POLITECNICO DI TORINO

Master degree course in
Mathematical Engineering

Master Degree Thesis

Heuristics for the solution of Mixed-Integer Linear Programming problems

Supervisor
Prof. Paolo Brandimarte

Candidate
Lorenzo Bonasera

Academic year 2020-2021

Alla mia famiglia

Summary

Many discrete optimization problems can be reframed as Integer Linear Programming (ILP) problems, whose complexity class is NP-Hard. Due to its NP-hardness, solving a ILP problem to optimality often requires a huge computational effort. Therefore, it's necessary to adopt greedy algorithms and heuristics that are capable of finding satisfying solutions at a lower cost. The most common heuristics don't require a mathematical model and they are based on local search, e.g. tabu search, or population search, like genetic algorithms or particle swarm optimization. Those heuristics are not easy to apply in case of complicated constraints and/or when dealing with optimization problems that contain both continuous and discrete decision variables, known as Mixed-Integer Linear Programming (MILP) problems, hence the need of more adaptable and general heuristics. Thus, different algorithms have been proposed, like fix-and-relax, iterated local search and kernel search, based on mathematical models which impose proper forms of restriction on decision variables. This work aims to investigate and apply the kernel search framework on a specific case of study, in order to propose a reliable and well-performing problem-specific variation of the heuristic. The chosen case study is a stochastic version of the classical multi-item Capacitated Lot-Sizing Problem (CLSP), in which demand uncertainty is modelled through a scenario tree, resulting in a multi-stage mixed-integer stochastic programming model.

Acknowledgements

I miei ringraziamenti vanno alle persone che hanno contribuito, direttamente o indirettamente, ad arricchire e rendere migliori i giorni dei miei anni universitari, dei quali la scrittura di questa tesi rappresenta la degna conclusione.

Ringrazio innanzitutto il Prof. Paolo Brandimarte per aver accettato di essere il mio relatore, e per aver impartito durante questi anni preziosi insegnamenti, spesso volti a far cadere dal piedistallo, giustamente, noi studenti di ingegneria matematica.
Ringrazio i miei più cari amici, coloro che sono entrati a far parte della mia vita e dalla quale non sono più usciti. I miei compagni del collegio, in particolare: Laura, Simona, Mattia, Rosalinda e Adele, i quali hanno riso con gusto persino alle mie peggiori battute; Francesca, per aver fatto luce nei momenti più bui a ritmo di bachata; Davide Falcone, che grazie alla sua fantasia è riuscito a tirarmi su di morale in qualsiasi contesto; Luciana, perché è stata in grado di ridimensionare il mio ego; Michele, lui che le euristiche le ha sempre sapute, per il suo incrollabile supporto; Davide Rossetti, per essere sempre stato al mio fianco durante le nostre indimenticabili avventure, pandemia o meno.
Ringrazio i miei due *pupilli*, Sebastiano ed Emiliano, che non hanno mai smesso di essere presenti e vicini a me, pronti a teorizzare l'impossibile.
Ringrazio i miei due *maestri*: Francesco, perché gli devo veramente tutto, e Sabrina, per avermi aiutato a comprendere meglio me stesso.
Ringrazio Luca, che mi ha sostenuto di fronte alle incertezze della vita. Salvatore, con il quale ho affrontato le assurdità della stessa. Luigi, perché un lontano giorno alla villa Farina decidemmo di collezionare i nostri aneddoti più divertenti, e da quel momento non abbiamo più interrotto.
Ringrazio infine i miei genitori, mia sorella e i parenti più stretti, a cui questa tesi è dedicata, per aver creduto in me più di ogni altra cosa.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Discrete optimization problems are ubiquitous in almost every engineering field, since the real world is not granted to be continuous. Those problems are involved in many of the fundamental models of Operational Research: application domains are often related to logistics, which includes a very broad collection of optimization models, finance, networks and optimal transportation. Indeed, discrete variables in mathematical linear models are useful to represent binary decisions, *e.g.* if setup of industrial machinery is carried out or not, and to express integer quantities, for example in purchase or production decisions. The discipline of Mathematical Programming provides a set of tools to deal with discrete linear optimization problems, that are formulated through Integer Linear Programming (ILP) models.

The presence of discrete variables in optimization models makes problems very challenging, since it introduces nonconvex feasible regions, significantly complicating solution processes. In a very restricted number of cases, such as some network flow problems and linear assignment problem, it is possible to solve the continuous relaxation of the optimization model obtaining an integer solution. Otherwise, computationally expensive exact algorithms, *e.g.* Branch and Bound, are needed to solve ILP problems to optimality: those algorithms perform a tree-based search in order to explore through an enumeration process the whole feasible region, requiring a computing time which grows exponentially with the dimension of the problem, in the worst case. Hence, greedy algorithms and heuristic strategies are usually performed when dealing with very complex problems, aiming to achieve satisfying suboptimal solutions with a tolerable computational effort. Those heuristics are often based on searching processes that restrict the whole feasible set to local regions: in order to contrast the exponential growth of different solutions, it is indeed necessary to freeze a certain number of discrete variables, making the problem easier to solve at the expense of a controllable degree of suboptimality.

Mathematical Programming can also model optimization problems that contain both discrete and continuous variables, formulating them as Mixed-Integer Linear Programming (MILP) models. The most common heuristics, such as tabu search, genetic algorithms or particle swarm optimization, are not based on mathematical models and they struggle to tackle those mixed problems. Therefore it is necessary to adopt alternative strategies that can handle MILP problems in a flexible and efficient way, *e.g.* fix-and-relax or beam search, properly imposing specific restrictions to integer variables that can evolve during the solution process. This work aims to investigate, develop and test a general heuristic framework known as Kernel Search, comparing its different versions to standard and solid approaches, and to apply it to a challenging case study related to Operational Research.

The case study we propose is a stochastic variation of the well known dynamic lot-sizing problem, that was first formulated in Brandimarte [2006]. It consists of a multi-item Capacitated Lot-Sizing Problem (CLSP) in which demand uncertainty of items is modeled through a scenario tree over different time periods, constructed with discrete probabilities. Such fixed-charge lot-sizing problem is formulated through a model based on plant-location, in order to tighten big-M constraints and reducing the integrality gap, by disaggregating each item production over different time periods. Due to its dynamic and stochastic nature, this strong reformulation of the lot-sizing problem contains a huge number of discrete and continuous variables, proving to be an effective case study to test properties and features of the Kernel Search heuristic framework.

This thesis is organized as follows. In Chapter 2 we shortly expose the general mathematical theory behind Mixed-Integer Linear Programming models, including the most common solving techniques and approaches. In Chapter 3 we introduce the Kernel Search heuristic framework, giving an exhaustive presentation of different implementations and related algorithms. In Chapter 4 we present the testing environment and our performance evaluation policy, comparing heuristics to exact methods and analysing experimental results in an extensive manner. Finally, in Appendix A and B we extensively report code and tables.

Chapter 2

Mixed-Integer Linear Programming

We define a general Mixed-Integer Linear Programming problem in the following way:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & z = \mathbf{c}^\intercal \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b}, \\ & x_j \geq 0 \qquad j \in \mathcal{C}, \\ & x_j \in \{0,1\} \qquad j \in \mathcal{B}, \\ & x_j \in \mathbb{Z}_{\geq 0} \qquad j \in \mathcal{I} \end{aligned} \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of decision variables, $\mathbf{c} \in \mathbb{R}^n$ is the vector of objective coefficient, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the matrix containing inequality constraints coefficients and $\mathbf{b} \in \mathbb{R}^m$ is the right-hand side coefficients vector. Indices of decision variables vary on three different sets: $\mathcal{C}$ represents the set of continuous variables, $\mathcal{B}$ denotes the set of binary variables and $\mathcal{C}$ stands for the set of integer variables. This formulation is based on the canonical form of a Linear Programming (LP) model. The optimization problem is called *Mixed-Integer* because it contains both continuous and integer decision variables. It is said to be a *binary* 0-1 MILP problem whether $\mathcal{I} = \emptyset$, while it is called a *pure* Integer Linear Programming problem when $\mathcal{C} = \emptyset$.

For sake of simplicity and without loss of generality, we restrict our dissertation to binary 0-1 MILP problems, thanks to the following

Observation. *Every MILP problem can be reformulated as a binary 0-1 MILP problem.*

Indeed, given a general MILP problem in the form 2.1, it is possible to reframe it as a binary 0-1 MILP problem through properly transforming each integer decision variable Liberti et al. [2009]. We first suppose that every integer variable x_j, $j \in \mathcal{I}$ has domain $\mathcal{D}_j = \{L_j, \ldots, U_j\}$ in the set of positive integer numbers $\mathbb{Z}_{\geq 0}$, where L_j, U_j are its lower and upper bound respectively. Then, we apply the following transformation $\forall\, j \in \mathcal{I}$:

1. $\forall\, i \in \mathcal{D}_j$, add a binary variable $x_j^{(i)}$ to $\mathcal{B}$;
2. add a constraint $\sum_{i \in \mathcal{D}_j} x_j^{(i)} = 1$ to the original problem;
3. replace all occurrences of x_j in the original problem with the expression $\sum_{i \in \mathcal{D}_j} i x_j^{(i)}$.

