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	5.483	4.615	1.210	4.826	10.038	14.325
Parallel GRASP	5.894	4.854	1.378	5.548	10.613	15.893
(CMP) PLI	18.072	17.248	8.077	12.839	23.076	111.460
(CMP) Mod Prim	15.251	11.525	4.303	13.605	24.266	42.329
Mod Prim	24.075	18.111	8.842	26.799	36.537	105.731
GRASP + LS	5.302	4.490	1.147	4.596	9.897	14.253
Parallel GRASP + LS	5.439	4.535	1.277	4.912	9.973	14.255
(CMP) PLI + LS	6.746	4.863	2.677	5.920	11.484	18.176
(CMP) Mod Prim + LS	7.096	5.660	2.242	6.214	12.331	19.455
Mod Prim + LS	5.827	4.684	1.752	5.255	10.672	15.130

Table 4.6: All Instances besides Reduced Set - Table of results, (%) to lower bound

For all other instances besides the ones on the reduced set, we achieved an average result between 5% and 7%, with an at most 19.455% gap to the lower bound. We can highlight that before the local search, we had instances with a gap over 111%, and after the local search, all gaps of this 230 instances are below 20%.

4.2.2 Heuristics Results by Cable and Trench Cost

We present the results of our main instance set segmented by cable and trench cost. We briefly comment on the results at the end of this subsection. For each value of cable and trench, we present an image with the lower bound gap, followed by a table summarizing its results.

$\gamma = 1.0, \tau = 0.01$

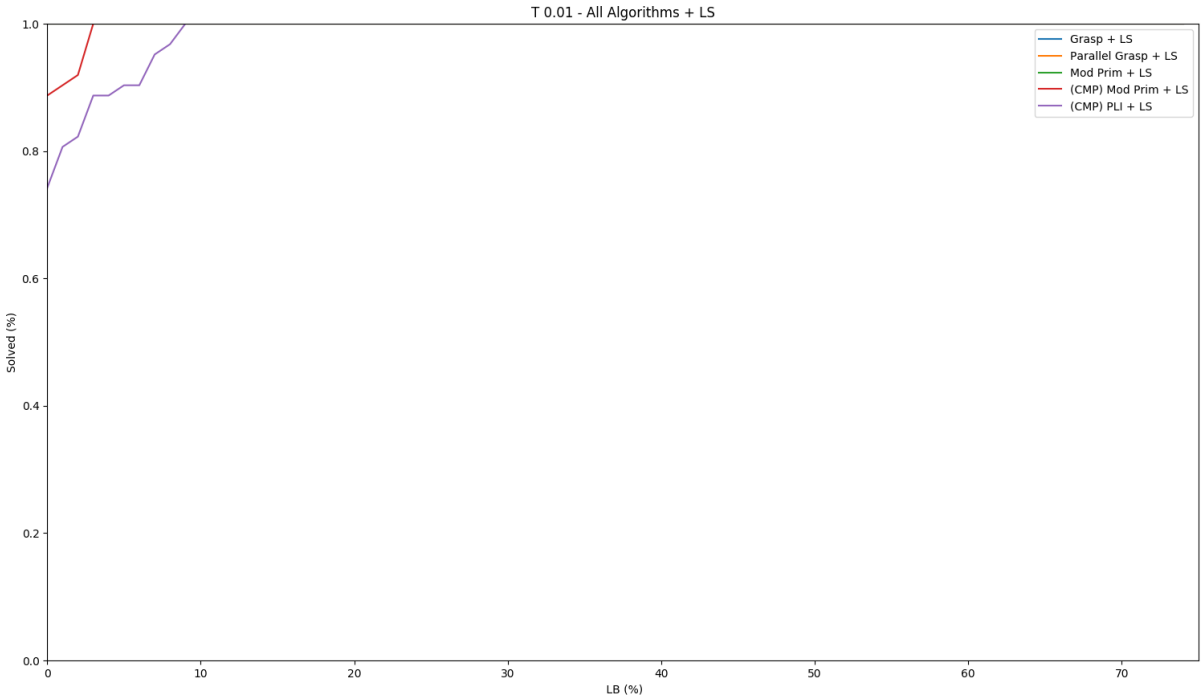


Figure 4.8: All Instances - $\gamma = 1.0, \tau = 0.01$ - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	0.169	0.147	0.050	0.075	0.338	0.499
Parallel GRASP	0.174	0.148	0.057	0.076	0.339	0.501
(CMP) PLI	24.997	29.636	6.569	11.407	38.481	111.460
(CMP) Mod Prim	8.265	11.311	1.201	2.356	13.733	40.479
Mod Prim	0.297	0.177	0.138	0.236	0.474	0.573
GRASP + LS	0.167	0.146	0.048	0.073	0.338	0.499
Parallel GRASP + LS	0.170	0.146	0.052	0.075	0.338	0.499
(CMP) PLI + LS	1.542	2.531	0.189	0.478	0.993	9.775
(CMP) Mod Prim + LS	0.583	0.950	0.114	0.226	0.511	3.794
Mod Prim + LS	0.179	0.148	0.057	0.084	0.332	0.498

Table 4.7: All Instances - $\gamma = 1.0, \tau = 0.01$ - Table of results, (%) to lower bound

$\gamma = 1.0, \tau = 1.0$

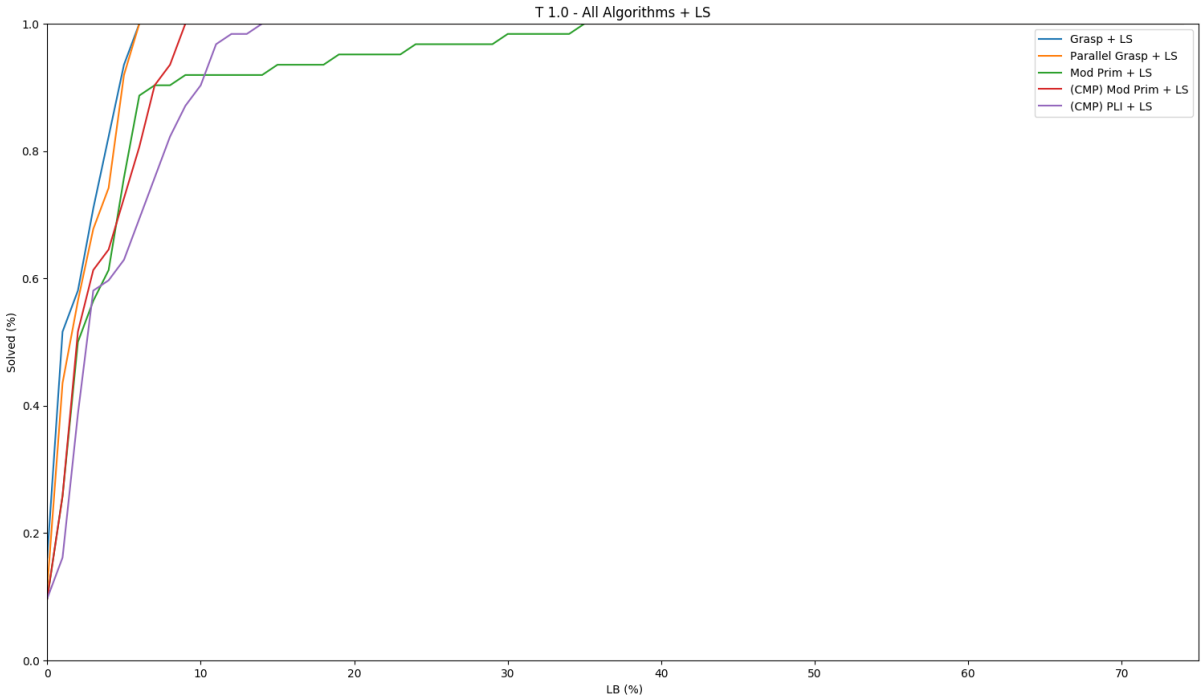


Figure 4.9: All Instances - $\gamma = 1.0, \tau = 1.0$ - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	2.886	1.875	1.585	2.008	4.536	6.438
Parallel GRASP	3.369	2.128	1.814	2.491	5.364	7.513
(CMP) PLI	11.251	8.163	5.895	8.575	15.102	48.571
(CMP) Mod Prim	6.061	4.143	3.119	4.568	9.796	14.926
Mod Prim	15.803	11.619	9.211	13.692	15.020	52.349
GRASP + LS	2.769	1.807	1.513	1.908	4.330	6.169
Parallel GRASP + LS	3.005	1.907	1.625	2.105	5.007	6.534
(CMP) PLI + LS	5.053	3.628	2.417	3.440	7.641	14.364
(CMP) Mod Prim + LS	4.064	2.730	2.014	2.946	6.524	9.905
Mod Prim + LS	5.230	6.620	1.978	2.773	5.944	35.221