The presence of integrality requirements on decision variables implies the following limitations: the problem is not convex and it is not possible to make use of instruments belonging to real analysis, such as local derivatives and gradient methods. Moreover, LP standard solving techniques like the simplex algorithm cannot guarantee a feasible integer solution, and it is not possible to verify optimality of a solution by simply checking its reduced costs. When dealing with MILP problems, in order to verify optimality of a given solution it is necessary to explore the whole feasibility region through an enumeration approach, facing an exponential complexity in the worst case. For these reasons, MILP problems are classified as NP-Hard.

In order to tackle such difficulties, it is necessary to relax some integrality constraints of the original problem, and to generate simpler subproblems, hopefully reducing the computational effort.

2.1 LP relaxation

We define the *LP relaxation* of a given Mixed-Integer Linear Programming problem in the form 2.1 as the following optimization problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \tilde{z} = \mathbf{c}^\intercal \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b}, \\ & x_j \geq 0 \qquad \forall\, j \end{aligned} \tag{2.2}$$

where all the integrality constraints are dropped. This reformulation allows us to exploit all the advantages of the methods available for the solution of standard LP problems, gaining convexity. Such problem admits a feasible set $\tilde{S}$ that contains the feasible set S of the MILP problem: by solving the LP relaxation we find a lower bound on the optimal value of the original objective function. More rigorously:

Remark. *Given two feasibility sets $S \subseteq \tilde{S}$ and an objective function $f(x)$, it holds that*

$$\tilde{z}^* = \min_{x \in \tilde{S}} f(x) \leq \min_{x \in S} f(x) = z^*$$

For example, if we consider a binary MILP problem with only two decision variables x_1 and x_2, it's easy to see that its LP relaxation admits a larger solution space 2.1, that possibly contains a better optimal value. A first and immediate consequence is represented by the following theorem, that establishes an important relationship about feasibility regions of a general MILP problem and its continuous relaxation:

Theorem (LP infeasibility)**.** *Given a general MILP problem in the form 2.1, if its LP relaxation is infeasible, then the original MILP problem is also infeasible.*

Proof. We have that the feasible set of the continuous relaxation is empty, that is $\tilde{S} = \emptyset$. We also know that $S \subseteq \tilde{S}$, since $\tilde{S}$ contains all the feasible solutions of the original MILP problem. So it follows that $S \subseteq \tilde{S} = \emptyset \implies S = \emptyset$, that means the MILP problem does not admit a feasible solution. □

The converse of the above Theorem is not true in general: proving infeasibility of a MILP problem does not provide useful informations about its continuous relaxation.

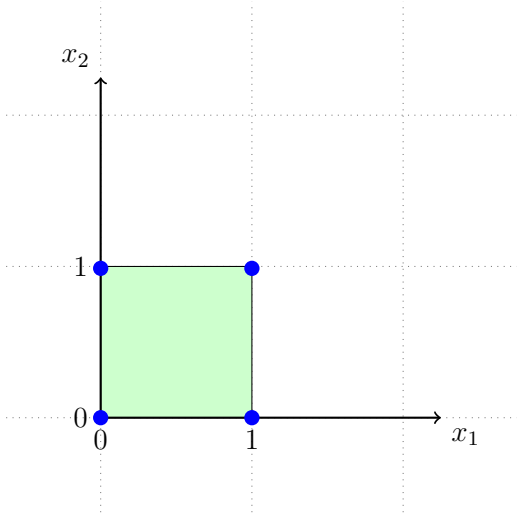


Figure 2.1: The binary MILP problem admits only four values as feasible solutions, while its LP relaxation has an enlarged dense feasible set, that is also convex.

Assuming that a given MILP problem is feasible and limited, it is needful to note that the optimal solution of the corresponding LP relaxation $\tilde{\mathbf{x}}^*$ is not necessarily integral, and any rounding approach does not guarantee feasibility for the starting problem. Indeed, finding a feasible solution for a MILP problem through rounding procedures for $\tilde{\mathbf{x}}^*$ is not an easy task. A method that has proven to be successful is the Feasibility Pump, which is a heuristic created by Fischetti et al. [2005]. This method aims to minimize the L^1-norm distance between a feasible but not necessarily integer solution of the LP relaxation $\tilde{\mathbf{x}} \in \tilde{S}$ and a point in the solution space obtained by rounding $\mathbf{x}_I$, but not necessarily feasible for the LP relaxation. In this way, the heuristic searches for a integer feasible solution for the MILP problem by iteratively reducing such distance.

If the rounded solution of the LP relaxation $\lfloor \tilde{\mathbf{x}}^* \rceil$ [1] is feasible for the MILP problem, the corresponding value of the objective function $\tilde{z}_I^*$ is not too distant from the MILP optimal value z^*. Let us clarify the most general relationship between those three values through the following picture:

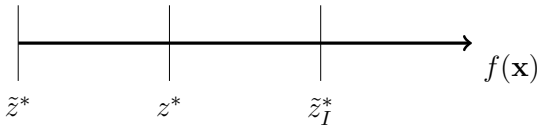


Figure 2.2: Relationship between optimal value of the LP relaxation, optimal value of the MILP problem and objective function value of the rounded solution.

It is easy to see that $\tilde{z}^*$ provides a lower bound for the optimal value of the integer problem, while the latter acts as a lower bound for the value of the rounded solution $\tilde{z}_I^*$. The following Theorem states the relationship between those values in a specific case, when solving the LP relaxation already provides an optimal integer solution:

Theorem (LP integrality)**.** *Given a general MILP problem in the form 2.1 and the corresponding LP relaxation in the form 2.2, if the optimal solution of the LP relaxation $\tilde{\boldsymbol{x}}^*$ is feasible for the original problem, then it is optimal also for the MILP problem, and it holds that $\tilde{z}^* = z^*$.*

Proof. We distinguish two cases: $S = \tilde{S}$ and $S \subset \tilde{S}$. In the first case, the two feasibility

[1] We denote with $\lfloor \cdot \rceil$ the function that rounds each vector component to the nearest integer.

regions coincide, so do the two problem formulations 2.1 and 2.2, and it holds that $\tilde{z}^* = z^*$. In the second case, we have that $\tilde{\mathbf{x}}^*$ is the optimal solution within the set $\tilde{S}$. We also have that $\tilde{\mathbf{x}}^* \in S$, since it is feasible for the MILP problem by hypothesis. Then, $\tilde{\mathbf{x}}^*$ must be optimal for the problem having S as feasibility region, that is the original MILP problem: it follows that $\tilde{z}^* = z^*$. □

LP relaxation is, thus, a crucial technique to solve MILP and ILP problems, and it is a fundamental step of almost every algorithm and heuristic.

2.2 Branch and Bound

The first and most general solution approach to discrete optimization problems is called *Branch and Bound*. It is based on a recursive decomposition principle, used in conjunction with a standard non-integer solution method. The way this approach works is based on its name, and we will describe it focusing on binary minimization problems without loss of generality.

Given a 0-1 MILP, it is necessary to explore its whole set of feasible solutions by enumerating it, in order to find the optimal one: it's natural to consider the solution space as a binary search tree, that starts at the root node with the original problem. At the root node, the solution of the corresponding LP relaxation $\tilde{\mathbf{x}}^*$ is computed. Then, the tree *branches* on a binary variable x_j, $j \in \mathcal{B}$, whose component in the solution vector is continuous, by fixing its value, generating a left node that contains the subproblem with $x_j = 0$, and a right node that contains the symmetric subproblem with $x_j = 1$. Again, the corresponding LP relaxation of each subproblem is solved, obtaining a partial continuous solution $\tilde{\mathbf{x}}_j$. The branching process is recursively applied until every component of vector $\tilde{\mathbf{x}}_j$ associated with a binary variable has an integer value, generating leaf nodes that represent every candidate solution. The picture 2.3 describes the branching process applied to a toy binary problem that contains three decision variables.

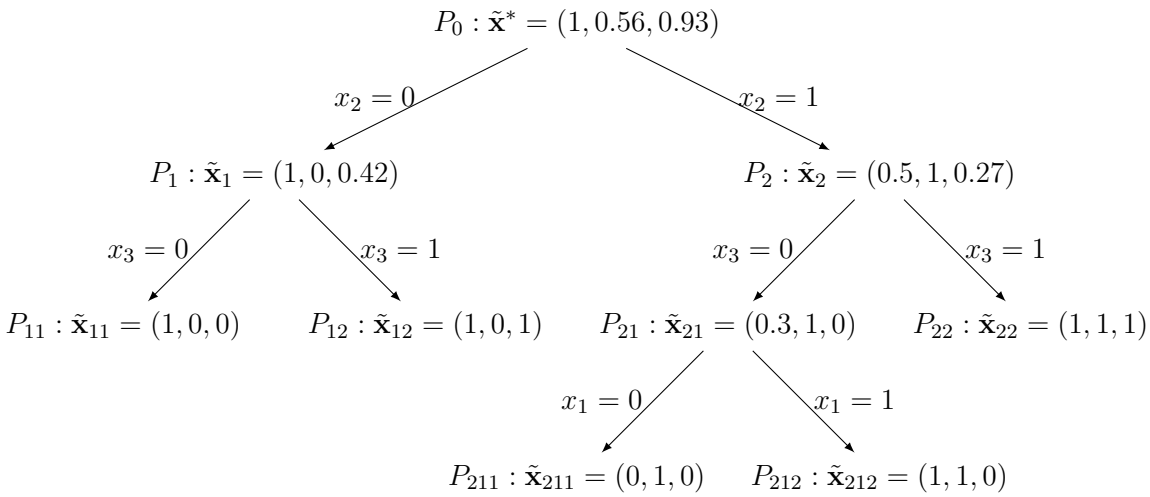


Figure 2.3: Example of search tree for a toy problem.

The number of generated leaves in the worst case is $2^{|\mathcal{B}|}$, showing the exponential complexity of a MILP problem. In order to reduce computational effort, the Branch and Bound method *prunes* some nodes of the tree. This pruning process is done by exploiting the optimal value of the LP relaxation $\tilde{z}_j^*$ of each node, that works as a lower *bound* for all the integer solutions that correspond to its children nodes.

If an integer solution of the LP relaxation at a certain node is found, the value of its objective function becomes the *incumbent solution* ν^*, that is updated every time an integer solution with a better value is met. If another node in the tree presents a lower bound that is greater than the current incumbent solution, that is $\tilde{z}_j^* \geq \nu^*$, that node is pruned, since its branching cannot contain better solutions. Is it possible to reduce the computational time guaranteeing a defined grade of suboptimality, relaxing the above condition in absolute terms:

$$\tilde{z}_j^* \geq \nu^* - \epsilon$$

introducing a threshold ϵ representing the minimal absolute improvement over the incumbent solution, or in relative terms:

$$\tilde{z}_j^* \geq \nu^*(1 - \epsilon)$$

where ϵ represents the maximum percentage suboptimality. If the LP relaxation at a certain node proves to be infeasible, the algorithm prunes that node by Theorem LP infeasibility. There are several exploration rules that this method can follow in order to choose the branching node. A simple strategy is knows as *depth first*, in which the algorithm branches until it arrives to a leaf, exploring the depth of the tree: in this way, it easier to find feasible solutions, at the risk of exploring a subtree that is not promising. Another strategy is called *best first*, in which the algorithm chooses the most promising node, that is the one with the lowest LP relaxed objective value: this reduces the total number of branches, at the cost of slowing down the search for a feasible solution. Different mixed strategies are also found in literature, and they are beyond the purpose of our work.

The Branch and Bound method is over when all the partial solutions are branched or pruned. In this case, if an incumbent solution ν^* is found, it coincides with the optimal feasible solution for the MILP problem by Theorem LP integrality, otherwise such problem is declared infeasible. Is it possible to terminate the algorithm at a proper node by imposing another suboptimality condition:

$$\frac{\nu^* - \tilde{z}_j^*}{\frac{1}{2}|\nu^* + \tilde{z}_j^*|} \leq \sigma$$

where σ is a given threshold representing the minimum improvement required, in terms of distance between incumbent solution and the value of the found LP relaxed solution. Thanks to its reliability and accuracy, the Branch and Bound approach works as a basis for many other solution methods for discrete optimization problems, such as exact algorithms like *Branch and Cut* and various heuristics contained in commercial MILP/ILP solvers.

2.3 Case study: Capacitated Lot-Sizing Problem

We state our MILP case study as follows, basing our work on Brandimarte [2006]: a stochastic version of the multi-item Capacitated Lot-Sizing Problem (CLSP), in which demand uncertainty is represented by means of a scenario tree containing discrete probabilities. The proposed problem is modeled through a plant-location strong formulation, in order to reduce the integrality gap and make the optimization problem less computationally intensive: in this way, lower bounds obtained through the LP relaxation are more informative and related underestimation of the true objective value is less impacting for

Branch and Bound pruning. The plant location formulation is based on disaggregating items production on future time periods, rather than producing items only to meet the current demand: the optimization model can be thought as a network flow problem, in which supply and demand nodes have different time indexes. The idea is to ship commodities and products in time instead of in space, at the expense of inventory costs rather than transport ones. We assume that the most strict capacity is fixed and coincides with the bottleneck resource.

The deterministic formulation of the CLSP model is the following one:

$$\begin{aligned}
\text{minimize} \quad & \sum_i \sum_t \sum_{p \geq t} h_i(p-t) y_{itp} + \sum_i \sum_t f_i s_{it} && && \text{(2.3a)} \\
\text{subject to} \quad & \sum_{t \leq p} y_{itp} \geq d_{ip} && \forall\, i, p, && \text{(2.3b)} \\
& y_{itp} \leq d_{ip} s_{it} && \forall\, i, t, p \geq t, && \text{(2.3c)} \\
& \sum_i \sum_{p \geq t} r_i y_{itp} + \sum_i r'_i s_{it} \leq R && \forall\, t, && \text{(2.3d)} \\
& y_{itp} \geq 0, && && \text{(2.3e)} \\
& s_{it} \in \{0, 1\} && && \text{(2.3f)}
\end{aligned}$$

and each term is described as follows:

- y_{itp} is a decision variable, representing the amount of item i that is produced during time period t in order to meet demand in current or future time period p;
- s_{it} is a decision variable, representing the binary setup variable of item i produced in time period t;
- d_{ip} stands for the deterministic demand of item i during current of future time period p;
- h_i and f_i are respectively the inventory and setup costs for item i;
- R represents the capacity of the bottleneck resource;
- r_i and r'_i denote unit processing time and setup time for item i.

The objective function 2.3a contains two terms: the first one represents the overall inventory cost that is protracted over time, and the second one stands for the total setup cost. The aim is to satisfy demand at minimum cost. Constraints are explained in the following manner:

- 2.3b ensures that demand is always satisfied through present and past production;
- 2.3c is the big-M constraint and is related to binary setup decisions;
- 2.3d limits the total cost in terms of total processing and setup time;
- 2.3e and 2.3f stand for the non-negativity condition of production amount and binary nature of setup decision variables, respectively.

Plant location formulation has an evident benefit: the big-M constraint is more tight since it coincides with the demand in only one time period, resulting in more useful lower bounds to be computed in LP relaxation and a consequently better pruning for Branch and Bound-based solving methods. However, there are a few downsides. First of all, the model is not elastic: in extreme scenarios, it may happen that the problem becomes infeasible, leaving the user without any result. Secondly, it disregards nonlinearities in transportation costs and uncertainty in demand and capacity. Finally, this model does not take into account any end-of-horizon effects: the end inventory is always zero in optimal solutions, and this can contrast with some concrete applications. Furthermore, possible leftover inventory is not optimized. The solution adopted in Brandimarte [2006] is to equip the above model with a parsimonious but effective representation of demand uncertainty, reformulating the optimization problem to make it more elastic and to partially overcome end-of-horizon effects.

2.3.1 Uncertainty representation

A simple and functional way to represent demand uncertainty is to adopt a tree-based scenario generation, equipped with a discrete probability distribution of items demand, making the problem multi-stage. The unique root node of the tree represents the current state, in which demand value becomes realized and first-stage decision must be taken. For each future time period, the tree branches creating other nodes and modeling uncertain scenarios. Then, future planning decisions are made depending on expected values of the successive branching nodes. In Figure 2.4 we depict an example of a tree-based scenario. Each branch in the tree contains a conditional probability that is used to compute expected value of total costs, forming the objective function of the extended model. By considering the expected value of costs we are assuming a risk-neutral attitude of the user, which needs not to represent the most general case. The branching structure of the tree can be useful to partially mitigate end-of-horizon effects: it is indeed possible to add a branching factor equal to 1 to the end in order to add other nodes, eliminating the uncertainty factor but extending the time horizon without generating other scenarios.

We remark that the stochastic structure of this model is non-anticipative: the demand value is only realized during the current time bucket (root node at the beginning), while future demand is completely unknown, except its expected value. In this way, it is possible to separate the single-stage decision into multi-stage different decisions. The addiction of this stochastic structure makes the plant-location reformulation three-dimensional: items are shipped between different time periods, so the same amount of items must be transported to all the nodes belonging to target period, in order to match the worst case demand scenario.

The stochastic model can be further enriched. Introducing leftover inventory as a new set of decision variables it is possible to manage over-hedging situations, in which future demand is overestimated and there is an amount of unsold items, making the model more elastic: we assume that products are not perishable and that unsold items can satisfy future demand. Another useful addiction is to permit lost sales, implementing another set of decision variables to control the amount of loss with a suitable penalization, making the model elastic to extreme demand scenarios and allowing the user to also manage under-hedging situations. The following notation is used to describe the stochastic tree-based structure, where n is a generic node of the scenario tree $\mathcal{N}$:

- $\mathcal{T}$ is the set of leaves in the tree;
- $T(N)$ returns the time period of the input node;
- $a(n)$ returns the immediate predeccessor for the input node;

- $\Omega(n,t)$ returns the unique ancestor of node n at time period t;
- $\Sigma(n,t)$ returns the collection of successor nodes of the input node at time period t.