Table 4.8: All Instances - $\gamma = 1.0, \tau = 1.0$ - Table of results, (%) to lower bound

$\gamma = 1.0, \tau = 5.0$

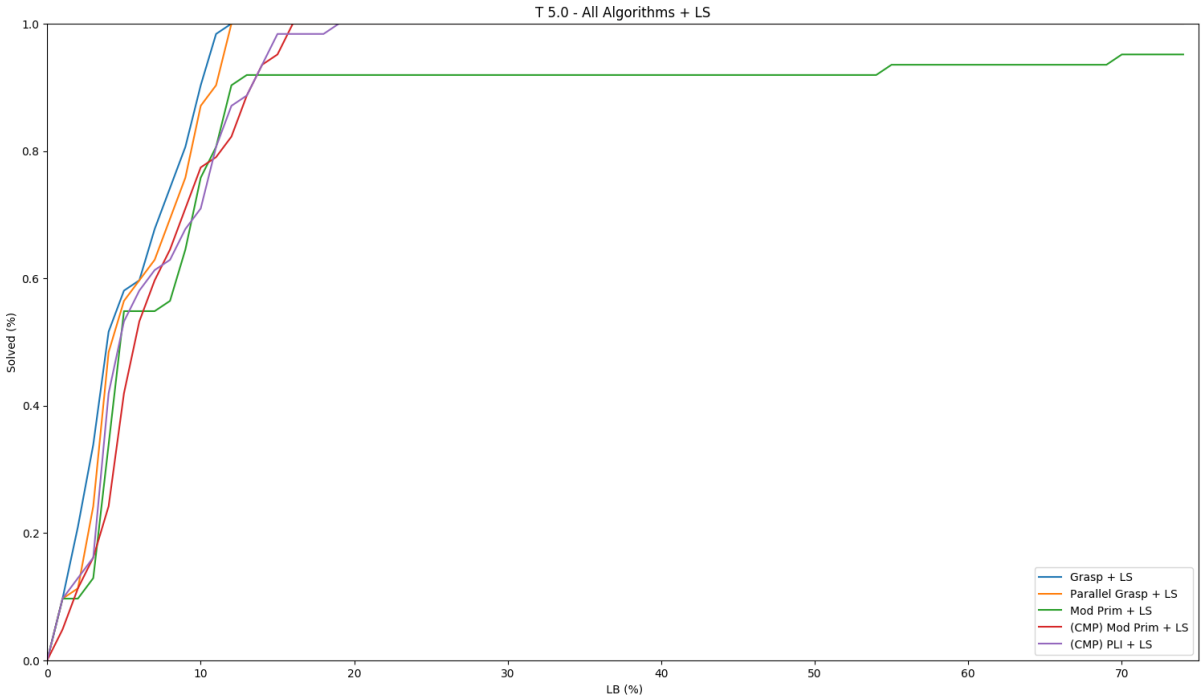


Figure 4.10: All Instances - $\gamma = 1.0, \tau = 5.0$ - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	6.367	3.429	3.895	5.082	9.544	12.431
Parallel GRASP	7.206	3.827	4.539	5.887	10.625	14.369
(CMP) PLI	13.201	7.583	7.683	11.760	17.779	48.164
(CMP) Mod Prim	12.682	7.406	8.425	10.168	15.730	27.821
Mod Prim	54.836	65.094	27.210	34.594	42.752	290.084
GRASP + LS	6.121	3.315	3.729	4.839	9.245	12.039
Parallel GRASP + LS	6.487	3.458	4.053	5.055	9.852	12.990
(CMP) PLI + LS	7.506	4.467	4.300	5.675	11.233	19.453
(CMP) Mod Prim + LS	7.760	4.197	5.012	6.314	10.482	16.316
Mod Prim + LS	15.204	31.341	4.602	5.662	10.935	190.920

Table 4.9: All Instances - $\gamma = 1.0, \tau = 5.0$ - Table of results, (%) to lower bound

$\gamma = 1.0, \tau = 10.0$

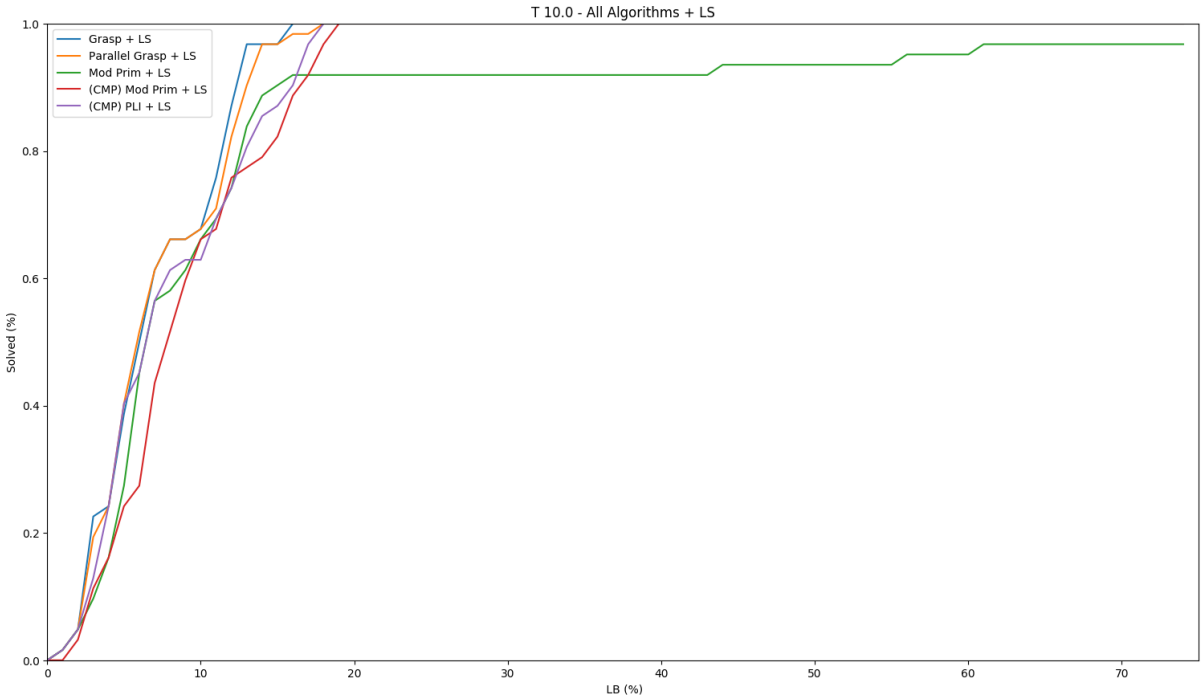


Figure 4.11: All Instances - $\gamma = 1.0, \tau = 10.0$ - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	8.194	4.053	5.560	7.131	12.245	17.174
Parallel GRASP	8.863	4.372	6.110	7.726	12.808	20.315
(CMP) PLI	15.480	8.611	9.353	12.810	22.085	49.432
(CMP) Mod Prim	17.682	9.391	11.677	16.590	20.836	37.407
Mod Prim	75.095	115.688	34.291	40.638	49.925	630.626
GRASP + LS	7.898	3.935	5.293	6.804	11.905	16.538
Parallel GRASP + LS	8.173	4.076	5.658	6.862	12.226	18.624
(CMP) PLI + LS	8.921	4.820	5.633	7.198	13.372	18.738
(CMP) Mod Prim + LS	9.902	4.795	6.537	8.470	12.666	19.280
Mod Prim + LS	19.139	51.132	5.989	7.146	12.918	364.468

Table 4.10: All Instances - $\gamma = 1.0, \tau = 10.0$ - Table of results, (%) to lower bound