We formulate the extended stochastic multi-stage version of the optimization problem 2.3 (SCLSP) as follows:

$$\text{minimize} \quad \sum_{n\in\mathcal{N}} p^{[n]} \left[\sum_i \left(f_i s_i^{[n]} + h_i I_i^{[n]} + g_i z_i^{[n]}\right)\right] + \sum_{n\in\mathcal{N}\setminus\mathcal{T}} p^{[n]} \left[\sum_i \sum_{\tau>T(n)} h_i\left(\tau - T(n)\right) y_{i\tau}^{[n]}\right] \tag{2.4a}$$

$$\text{subject to} \quad I_i^{[a(n)]} + \sum_{t<T(n)} y_{i,T(n)}^{[\Omega(n,t)]} + y_{i,T(n)}^{[n]} = d_i^{[n]} + I_i^{[n]} - z_i^{[n]} \qquad \forall\, i,n, \tag{2.4b}$$

$$y_{i\tau}^{[n]} \le \left(\max_{j\in\Sigma(n,\tau)} d_i^{[j]}\right) s_i^{[n]} \qquad \forall\, i,n,\tau > T(n), \tag{2.4c}$$

$$y_{i,T(n)}^{[n]} \le d_i^{[n]} s_i^{[n]} \qquad \forall\, i,n, \tag{2.4d}$$

$$\sum_i \sum_{\tau\ge T(n)} r_i y_{i\tau}^{[n]} + \sum_i r_i' s_i^{[n]} \le R \qquad \forall\, n, \tag{2.4e}$$

$$y_{i\tau}^{[n]},\ I_i^{[n]},\ z_i^{[n]} \ge 0, \tag{2.4f}$$

$$s_i^{[n]} \in \{0,1\} \tag{2.4g}$$

where there are mode decision variables, each one indexed through the corresponding node of the tree. More in detail:

- $y_{i\tau}^{[n]}$ represents the amount of item i that is produced in node n in order to meet demand in time period $\tau \ge T(n)$;
- $s_i^{[n]}$ stands for the binary setup variable for item i produced in node n;
- $I_i^{[n]}$ represents the leftover inventory of unsold item i in node n;
- $z_i^{[n]}$ is the decision variable related to unmet demand, that is lost sales of item i at node n;

with the additional coefficients:

- $p^{[n]}$ is the unconditional probability of node n;
- g_i is the penalty for unmet demand of item i;

The objective function 2.4a contains two terms. The first one expresses the expected value of the overall cost, including inventory leftover, setup and unmet demand. The second one is the expected value of the inventory costs over time, where the set of leaves is not considered. Constraints are explained as follows:

- 2.4b acts as a balance equation a given node n. Indeed, the left side including past leftovers, previous and current production towards a given time period t has to match the right side, containing demand of node n, planned inventory leftover and unmet demand;

- 2.4c is the *future* big-M constraint, since it is related to binary setup decisions for future demand nodes. Note that only the maximum of future demands is considered, making the integrality gap stricter;
- 2.4d is the *current* big-M constraint, for setup decision of the same time bucket of demand nodes;
- 2.4e is the same of the simpler model, expressing the limit of capacity;
- 2.4f and 2.4g stand for the non-negativity condition of production amount, inventory leftovers and unmet demand, whereas setup variables are once again constrained to be binary.

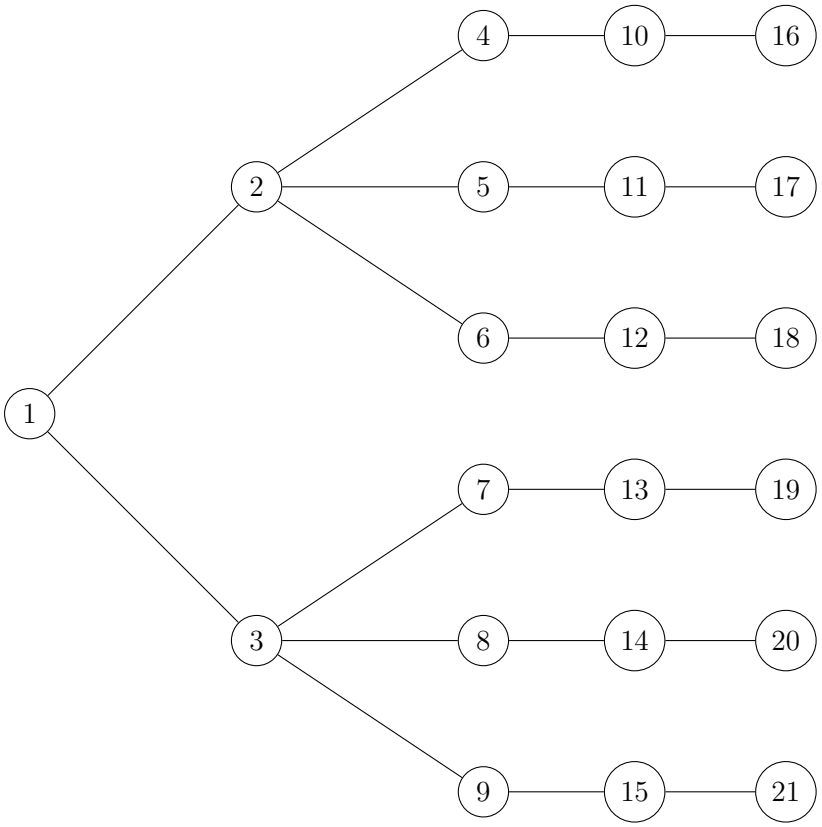


Figure 2.4: Tree-based scenario with 5 time periods and branching structure $(1, 2, 3, 1, 1)$.

It becomes evident how the Stochastic reformulation of the multi-item CLSP (SCLSP) may result in very complex and hard to solve optimization problems, especially depending on the branching structure of the tree: the larger is the number of nodes, the larger becomes the number of continuous and binary variables. For this reason, it constitutes a challenging problem for MILP solving methods.

Chapter 3

Kernel Search

The Kernel Search was first introduced in Angelelli et al. [2012] and it is a heuristic framework that can be applied to any MILP problem with binary variables. As a heuristic, Kernel Search is able to solve problems whose complexity cannot be tackled by an exact algorithm, providing high-quality solutions in a practicable amount of time. The fundamental idea of Kernel Search it's suggested by its name: to decompose the original problem by properly restricting its feasibility region into a *kernel*, and to progressively expand it by *searching* for different feasible solutions. The framework relies on a general-purpose LP and MILP solver, in order to optimally solve each restricted sub-problem that is obtained: solutions found by the solver provide useful informations about decision variables and accordingly lead the exploration of feasibility region. In this work, we base our own Kernel Search implementation on a commercial mathematical optimization solver called Gurobi Gurobi Optimization, LLC [2021].

There are three major features that make Kernel Search framework reliable and effective. First of all, it is structured as a very intuitive and flexible greedy algorithm: being a heuristic approach, it tackles hard problems in a clever and easily customizable way, shredding their complexity to a simpler level, as a human mental process. Secondly, it relies on a well performing optimization solver that tackles the most demanding part of the search, thus requiring an easier implementation than other heuristics. Last but not least, Kernel Search is very adaptable according to the given problem: indeed, it can either perform as a general-purpose heuristic and as a task-specific method. The way in which the heuristic explores solution space is based on a trade-off between quality of the solution and spent execution time, that can be tuned through a set of parameters: hence, the algorithm can easily be adapted to a specific problem by properly adjusting its exploration policy. This adaptation process can also be automatic: the heuristic can assess by itself the difficulty of the problem, and it can suitably change its solving strategy.

3.1 Basic Kernel Search

In this section we describe the basic implementation of Kernel Search applied to binary MILPs. Our baseline of this heuristic is based on revisited versions of the works of Angelelli et al. [2010] and Guastaroba and Speranza [2012]. We implement our version of the heuristic by using Python as programming language, that works very well with the Gurobi interface. The main procedure is divided into two steps: *initialization* phase and *extension* phase.

3.1.1 *Initialization* phase

During the first step, the LP relaxation of the original MILP problem is solved: if it proves to be infeasible, then by Theorem LP infeasibility the query problem is declared infeasible, and the algorithm stops. Otherwise, the heuristic exploits the found optimal solution vector $\tilde{\mathbf{x}}$ to define the *initial kernel* K, that is the starting set of variables on which the original problem is restricted: if a binary variable x_j, $j \in \mathcal{B}$ takes a strictly positive value in the LP relaxed solution vector $\tilde{\mathbf{x}}$, it is added to the set K. In this way, the heuristic tries to identify which binary variable is more promising, hopefully reducing the problem dimension. Since MILPs may contain continuous variables x_j, $j \in \mathcal{C}$, those are automatically added to the initial kernel. The idea is to reduce problem dimension by restricting the total number of integer variables, obtaining a subproblem *MILP(K)*[1] that is easier to solve using Gurobi.

If *MILP(K)* is feasible and it is solved, the found feasible and possibly optimal solution is memorized and its value is saved as the *cut-off value* ν^*: it works as a lower bound to be reached for each successive subproblem. If the problem is infeasible, ν^* is set to $+\infty$. Since *MILP(K)* may either be infeasible or may provide a low-quality solution with respect to the original problem, the inclusion of unused variables in the restricted problem can improve both situations. After defining the elements belonging to K, the algorithm *ranks* remaining variables depending on their reduced costs in the LP relaxed solution: the lesser the reduced cost of a variable $x_j \notin K$ is, the more promising the variable becomes. Thus, variables are ordered according to their expected goodness and they are passed to the next step, along with kernel K and cut-off value ν^*. The algorithm *Initialization* phase summarizes the mentioned procedure in pseudocode, and the corresponding Python code can be found in the Appendix A through listing *Initialization* phase code.