$\gamma = 1.0, \tau = 100.0$

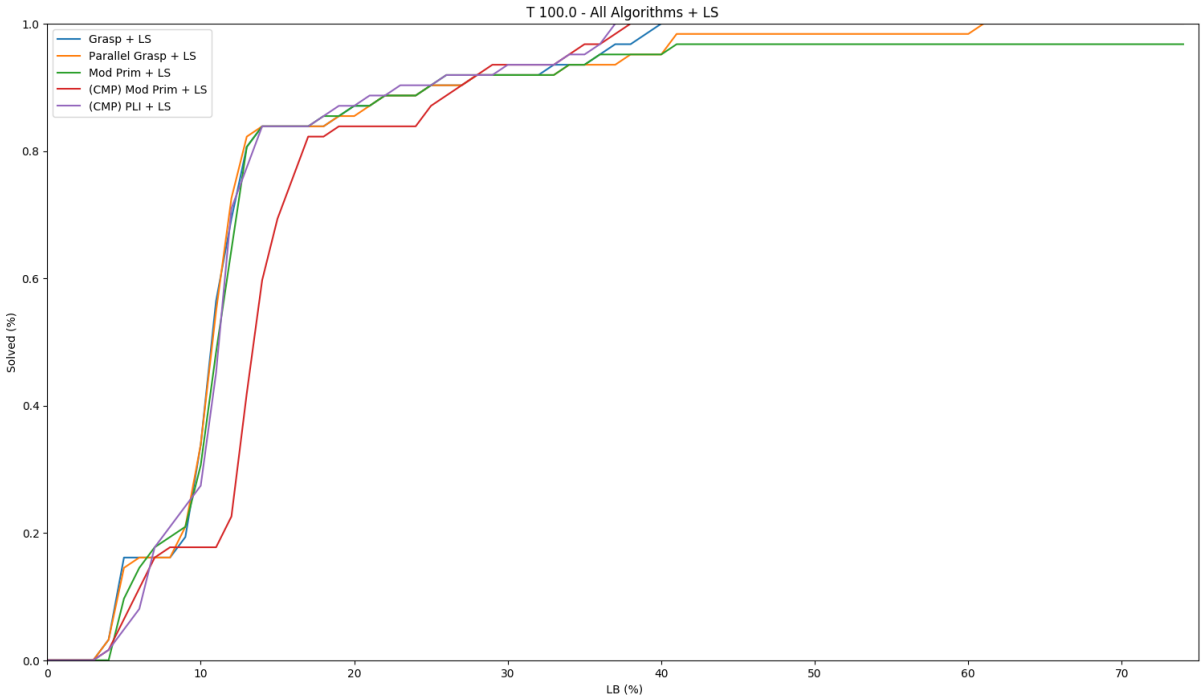


Figure 4.12: All Instances - $\gamma = 1.0, \tau = 100.0$ - Algorithms + LS

	avg	stdev	Q1	median	Q3	max
GRASP	14.470	9.251	10.800	12.082	13.854	41.777
Parallel GRASP	15.050	10.740	11.017	12.639	13.954	63.998
(CMP) PLI	23.780	9.677	15.038	25.193	28.634	57.734
(CMP) Mod Prim	27.938	9.142	19.110	29.264	33.900	42.703
Mod Prim	107.825	272.614	33.276	44.468	59.791	1946.668
GRASP + LS	13.789	8.385	10.426	11.730	13.229	40.544
Parallel GRASP + LS	14.284	10.130	10.468	11.792	13.051	61.384
(CMP) PLI + LS	13.821	7.549	10.401	12.168	13.135	37.898
(CMP) Mod Prim + LS	15.922	7.705	13.165	14.512	16.568	38.167
Mod Prim + LS	24.659	64.743	10.617	12.078	13.324	393.450

Table 4.11: All Instances - $\gamma = 1.0, \tau = 100.0$ - Table of results, (%) to lower bound

Looking at the above images and tables, we can notice how different values of cables and tranches have a significant impact on the results.

At first, with $\tau = 0.01$, most heuristics solve the instances near to 0% gap to the lower bound. As the τ value increases, the problem becomes more difficult, rising the average gap found by the heuristics. Its worth mentioning that the problem will not become indefinitely hard as the value of τ increases. For larger values of τ, the problem will be closer to the MST problem and therefore will, again, become easy to solve. To show it, we run in the RIS, the Mod Prim algorithm, and we present the average gap (%) to the lower bound in the Table 4.12.

τ	0.001	0.01	1	5	10	100	1,000	10,000	100,000	1,000,000	10,000,000
avg	0.04	0.32	27.64	112.84	167.04	308.08	136.43	21.05	2.24	0.22	0.02
stdev	0.03	0.20	15.36	98.11	177.76	493.53	199.46	28.55	3.03	0.30	0.03

Table 4.12: (%) to lower bound for Mod Prim in the RIS for different cable/trench ratios

4.2.3 Local Search

We have already presented some results related to the local search algorithm. The goal of this Section is to show how the local search improves the solution quality over the time.

First, let us present a figure of the local search when applied to the main instance set. We are tracking the average solution value of each algorithm over 35 seconds where the local search algorithm is running.

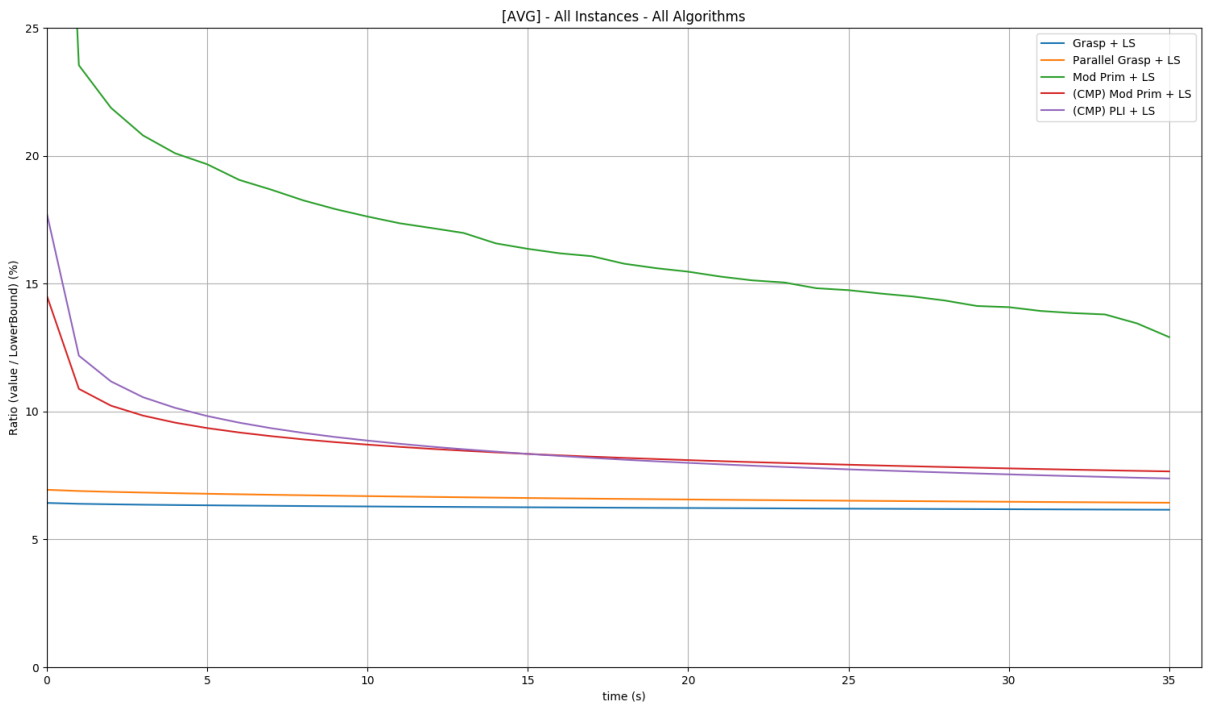


Figure 4.13: All Instances - Local Search improvement over time

Following, we present the same graph for the instances of the reduced instance set, respectively, Spiral, Vasko, and Maps.

As the gap of Mod Prim algorithm is too high for this instance set, we present the gap from $[0, 200]$.

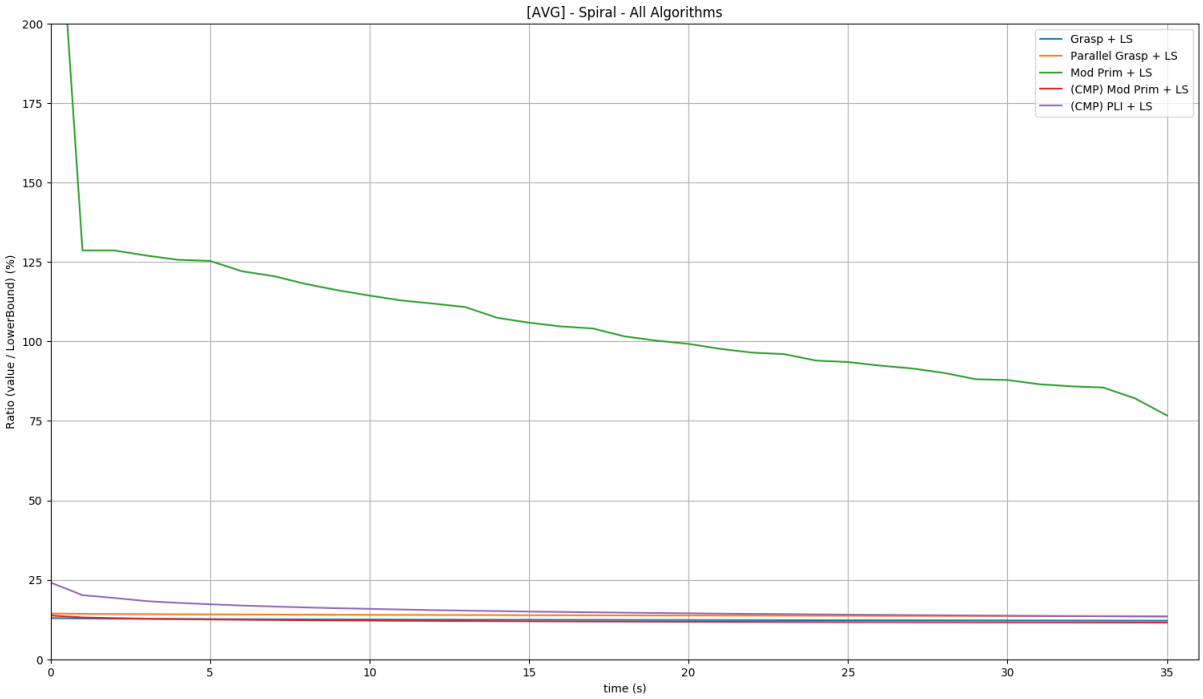


Figure 4.14: Spiral Instance Set - Local Search improvement over time [0, 200]

The average gap for the Spiral set in the range of $[0, 25]$.

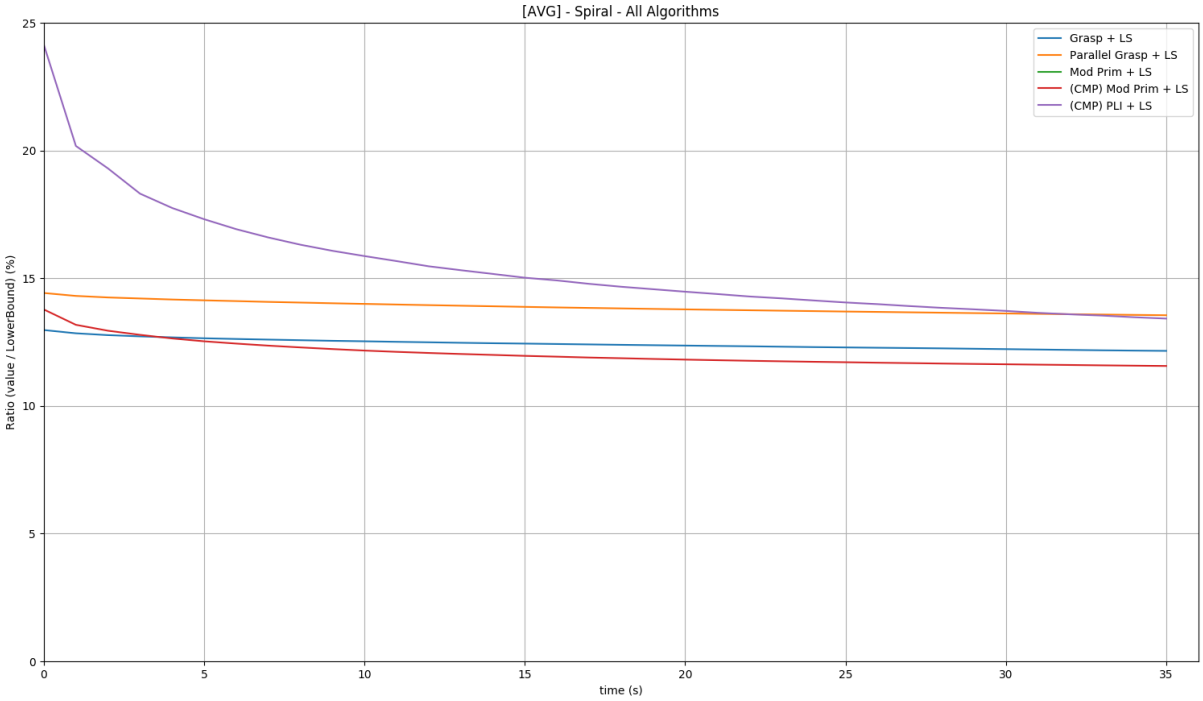


Figure 4.15: Spiral Instance Set - Local Search improvement over time [0, 25]

The average gap for the Vasko set in the range of $[0, 25]$.

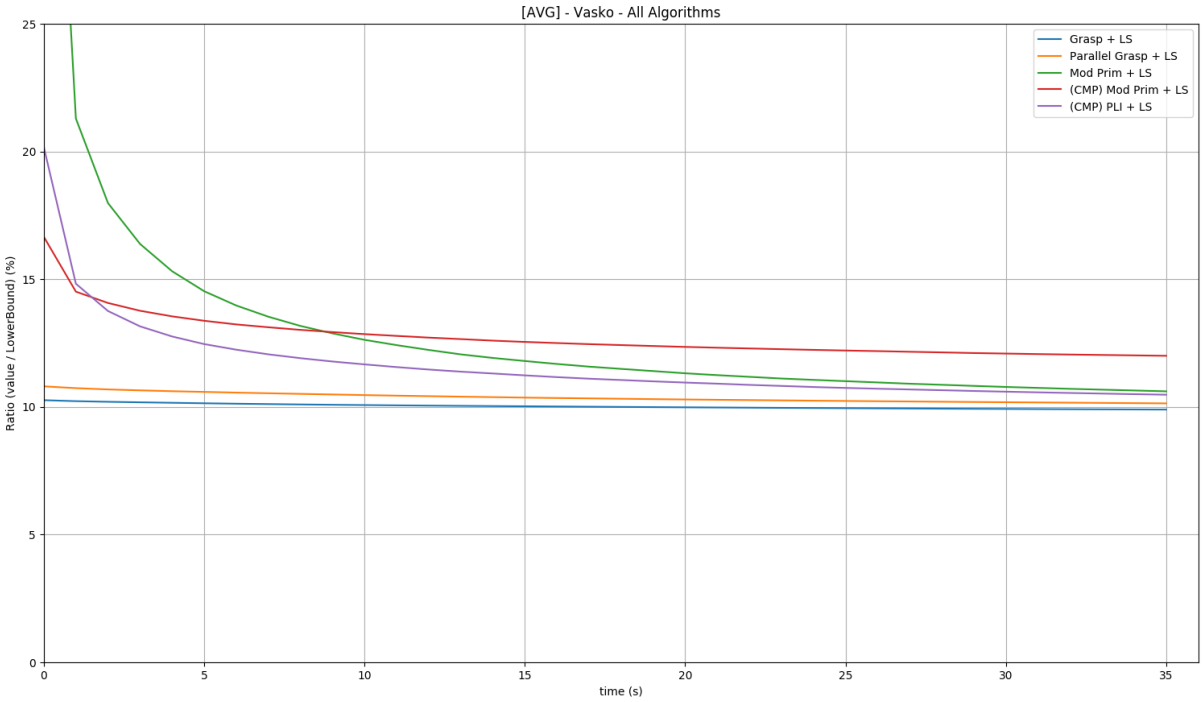


Figure 4.16: Vasko Instance Set - Local Search improvement over time [0, 25]

The average gap for the Maps set in the range of $[0, 25]$.