Algorithm 1: Kernel Search *inizialization* phase

Data: MILP problem and set $\mathcal{X}$ of n variables.
Result: Kernel K, set of remaining variables $\mathcal{X}$, solution $\mathbf{x}^*$ and cut-off value ν^*.
1: Solve the LP relaxation of the MILP problem;
if *LP relaxation is infeasible* **then**
  STOP;
else
  Get LP relaxed optimal solution $\tilde{\mathbf{x}} = (\tilde{x}_i, \ldots, \tilde{x}_n)$;
end
2: $K \leftarrow \emptyset$;
for $i \leftarrow 1$ *to* n **do**
  if $\tilde{x}_i > 0$ **then**
    $K \leftarrow K \cup \{x_i\}$;
    $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x_i\}$;
  end
end
3: Compute reduced costs of each variable $x \in \mathcal{X}$;
4: Sort variables $x \in \mathcal{X}$ in non-decreasing order of their reduced costs;
5: $\nu^* \leftarrow +\infty$;
6: Solve the restricted problem *MILP(K)*;
if *MILP(K) is feasible with optimal solution* $\boldsymbol{x}$ *and value* z **then**
  $\nu^* \leftarrow z$;
  $\mathbf{x}^* \leftarrow \mathbf{x}$;
end

[1] We denote with *MILP*$(\cdot)$ the starting MILP problem constrained to contain only the input set of variables.

3.1.2 *Extension* phase

During the second phase, sorted variables outside the kernel are partitioned into *buckets* of a given length L_b, generating a total number $N_b = \lceil \frac{n-|K|}{L_b} \rceil$ of buckets B_i, $i = 1, \ldots, N_b$. The goal is to afterwards expand the initial kernel K through the inclusion of variables that are contained in one bucket at time, solving a collection of subproblems *MILP(*$K \cup B_i$*)*, $i = 1, \ldots, N_b$. The parameter L_b controls the total number and the difficulty of those subproblems: a lower value corresponds to a higher number of simpler subproblems to be solved, since each one has a lower dimension. A higher value of L_b, instead, decreases the total number of subproblems, but each one becomes more complex.

Buckets $\{B_i\}_{i=1}^{N_b}$ are iteratively explored during the *extension* phase. It is possible to introduce a parameter $\overline{N}_b < N_b$ that controls the maximum number of buckets to be explored: the lower $\overline{N}_b$ is, the faster Kernel Search becomes, at the cost of a worse quality solution. Buckets are explored accordingly to the ranking that is assigned to variables during the previous phase: the first bucket to be explored contains the most promising variables, whereas the last bucket contains the least favourable ones. Each time a bucket B_i is processed, a new subproblem *MILP(*$K \cup B_i$*)* is created, with the following additional constraints:

$$\sum_{j \in B_i} x_j \geq 1 \tag{3.1}$$

$$\text{Cut-off value set to } \nu^* \tag{3.2}$$

The first constraint ensures that at least one variable of the bucket is included in the solution, whereas the second constraint establishes a bound to be reached, since we are interested only in those solutions that improve the current best one: if *MILP(*$K \cup B_i$*)* is feasible, it means that the value of the found solution is greater that ν^*, so the latter and the current best solution $\mathbf{x}^*$ are updated. If a variable belonging to the processed bucket $x_j \in B_i$ has a non-zero value in the solution of a feasible *MILP(*$K \cup B_i$*)*, it proves to be useful and it is permanently added to the kernel. If after processing all buckets no feasible solution has been found, that is $\nu^* = +\infty$, the algorithm stops and the original problem remains unsolved. Otherwise, the current best found solution $\mathbf{x}^*$ and its value ν^* are returned as final output. The algorithm *Extension* phase summarizes the mentioned procedure in pseudocode, and the corresponding Python code is situated in the Appendix A through listing *Extension* phase code.

3.2 Adaptive Kernel Search

One of the issues of the basic implementation of Kernel Search heuristic is that the algorithm does not directly search for a feasible solution of the initial restricted problem: if the complete MILP is feasible but the first *MILP(K)* proves to be infeasible, it is necessary to explore and include non-kernel variables before the generation of buckets, in order to hopefully obtain feasibility. One possible example of a worst case scenario is when feasibility is only achieved with the simultaneous presence of two variables belonging to different buckets: in that case, the algorithm will fail and the original problem will remain unsolved. The solution adopted by Guastaroba et al. [2017] is to properly expand the initial kernel K before the generation of buckets $\{B_i\}_{i=1}^{N_b}$, until a feasible solution is found. A further enlargement of the kernel before the generation of buckets can also be performed accordingly to the difficulty of the given instance: those features are what makes Kernel Search a reliable, efficient and flexible heuristic. After getting a feasible

Algorithm 2: Kernel Search *extension* phase

Data: MILP problem, kernel K, set of sorted variables $\mathcal{X}$, current best solution $\mathbf{x}^*$ and cut-off value ν^*.
Result: Optimal solution $\mathbf{x}^*$ and its value ν^*.
1: Partition $\mathcal{X}$ into buckets $\{B_i\}_{i=1}^{N_b}$ of length L_b;
for $i \leftarrow 1$ *to* $\min\{\overline{N}_b, N_b\}$ **do**
  2: Process bucket B_i;
  3: Add constraint $\sum_{j \in B_i} x_j \geq 1$ to *MILP(K)*;
  4: Set cuf-off value ν^* to *MILP(K)*;
  5: Solve augmented problem *MILP*$(K \cup B_i)$;
  if *MILP*$(K \cup B_i)$ *is feasible with (possibly optimal) solution* $\boldsymbol{x}$ *and value* z **then**
    for $j \in B_i$ **do**
      if $x_j > 0$ **then**
        $K \leftarrow K \cup \{x_j\}$;
        $\nu^* \leftarrow z$;
        $\mathbf{x}^* \leftarrow \mathbf{x}$;
      end
    end
  end
  6: Remove constraint $\sum_{j \in B_i} x_j \geq 1$ from *MILP(K)*;
end

solution, the heuristic computes how much time was consumed to obtain such solution, and it automatically assesses the difficulty of the original problem as consequence. This variation is called Adaptive Kernel Search (AKS). The aforementioned procedures are applied after the *Initialization* phase and right before the *Extension* phase, and we describe them in detail in the following subsections.

3.2.1 *Feasibility* step

The idea is to iteratively increase the size of the initial kernel until the restricted problem *MILP(K)* admits a feasible and possibly optimal solution, by permanently including a fixed amount of sorted variables. The number of variables that can be added to the kernel during each iteration is proportional to its initial size, and it is equal to $w \cdot |K|$, where $w > 0$ is a parameter set by the user. If there are remaining variables, they are passed to the successive phase; otherwise, since the kernel K contains all possible variables, the heuristic stops. The algorithm *Feasibility* step summarizes the mentioned procedure in pseudocode, and the corresponding Python code can be found in the Appendix A in the first part of listing *Adaptation* phase code.

3.2.2 *Adaptation* phase

During this phase, the heuristic firstly assess the difficulty of the original problem depending on how much time $t_{MILP(K)}$[2] was spent solving the last feasible *MILP(K)*, also considering the *Feasibility* step. An instance is classified in the following way:

- Easy if: $t_{MILP(\,\cdot\,)} \leq t_{Easy}$

[2]We denote with $t_{MILP(\,\cdot\,)}$ the time spent on solving the input problem.

Algorithm 3: Adaptive Kernel Search *feasibility* step

Data: MILP problem, kernel K, set of sorted variables $\mathcal{X}$.
Result: Kernel K, set of sorted variables $\mathcal{X}$, solution $\mathbf{x}^*$ and cut-off value ν^*.
while *MILP(K) is not feasible* **do**
  1: Extract from $\mathcal{X}$ the first $w \cdot |K|$ variables;
  2: Add to K extracted variables;
  3: Solve the restricted problem *MILP(K)*;
  if *MILP(K) is feasible with (possibly optimal) solution* $\boldsymbol{x}$ *and value* z **then**
    $\nu^* \leftarrow z$;
    $\mathbf{x}^* \leftarrow \mathbf{x}$;
  end
  if $\mathcal{X} = \emptyset$ **then**
    STOP;
  end
end

- HARD if: $t_{MILP(\,\cdot\,)} \geq t_{Hard}$ with no optimal solution
- NORMAL otherwise

where t_{Easy} and t_{Hard} are parameter specified by the user. If a problem is classified as EASY, the kernel is augmented by permanently adding to K a set of excluded integer variables K^+ of fixed size $q \cdot |K|$, where $q > 0$ is a given parameter, generating a new subproblem *MILP*$(K \cup K^+)$. Kernel enlargement is repeated until $t_{MILP(K \cup K^+)} \leq t_{Easy}$. During each iteration, the following constraints are added to each new restricted problem:

$$\sum_{j \in K^+} x_j \geq 1 \tag{3.3}$$

$$\text{Cut-off value set to } \nu^* \tag{3.4}$$

that are equivalent to 3.1 and 3.2. If *MILP*$(K \cup K^+)$ is solved to optimality, a better optimal solution $\mathbf{x}^*$ is found as consequence, with corresponding value z^* that updates the current cut-off value. We remark that the aim of this phase is not to get a feasible solution, because this task is accomplished by the previous one. The goal is to exploit the easiness of the current instance in order to expand kernel as much as possible, reducing the computational effort in the successive phase and hopefully obtaining better solutions. After this procedure, if there are no remaining variables, the algorithms stops: kernel K contains all variables and there is no point in executing the heuristic. Otherwise, left variables are partitioned into buckets as usual. Algorithm *Easy instance* summarizes the described procedure, and its corresponding Python code is situated in the Appendix A, in the second part of listing *Adaptation* phase code.