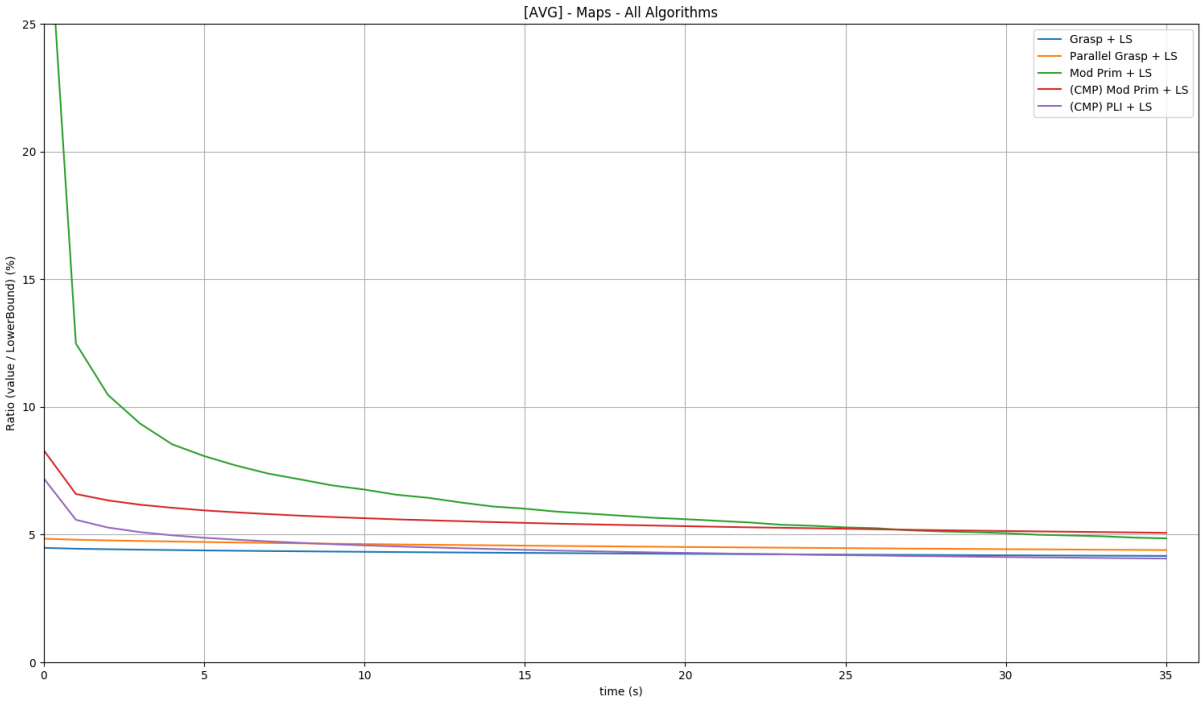


Figure 4.17: Maps Instance Set - Local Search improvement over time [0, 25]

For all instance sets, the most significant changes occur before the 10 seconds of running time. This show how fast this heuristic is, even when applied to large-scale instances.

4.2.4 Edge Reduction Rules

We study the number of edges in our reduced instance set when applying each of these reduction rules. Let us start by presenting a table summarizing the number of edges of each instance.

	avg	Q1	median	Q3	max
R1 + R2 + R3	5673971	477282	2235660	6813934	51152078
R1 + R3	5673971	477282	2235660	6813934	51152078
R1 + R2	29446974	5480550	15512797	31763918	180115074
R1	39725632	13305512	22955939	43255897	180178346
none	417919886	219038760	342145091	624150380	1136195556

Table 4.13: Number of edges with edge reduction rules for the reduced instance set

We can also analyse this data using the average degree of each vertex.

	avg	stdev	Q1	median	Q3	max
R1 + R2 + R3	266.051	381.987	27.135	118.157	392.418	2245.975
R1 + R3	266.051	381.987	27.135	118.157	392.418	2245.975
R1 + R2	1315.113	1357.569	351.937	836.719	1566.077	5343.392
R1	1775.586	1346.844	885.822	1446.066	2206.442	5345.270
none	19210.000	7032.867	14795.250	18430.000	24982.500	33707.000

Table 4.14: Vertex degree with edge reduction rules for the reduced instance set

Notice that after applying all reduction rules, we are only considering, on average, **1.35%** edges of the complete graph.

Here we can also comment that for our instance set (and the values of β chosen for reduction rule 3), we had no difference between applying reductions R1 + R2 + R3 or just R1 + R3.

An important aspect to be evaluated is how this reduction rules impact our heuristics solutions. Keeping the 10 minutes of the time limit, we executed our reduced instance set with each of those combinations of edge reduction rules.

Reduction rule 1 ($\lambda = 10\%$)

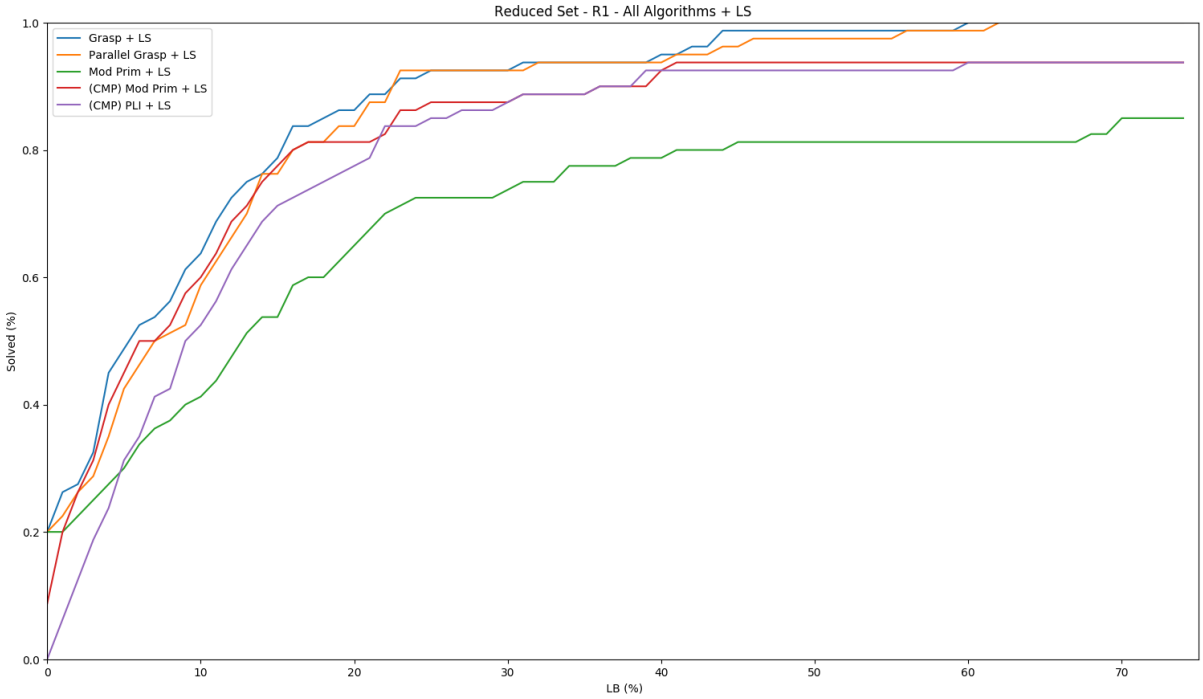


Figure 4.18: Reduced Instance Set - R1

	avg	stdev	Q1	median	Q3	max
GRASP	10.872	12.285	2.002	6.853	14.261	61.850
Parallel GRASP	13.175	13.490	3.430	9.703	17.705	65.163
(CMP) PLI	43.910	106.359	9.421	14.418	24.184	474.184
(CMP) Mod Prim	39.486	107.432	3.520	9.096	20.191	474.184
Mod Prim	127.546	260.944	16.341	50.395	83.529	1946.668
GRASP + LS	10.395	11.969	1.801	6.383	13.887	60.882
Parallel GRASP + LS	11.564	12.623	2.772	8.325	14.324	62.645
(CMP) PLI + LS	39.598	107.285	5.163	10.221	18.920	474.184
(CMP) Mod Prim + LS	37.337	107.802	2.988	7.630	14.543	474.184
Mod Prim + LS	46.482	90.173	4.434	13.882	32.025	463.638

Table 4.15: Reduced Instance Set - R1 - Table of results, (%) to lower bound

Reduction rule 1 ($\lambda = 10\%$) **and Reduction Rule 2**

Another experiment, applying the Reduction rule 1, with an $\lambda = 10\%$ and the Reduction Rule 2.

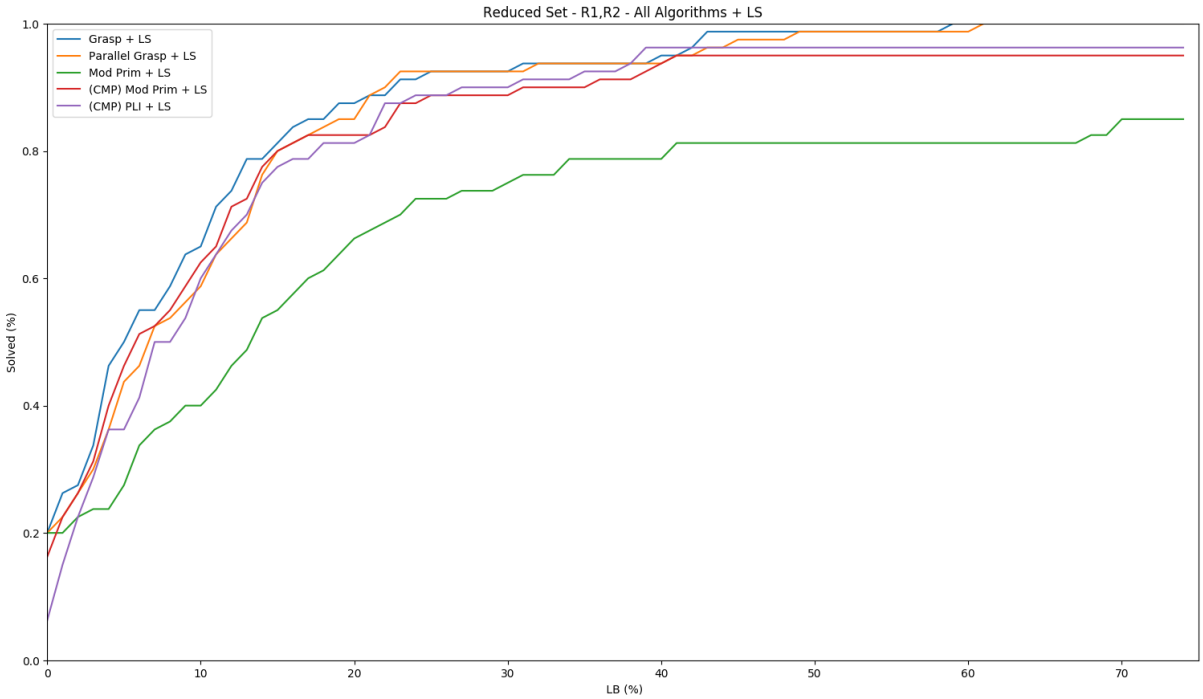


Figure 4.19: Reduced Instance Set - R1 + R2

	avg	stdev	Q1	median	Q3	max
GRASP	10.420	11.976	2.064	6.368	13.676	60.343
Parallel GRASP	12.606	12.936	3.275	9.559	17.230	63.633
(CMP) PLI	32.561	80.779	9.405	14.180	21.892	467.700
(CMP) Mod Prim	33.724	95.501	4.235	8.976	18.929	472.886
Mod Prim	127.546	260.944	16.341	50.395	83.529	1946.668
GRASP + LS	10.056	11.747	1.841	5.998	13.275	59.590
Parallel GRASP + LS	11.147	12.164	2.800	7.457	14.641	61.576
(CMP) PLI + LS	26.484	81.612	3.564	8.492	14.741	467.700
(CMP) Mod Prim + LS	31.290	95.876	2.899	6.763	14.173	472.886
Mod Prim + LS	43.952	85.883	5.077	14.408	30.820	463.638

Table 4.16: Reduced Instance Set - R1 + R2 - Table of results, (%) to lower bound

Reduction rule 1 ($\lambda = 10\%$) and Reduction Rule 3 ($\beta = 1.0$)

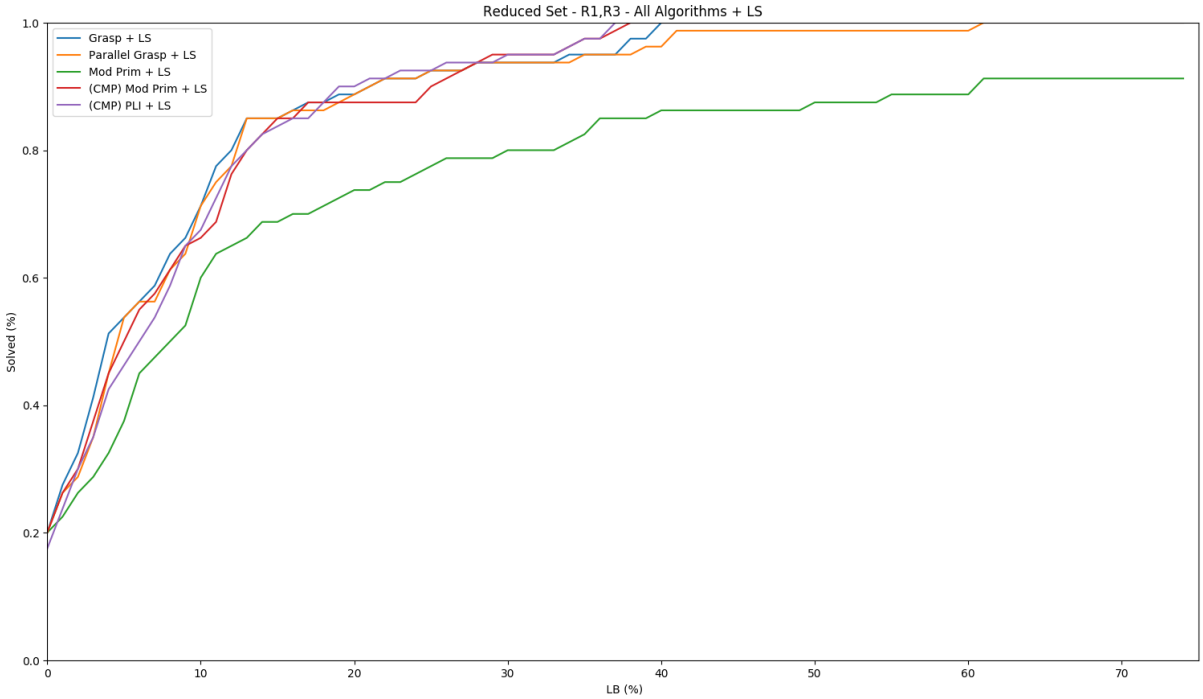


Figure 4.20: Reduced Instance Set - R1 + R3

	avg	stdev	Q1	median	Q3	max
GRASP	9.237	10.807	1.574	4.866	12.196	44.880
Parallel GRASP	10.172	11.902	2.020	6.192	12.780	65.047
(CMP) PLI	16.792	12.774	7.114	13.617	21.905	57.734
(CMP) Mod Prim	12.438	11.608	3.787	7.733	18.405	42.703
Mod Prim	127.523	260.950	16.341	50.287	83.529	1946.668
GRASP + LS	8.713	9.974	1.498	4.648	11.457	40.758
Parallel GRASP + LS	9.399	11.271	1.744	5.367	11.874	61.850
(CMP) PLI + LS	9.158	9.214	2.112	6.996	12.289	37.900
(CMP) Mod Prim + LS	9.222	9.709	1.774	5.820	12.542	38.157
Mod Prim + LS	32.886	75.683	2.888	8.959	22.793	393.450

Table 4.17: Reduced Instance Set - R1 + R3 - Table of results, (%) to lower bound

Notice that even considering only an average of **1.35%** of the edges, we had better results than without the reduction rules. One reason is the considerably smaller number of edges and the resulting speedup on the heuristics.

We can also consider the impact of the number of edges, on how fast the local search improves, on average, each solution.

In the images below we have the average result of lower bound over time, for each set of reduction rules.