If an instance is classified as HARD, that is the last feasible *MILP(K)* was not solved to optimality within t_{Hard} and only a feaasible solution was found, it means that the dimension of the restricted problem is too difficult to handle through the commercial solver: the goal is to decrease the size of subproblems in the *Extension* phase by permanently fixing the value of some integer variables that are not in the kernel. More in detail, binary variables outside K are fixed according to the following criterion, that is based on the LP relaxed solution of the starting problem:

$$\text{if} \quad \tilde{x}_j > 1 - \epsilon \implies x_j = 1, \qquad \forall\, j \in \mathcal{B} \tag{3.5}$$

Algorithm 4: Adaptive Kernel Search *easy instance*

Data: EASY MILP problem, kernel K, set of sorted variables $\mathcal{X}$, current best solution $\mathbf{x}^*$ and cut-off value ν^*.
Result: Kernel K, set of sorted variables $\mathcal{X}$, solution $\mathbf{x}^*$ and cut-off value ν^*.
1: $K^+ \leftarrow \emptyset$;
while $t_{MILP(K)} \leq t_{Easy}$ **do**
  2: $K^+ \leftarrow$ the first $q \cdot |K|$ variables from $\mathcal{X}$;
  3: $K \leftarrow K \cup K^+$;
  4: Add constraint $\sum_{j \in K^+} x_j \geq 1$ to *MILP(K)*;
  5: Solve the restricted problem *MILP(K)*;
  if *MILP(K) is feasible with (possibly optimal) solution* $\boldsymbol{x}$ *and value* z **then**
    $\nu^* \leftarrow z$;
    $\mathbf{x}^* \leftarrow \mathbf{x}$;
  end
  if $\mathcal{X} = \emptyset$ **then**
    STOP;
  end
  6: Remove constraint $\sum_{j \in K^+} x_j \geq 1$ from *MILP(K)*;
end

where $\epsilon > 0$ is a user-defined parameter that controls the sensitivity of such condition: higher values of ϵ imply a more nervous heuristic, fixing more variables and thus obtaining quicker but possibly worse solutions. Lower values of ϵ, instead, lead the heuristic to explore more different variables, hopefully enhancing the quality of solutions, at the cost of a slower algorithm. If there are integer non-binary variables in the MILP original problem, those variables are fixed to the nearest integer in their corresponding value in the solution of the LP relaxation. Algorithm *Hard instance* summarizes the above procedure, and its corresponding Python code is situated in the Appendix A, in the second part of listing *Adaptation* phase code.

Algorithm 5: Adaptive Kernel Search *hard instance*

Data: HARD MILP problem, kernel K, set of sorted variables $\mathcal{X}$, current best solution $\mathbf{x}^*$ and cut-off value ν^*.
Result: Kernel K, set of sorted variables $\mathcal{X}$, solution $\mathbf{x}^*$ and cut-off value ν^*.
for $j \in \mathcal{X}$ **do**
  if $\tilde{x}_j > 1 - \epsilon$ **then**
    1: Add constraint $x_j = 1$ to *MILP(K)*;
  end
end

Finally, if an instance is classified as NORMAL, none of those algorithms is executed and the *Extension* phase takes place.

3.3 Time allocation

A proper allocation of computing time is a relevant point, for different reasons. First of all, it is necessary to balance the available total computing time between the three main phases: an unbalanced allocation may interfere with the exploration of other feasible solutions, or it may even prevent the heuristic to find a starting feasible solution. Secondly, it is mandatory to properly set available computing time in order to fairly compare Kernel Search to other heuristics or algorithms in terms of performance: since Kernel Search

actively exploits a commercial solver, it is required to limit the portion of time allocated to each call to the ILP/LP solver, and the total number of calls is not known from the beginning. Lastly, the *Adaptation* phase is specifically based on how much time is spent on solving different *MILP(K)* problems, so it becomes necessary to set parameters t_{Easy} and t_{Hard} accordingly to the available computing time. We thus present in detail our time allocation policy, based on the work of Guastaroba et al. [2017], for each step of Adaptive Kernel Search:

1. The user specifies the total available computing time t_{Max} for Adaptive Kernel Search.
2. The entire available time is allocated to the solution of the LP relaxation, updating the remaining available time $t_{Available}$. If the very first step of the heuristic consumes all the time without obtaining a relaxed solution, the algorithm stops.
3. Then, a fraction of time $t_{Available}$ is spent solving the first restricted problem *MILP(K)*. The time limit is set equal to $t_{Available} \,/\, (1 + N_b)$, where the number of buckets N_b is initially computed dividing the number of non-kernel variables by the size of the inizial kernel: that coincides with the number of subproblems to be solved without executing the *Adaptation* phase.
4. If the given instance requires the *Feasibility* step, then for each restricted problem to be solved in Algorithm 3 the time limit is set as twice the time allotted for the first *MILP(K)*, or if that is exceeds the overall computing time t_{Max}, it is set to $t_{Available}$.
5. Time thresholds of the *Adaptation* phase are set as follows: for EASY instances, t_{Easy} can be either set to a fixed value, *e.g.* 10 seconds, or to a fraction of total computing time, *e.g.* $t_{Max}/1800$, and it coincides with the amount of time allocated to each *MILP(K)* in Algorithm 4. Instead, for any HARD instance t_{Hard} is set equal to the time limit that was allocated to the most recent solved *MILP(K)*: in this way, if Gurobi cannot obtain an optimal solution, but only a feasible one, for the last restricted problem within the imposed time limit, the heuristic classifies such instance as HARD, properly reducing the dimension of the problem. This implies that Kernel Search automatically assesses the difficulty of a MILP problem depending on $t_{Available}$. For NORMAL instances, the *Adaptation* phase simply does not take place.
6. Lastly, the remaining time $t_{Available}$ is equally distributed for each restricted problem in the *Extension* phase. If a subproblem does not consume all of its allotted time, that unused time is distributed among the successive restricted problems.

If the basic version of Kernel Search is performed, steps 4 and 5 are not taken into account. A possible improvement of the *Extension* phase related to our time allocation policy is to extend each constraint 3.1 added to restricted problems in the following way, if $\nu^* < +\infty$:

$$\sum_{j \in B_i \cup K_0} x_j \geq 1 \tag{3.6}$$

where $K_0 = \{j \in K \,:\, x_j^* = 0\}$, that corresponds to the set of unused kernel variables in the current best found solution vector. This variation allows Adaptive Kernel Search to find an improving solution in which every variable in the current bucket is equal to zero: if *MILP(*$K \cup B_{i-1}$*)* has not been solved to optimality within its assigned time limit, it can happen that a better solution is found by solving *MILP(*$K \cup B_i$*)*, without considering bucket variables.

It is easy to see that our time allocation policy favors the achievement of an initial feasible solution. In fact, tasks in points 2 and 4 are allowed to consume all the available time, because it is essential to either obtain LP feasibility and to consequently search for a possible MILP feasible solution: the purpose of our heuristic is indeed to attack very complex problems, initially aiming to obtain a suboptimal solution within a reasonable computing time. If a problem proves to be easily solvable, the Adaptive Kernel Search accordingly reduces the optimality gap of the current solution, whereas it shrinks the dimension of hard but feasible problems, resulting in a very flexible and efficient heuristic.

Chapter 4

Experimental analysis

In this chapter we discuss in detail the case study and our computational experiments. Firstly, we describe the testing environment used to compare different implementations of the Kernel Search heuristic framework and the Brand and Bound exact algorithm on a broad collection of MILPs, discussing obtained results. We also expose the structure of our testing environment and experimental plan, discussing benchmark and measures of quality. Models are then applied to our case study discussed in Section 2.3, that represents a very challenging test for our heuristic framework and the exact solving method. We then discuss a second set of experiments related to the case study. Every experimental test was conducted in the following testing environment: HP Probook G7-440 personal computer, equipped with a Intel Core i5-10210U with a base frequency of 2.11 GHz processor, 16 gigabytes of RAM and Windows 10 64-bit as operating system. The implementation of every algorithm was written in Python, and the corresponding Python API of Gurobi version 9.1.1. was used to solve every optimization problem. Settings of the commercial solver Gurobi were set to default values. During computational experiments, the computing time was traced within the implementation code, exploiting the Python-based performance counter *perf_counter()*, which includes time elapsed during sleep and is system-wide. Different values for the overall available computing time t_{Max} were tested, and different time allocation settings for the Adaptive Kernel Search as well.

4.1 Comparison between different frameworks

In order to test and compare different models, we applied them on the Benchmark Set of *The Mixed Integer Programming Library* (MIPLIB) provided by Gleixner [2021], that represents a standard test set used to compare the performance of mixed integer solving algorithms. This problem set contains 240 MILP problems having different dimension and complexity, of which 8 are not feasible. In the following table we describe the dimension of some problems contained in the aforementioned benchmark set, specifying the number of binary, integer and continuous variables, and the number of constraints:

Table 4.1: Description of the first 50 MIPLIB problems.

Instance name	**Binaries**	**Integers**	**Continuous**	**Constraints**
30n20b8	18318	62	0	576
50v-10	1464	183	366	233
academictimetablesmall	28926	0	0	23294
air05	7195	0	0	426
app1-1	1225	0	1255	4926

Table 4.1: Description of the first 50 MIPLIB problems.

Instance name	Binaries	Integers	Continuous	Constraints
app1-2	13300	0	13571	53467
assign1-5-8	130	0	26	161
atlanta-ip	46667	106	1965	21732
b1c1s1	288	0	3584	3904
bab2	147912	0	0	17245
bab6	114240	0	0	29904
beasleyC3	1250	0	1250	1750
binkar10_1	170	0	2128	1026
blp-ar98	15806	0	215	1128
blp-ic98	13550	0	90	717
bnatt400	3600	0	0	5614
bnatt500	4500	0	0	7029
bppc4-08	1454	0	2	111
brazil3	23874	94	0	14646
buildingenergy	0	26287	128691	277594
cbs-cta	2467	0	22326	10112
chromaticindex1024-7	73728	0	0	67583
chromaticindex512-7	36864	0	0	33791
cmflsp50-24-8-8	1392	0	15000	3520
CMS750_4	7196	0	4501	16381
co-100	48417	0	0	2187
cod105	1024	0	0	1024
comp07-2idx	17155	109	0	21235
comp21-2idx	10792	71	0	14038
cost266-UUE	171	0	3990	1446
cryptanalysiskb128n5obj14	47830	1120	0	98021
cryptanalysiskb128n5obj16	47830	1120	0	98021
csched007	1457	0	301	351
csched008	1284	0	252	351
cvs16r128-89	3472	0	0	4633
dano3_3	69	0	13804	3202
dano3_5	115	0	13758	3202
decomp2	14387	0	0	10765
drayage-100-23	11025	0	65	4630
drayage-25-23	11025	0	65	4630
dws008-01	6608	0	4488	6064
eil33-2	4516	0	0	32
eilA101-2	65832	0	0	100
enlight_hard	100	100	0	100
ex10	0	17680	0	69608
ex9	0	10404	0	40962
exp-1-500-5-5	250	0	740	550
fast0507	63009	0	0	507
fastxgemm-n2r6s0t2	48	0	736	5998
fhnw-binpack4-4	481	0	39	620

The following experimental plan involves the comparison between Kernel Search heuristibc framework and pure Branch and Bound exact algorithm on the aforementioned well diversified collection of MILP problems. Parameters of the former were set to values reported in Table 4.2. Those values are more oriented towards high-quality solutions, and they ensures that every bucket is explored during the *Extension* phase. The length of each bucket coincides with the size of the initial kernel, and kernel enlargements during *Feasibility* step and *Adaptation* phase were set respectively to the 30% and 35% of the initial kernel size, whereas the threshold ϵ was set to 10 times the default integrality tolerance of Gurobi: an integrality restriction on a variable is considered satisfied when the variable's value is less than the aforementioned threshold from the nearest integer value, in the same way the Adaptive Kernel Search operates in case of HARD

instances. Pure Branch and Bound algorithm was applied directly by using Gurobi, disabling all of its presolving methods and internal heuristics through the Gurobi parameters setting showed in Table 4.3.

L_b	$\overline{N}_b$	w	q	ϵ
$\|K\|$	$+\infty$	0.30	0.35	10^{-5}

Table 4.2: (Adaptive) Kernel Search parameters setting.

Cuts	0
Heuristics	0
RINS	0
Presolve	0
Aggregate	0
Symmetry	0
Disconnected	0

Table 4.3: Gurobi parameters setting for pure Branch and Bound algorithm.

Results obtained on the Benchmark set during the experimental comparison between Adaptive Kernel Search and Branch and Bound with $t_{Max} = 60$ are exposed in the Appendix B, where we report the following quantities through complete tables:

- the total available time for both algorithms t_{Max}
- the name of the specific instance
- the value of the best solution found by Adaptive Kernel Search z^{AKS}
- the value of the best solution found by Branch and Bound z^{BB}
- the elapsed computing time for both methods CPU^{AKS} and CPU^{BB} (seconds)
- the difficulty of the instance assessed during the *Adaptation* phase
- the gap in percentage between the best solution found by Kernel Search and the optimal one Gap^{AKS}
- the gap in percentage between the best solution found by Branch and Bound and the optimal one Gap^{BB}

where each optimality *Gap* percentage is computed as $100 \times \frac{z^* - z}{|z^*|}$, where z^* is the optimal solution reported by MIPLIB and z is the found solution of a given method. If a given instance is infeasible, *Gap* is reported as *N/A*. Difficulty tags are the following ones: E if the instance is classified as Easy, N as Normal and H as Hard. If the Adaptive Kernel Search fails to obtain any feasible solution, the reported tag is U (unsolved).

If one method fails to find a feasible solution within the time limit, the best found solution reports the wording *Inf* and the *Gap* is reported through the wording *Fails*, whereas AKS computing time is set to *N/A*. Whether $Gap^{AKS} < Gap^{BB}$ or Adaptive Kernel Search manages to find a solution while Branch and Bound fails, the percentage is reported in **bold**. If $Gap^{AKS} = Gap^{BB}$, the former percentage is reported in **bold** if AKS solution was found in lesser time.

Tables in Appendix B are only related to the first experiment, for sake of simplicity and elegance. For different experimental settings we report only relevant results: the number of solved instance over 240 problems, the average optimality gap and the worst case optimality gap. In order to avoid presenting misleading results, we also computed the *clean* avegare optimality gap, by removing every percentage that exceeds the value 300% from acquired data. The following Table 4.4 summarizes the performance we measured from distinct algorithms and for different values of t_{Max}.

Algorithm	t_{Max} *(s)*	*Solved instances*	*Average gap*	*Worst case gap*	*Clean avg. gap*
Branch and Bound	60	145	692.77%	96190.48%	12.81%
Kernel Search	60	105	144.73%	8900.0%	14.89%
AKS with $t_{Easy} = 10$	60	127	104.48%	8900.0%	8.26%
AKS with $t_{Easy} = 0.03$	60	127	97.61%	8900.0%	12.46%
Branch and Bound	600	189	2939.44%	553400.0%	4.17%
Kernel Search	600	142	106.77%	8900.0%	11.25%
AKS with $t_{Easy} = 10$	600	173	68.45%	8900.0%	3.49%
AKS with $t_{Easy} = 0.33$	600	171	59.44%	8900.0%	5.08%

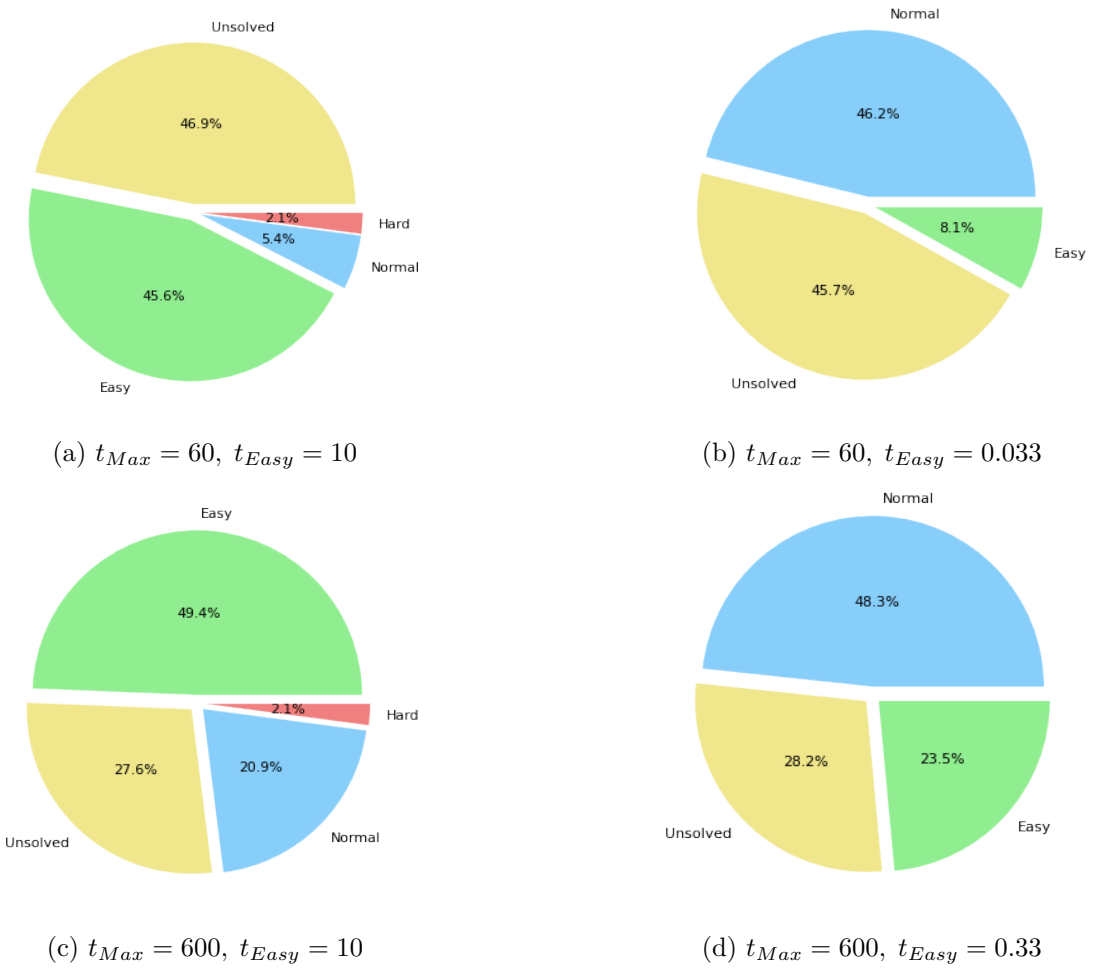
Table 4.4: Kernel Search *versus* Branch and Bound experimental results.