Reduction rule 1 ($\lambda = 10\%$)

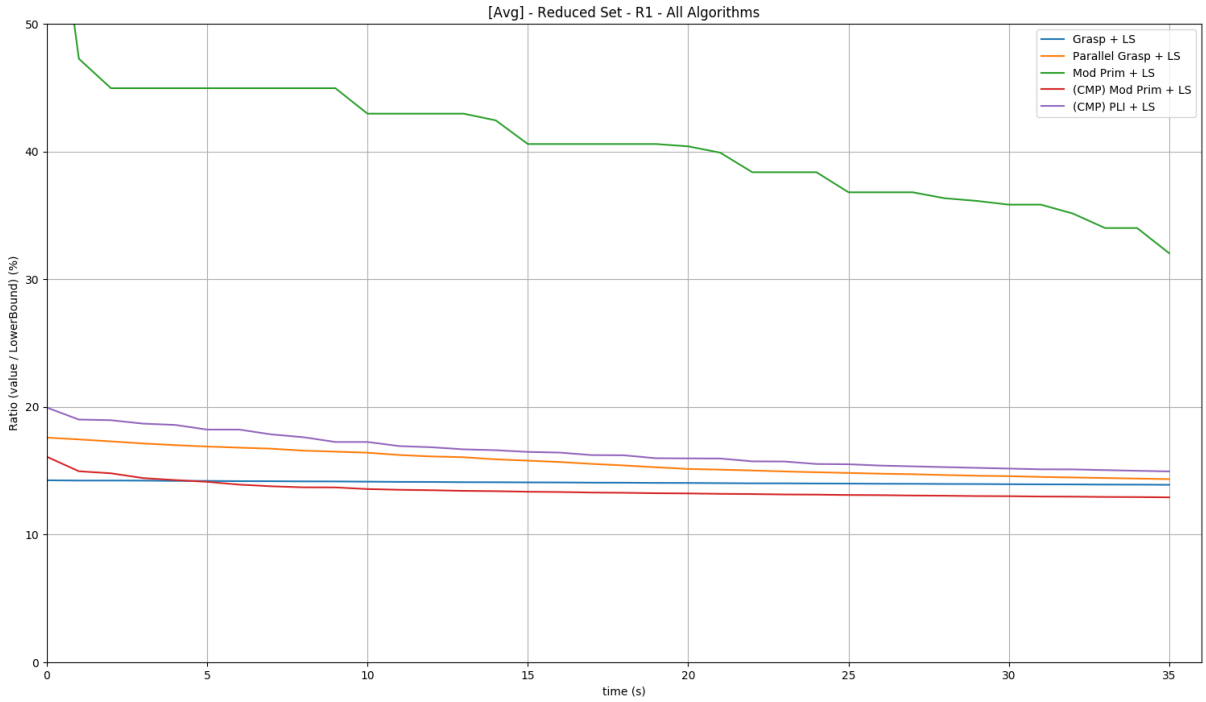


Figure 4.21: Reduced Instance Set - R1 - Local Search improvement over time

Reduction rule 1 ($\lambda = 10\%$) and Reduction Rule 2

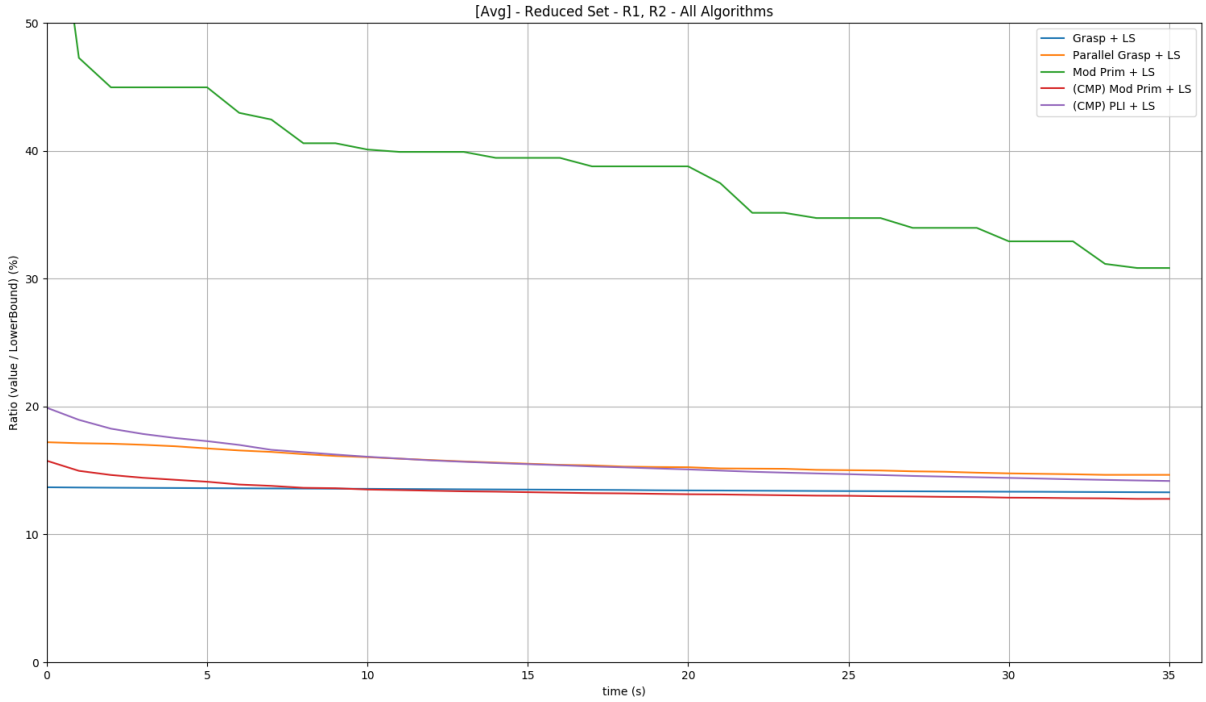


Figure 4.22: Reduced Instance Set - R1 + R2 - Local Search improvement over time

Reduction rule 1 ($\lambda = 10\%$) and Reduction Rule 3 ($\beta = 1.0$)

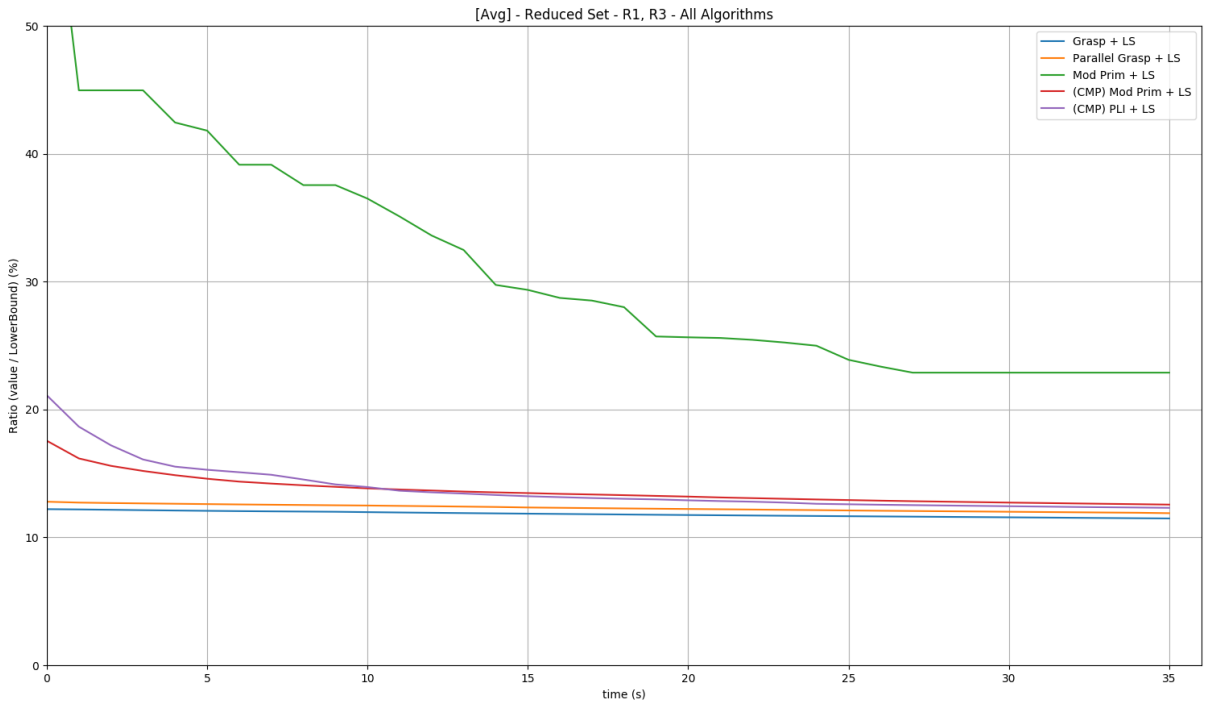


Figure 4.23: Reduced Instance Set - R1 + R3 - Local Search improvement over time

From these graphs, we can notice how the smaller number of edges allow the local search to find better results in a shorter period of time.

4.2.5 Lower Bound

In our research, we use the lower bound to compare our heuristic results. To present, at least a superficial understanding on the quality of our lower bounds, we generated an instance set following the same rule of the instances used in our main experiment, but limiting it to a size of 100 vertices (due to the fact that this is the limit which we are capable of finding the optimal value on a reasonable amount of time).

We calculate the optimal value running the multi-commodity model presented on Section 2.1.2, which we are calling Model 2. We are also calling the MILP Model found on Section 2.1.1 as Model 1.

This table shows the gap between the optimal value and each of these lower bound methods.

	avg	stdev	Q1	median	Q3	max
Model 1 - LP	10.158	12.581	0.743	7.029	12.438	72.456
Model 2 - LP	0.000	0.000	0.000	0.000	0.000	0.000
Lower Bound (MST + SSSP)	7.076	7.028	1.794	5.663	9.804	28.875
Lagrangian Relaxation	3.809	14.594	0.000	0.000	0.069	97.875

Table 4.18: Lower Bound to OPT - Instances of 100 vertices

The lower bound used on this research achieved an average of 7% distance to the optimal solution.

Notice that, there is almost no gap between Model 2 and the optimal value. We double check this, and it is really the results for the relaxed Model 2. Unfortunately, it is too computational expensive, and we cannot run instances larger than 100 vertices to test the limits of this relaxation. The surprise was Model 1 giving a worst lower bound than the lower bound (MST + SSSP). Also, we can see that the Lagrangian Relaxation, on average, provides a good bound, hitting, in some cases, the optimal value. But it also worth mentioning that it is not suitable for large-scale instances, due to its computational constraints.

Chapter 5

Conclusions

Summarizing the achievements of this research, we presented a set of edge reduction rules. These rules removed in our experiments, on average, 98.65% of the edges. As consequence, we were able to run fast heuristics and achieve better results on our local search algorithms. We can also highlight that two of those reduction rules are safe, therefore can be used to achieve exact results as it does not remove any edge used in optimal solutions.

We also presented a fast local search algorithm, suitable for large-scale instances, capable of significantly improving a heuristic solution. In our experiments, the local search achieved improvements of over 8x in the gap to the lower bound.

We developed new heuristics, such as a new GRASP based heuristic for the problem. We also presented a new variant, called the Cable Trench Problem with Demands. We used it on a new heuristic that has shown good experimental results and the potential to solve even larger scale instances.

We performed a comprehensive experiment on a time limit of 10 minutes for each instance, running our main instance set of 310 instances. Our solutions had an average gap to the lower bound of 6.149%, solving 75% of the instances below 10.2% gap. The largest gap of the best algorithm stays at 37.898%.

We have also performed an experiment to compare different lower bound algorithms for small size instances, in this research, we found that one of the MILP models when relaxed achieved a near 0% gap to the optimal solution, opening questions for further investigations.

5.1 Future work

We expect further experimental investigations to analyze the lower bound quality for the problem, aiming to study how the lower bound gap is compared to optimal solutions. We would also like to expand the instance-scale limitations for other lower bound methods, especially for MILP related lower bounds that have presented near 0% gap as result on small instances.

This research has also raised the benefits of applying edge reduction rules on CTP

instances for heuristics and exact algorithms. We believe that further investigations should be carried out to look for new edge reduction rules and kernelization techniques in general, especially regarding safe reduction rules.

Our local search procedure demonstrates a considerable improvement in our heuristic solutions. We expect that in a future research, new local search neighborhoods could be explored. Alternatively, techniques such as a Variable Neighbourhood Search could also be applied to look for local improvements.

Our results on the Reduce and Solve Framework are encouraging, especially due to the potential of solving instances of a larger size, and should be validated by an even larger-scale instances experiment. Other heuristics could also be tested to the CTPD, including genetic algorithms, and other linear programming based heuristics. Another fundamental issue for a future research is to explore new ways to represent a CTP instance as a CTPD instance, such as a k-minimal spanning forest, dividing into grids, and others.

Bibliography

Hatice Calik, Markus Leitner, and Martin Luipersbeck. A benders decomposition based framework for solving cable trench problems. *Computers Operations Research*, 81:128 – 140, 2017. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2016.12.015. URL `http://www.sciencedirect.com/science/article/pii/S0305054816303124`.

E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

J. N. Girard, P. Zarka, M. Tagger, L. Denis, D. Charrier, and A. Konovalenko. Antenna design and distribution for a lofar super station in nançy. In *General Assembly and Scientific Symposium, 2011 XXXth URSI*, pages 1–4, 2011. doi: 10.1109/URSIGASS.2011.6051277.

José Fernando Gonçalves and Mauricio G. C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, Oct 2011. ISSN 1572-9397. doi: 10.1007/s10732-010-9143-1. URL `https://doi.org/10.1007/s10732-010-9143-1`.

Amin Jamili and Farshad Ramezankhani. An extended mathematical programming model to optimize the cable trench route of power transmission in a metro depot. *International Journal of Transportation Engineereing*, 3(2):109–123, 2015. ISSN 2322-259X.

J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press. URL `https://projecteuclid.org/euclid.bsmsp/1200512992`.

Vladimir Marianov, Gabriel Gutiérrez-Jarpa, Carlos Obreque, and Oscar Cornejo. Lagrangean relaxation heuristics for the p-cable-trench problem. *Comp. Oper. Res.*, 39(3):620–628, 2012. ISSN 0305-0548. doi: 10.1016/j.cor.2011.05.015. URL `http://dx.doi.org/10.1016/j.cor.2011.05.015`.

R. H. Nielsen, M. T. Riaz, J. M. Pedersen, and O. B. Madsen. On the potential of using the cable trench problem in planning of ict access networks. In *ELMAR, 2008. 50th International Symposium*, volume 2, pages 585–588, 2008.

R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

M. G. C. Resende and C. C. Ribeiro. GRASP and path-relinking: Recent advances and applications. In T. Ibaraki and Y. Yoshitomi, editors, *Proceedings of the Fifth Metaheuristics International Conference (MIC2003)*, pages T6–1 – T6–6, 2003.

U. Rocha, N. Ramos, L. Melo, M. Benedito, A. Silva, R. Cano, F. Miyazawa, and E. Xavier. Abordagens heurísticas para o p-cabo-trincheira com localização de instalações. *Encontro de Teoria da Computação*, 2(1/2017), 2017. ISSN 2595-6116. URL `http://portaldeconteudo.sbc.org.br/index.php/etc/article/view/3196`.

Ulysses Rocha. Ulyssesrocha/ctpinstances: Instances, October 2018. URL `https://doi.org/10.5281/zenodo.1467759`.

Silvia Schwarze. The multi-commodity cable trench problem. In *Proceedings of the Twenty-third European Conference on Information Systems, ECIS 2015 Completed Research Papers*, 2015.

Francis J. Vasko, Robert S. Barbieri, Brian Q. Rieksts, Kenneth L. Reitmeyer, and Kenneth L. Stott Jr. The cable trench problem: combining the shortest path and minimum spanning tree problems. *Computers & Operations Research*, 29(5):441 – 458, 2002. doi: http://dx.doi.org/10.1016/S0305-0548(00)00083-6.

Francis J. Vasko, Eric Landquist, Gregory Kresge, Adam Tal, Yifeng Jiang, and Xenophon Papademetris. A simple and efficient strategy for solving very large-scale generalized cable-trench problems. *Networks*, 67(3):199–208, 2016. ISSN 1097-0037. doi: 10.1002/net.21614. URL `http://dx.doi.org/10.1002/net.21614`.

Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience publication, 1998.