The outcome is significantly interesting: in general, pure Branch and Bound method has proven to be capable of solving more instances, at the cost of scoring some extremely high optimality gaps in the worst cases. This means that the algorithm is still very strong at attacking very complex MILPs obtaining feasible solutions, but with a tight bound on computing time it struggles to find good quality ones. Instead, Kernel Search framework has proven to obtain a much lower average optimality gap, but leaving a relevant amount of problems unsolved: chosen parameters in Table 4.2 lead the heuristic to chase high quality solution, at the expense of loosing feasibility in the worst cases. Another notable result is that the Adaptive version of Kernel Search clearly obtained much better scores than its basic implementation: being able to fit itself to instances of different difficulty proves to be a remarkably effective feature. It's interesting to notice how distinct values of t_{Easy} affect the heuristic results: a proportionally large value causes the algorithm to score a slightly better clean average gap, meanwhile the overall mean gap is a bit higher. This means that kernel enlargements can actually favors the heuristic in the worst cases, making the algorithm more robust. In Figure 4.1 we describe how difficulties assessed by Adaptive Kernel Search are spread over the Benchmark set, depending on time allocation policy and t_{Max}: higher values of t_{Easy} correspond to a larger portion of EASY problems as expected, while a less strict time limit can significantly reduce the portion of unsolved problems.

Removing outlying data values makes the analysis clearer: clean average optimality gaps show how AKS and Branch and Bound obtained comparable performances, whereas the basic implementation of Kernel Search falls behind, especially when the total available time becomes larger. We remark that reported values of t_{Max} are very strict: this means that Kernel Search framework can be extremely useful in situations with a limited amount of available computing time, *e.g.* for the stochastic facility location in a humanitarian disaster setting discussed in Turkeš et al. [2021], so we consider satisfied of obtained performance. More in detail, we report in Table 4.5 the number of times, for each implementation of the Kernel Search heuristic framework, a model scored a better and a worse optimality gap than Branch and Bound, and the number of times a model obtained the same result in lesser and in more time than the exact algorithm.

Algorithm	t_{Max} *(s)*	*# better gap*	*# better time*	*# worse gap*	*# worse time*
Kernel Search	60	42	13	97	11
AKS with $t_{Easy} = 10$	60	57	8	77	27
AKS with $t_{Easy} = 0.03$	60	51	24	82	14
Kernel Search	600	44	23	115	10
AKS with $t_{Easy} = 10$	600	61	15	71	37
AKS with $t_{Easy} = 0.33$	600	60	29	78	24

Table 4.5: Kernel Search *versus* Branch and Bound performance comparison.



(a) $t_{Max} = 60,\ t_{Easy} = 10$

(b) $t_{Max} = 60,\ t_{Easy} = 0.033$

(c) $t_{Max} = 600,\ t_{Easy} = 10$

(d) $t_{Max} = 600,\ t_{Easy} = 0.33$

Figure 4.1: Pie charts representing different problem difficulty assessment policies.

It is one more time evident how the basic version of Kernel Search performed significantly worse than its refined variation. The above comparison reveals how setting t_{Easy} proportionally to t_{Max} can notably improve heuristic performance: an improper kernel enlargement can actually slow down the entire solving process, obtaining worse results. The comparison reveals that the exact Branch and Bound method is still hard to surpass, and that the Kernel Search framework proves to be a valid tool to tackle generic MILP problems, obtaining better results if we consider average cases, especially when there is lack of computational resources.

4.2 Case study: experiments and results

The following experimental plan refers to the chosen case study. The strategy is to explore and test different parameters of the Adaptive Kernel Search heuristic in order to find the most suitable ones for SCLSP of different dimensions and number of time periods: our goal is to obtain a set of good performing parameters for each different dimension of the MILP problem, in order to suitably shape various AKS versions depending on the problem complexity, and to better understand their impact on performance. Experiments were conducted on a dataset of SCLSP models of various dimensions, generated through Monte Carlo simulations. The dataset was split into a *train set* and a *test set* in proportion 70/30: the best performing parameters on the former are then evaluated on the latter. Since Branch and Bound does not need a proper parameters tuning, we performed an *in-sample* comparison between different models and algorithms. The testing environment is the same of the previous experimental plan, while Monte Carlo simulations and scenarios generation were implemented in MATLAB.

Dataset contains various instances of the Stochastic Capacitated Lot-Sizing Problem, each one named through its branching structure, as reported in Table 4.6. Larger numbers of time periods were not considered, since they made instances too difficult to solve in less than 3 hours. A first collection of runs of Adaptive Kernel Search on the specified instances with t_{Max} equal to 3600 seconds immediately revealed their notable complexity, since bigger problems were not solved to optimality: thus, the amount of t_{Max} allocated for each of those instances was set to 7200 seconds. Adaptive Kernel Search parameters w, q and *lbuck* were tuned for each bucket of time periods as exposed in Table 4.7, while t_{Easy} was set to $t_{Max}/1800$, since it performed well during the previous comparison. Results are exposed in Table 4.6 with the same notation of tables in Appendix B, expect for the optimality gap.

Instance name	z^{AKS}	z^{BB}	CPU^{AKS}	CPU^{BB}	*Difficulty*
1-4-1-1	16960.59	16960.59	**4.70**	3600.12	E
1-5-1-1	21199.17	21199.17	**2.66**	3600.09	E
1-6-1-1	19848.5	19848.5	**5.16**	3600.42	E
1-7-1-1	18005.49	18005.49	**8.64**	3600.03	E
1-8-1-1	21180.85	21180.85	**16.28**	3601.11	N
1-9-1-1	21611.49	21611.49	**23.2**	3600.08	N
1-2-2-1-1	28367.65	28367.65	**5.03**	7200.57	E
1-2-3-1-1	28719.94	28556.28	6.21	7200.09	N
1-3-4-1-1	**24780.68**	24949.5	**850.92**	7200.29	N
1-5-2-1-1	27753.94	27753.94	**553.09**	7201.03	N
1-6-2-1-1	**27442.01**	27533.65	**441.74**	7200.12	N
1-3-3-1-1	**29148.16**	29337.51	**16.2**	7200.38	N
1-4-4-1-1	**27639.14**	27862.18	**786.55**	7200.41	N
1-3-2-2-1-1	**31880.42**	32163.96	**39.95**	7200.45	N
1-3-3-2-1-1	**32385.19**	36498.76	**1766.78**	7200.95	N
1-4-2-2-1-1	*Inf*	31864.44	*N/A*	7200.79	U
1-4-3-2-1-1	*Inf*	36276.97	*N/A*	7200.28	U
1-4-4-2-1-1	*Inf*	33645.16	*N/A*	7200.11	U
1-3-2-2-2-1-1	*Inf*	*N/A*	*Inf*	*N/A*	U
1-3-3-2-2-1-1	*Inf*	36498.76	*N/A*	7200.56	U
1-4-4-2-2-1-1	*Inf*	*N/A*	*Inf*	*N/A*	U
1-4-3-3-2-1-1	*Inf*	27752.59	*N/A*	7201.13	U

Table 4.6: Experimental results on Monte Carlo-simulated SCLSP instances.

Time periods	q	w	*lbuck*
4	0.75	0.50	$\|K\|$
5	0.30	0.15	$0.75 * \|K\|$
6	0.30	0.05	$0.5 * \|K\|$
7	0.30	0.01	$0.25 * \|K\|$

Table 4.7: Adaptive Kernel Search best performing parameters for SCLSP.

Almost every instance with 4 time periods was classified as EASY, making q the most important parameter, that is the percentage of enlargement of the kernel size during the *Adaptive* step: larger values implied a slight but notable reduction in computing time, since the algorithm was able to reduce the amount of time spent during the searching phase. The length of buckets, in those cases, is useless, since the remaining non-kernel variables are very few. Every 4-periods instance was solved by AKS to optimality and in less than one minute, whereas Branch and Bound required almost the entire available time. Due to the lower dimension of 4-periods instances, both methods achieved the same optimal solution as expected.

Instances with 5 periods were mostly classified as NORMAL, making the q parameter irrelevant. Instead, the parameter w that regulates the *Feasibility* step has proven to be the most sensitive one, being very impacting on the achievement of a feasible solution: larger values of w made each restricted problem in Algorithm 4 more difficult to solve within its time limit, causing a decrease in computing performance of AKS. In fact, lowering the value of w remarkably increased the needed computing time. The length of each bucket remained an unimportant parameter, due to the number of non-kernel variables. AKS managed to solve every 5-period instance within an order or even two less of magnitude of computing time than the exact algorithm, sometimes obtaining a better objective value: in case of simpler problems and very strict time limits, Kernel Search heuristic outshines Branch and Bound, confirming our analysis in the previous section.

All the instances containing 6 and 7 time periods revealed themselves extremely hard to solve within two hours. In particular, AKS did not managed to solve any of the 7-periods instances, while it did solve some of the 6-periods ones, achieving a better objective value than Branch and Bound, in two order less of magnitude computing time. On the contrary, Branch and Bound algorithm was able to obtain a feasible solution for all the 6-periods instances and some of the 7-periods ones, consuming all the available time: thus, it still represents a reliable and valuable solving method to tackle discrete problems with such complexity, obtaining feasible solutions that can be very distant from the optimal value, as we deduced from results of the first experimental plan.

It is necessary to remark that the chosen case study has no dramatic consequences when a feasible solution is not obtained: in the worst cases, the user will incur in lost sales. Instead, it becomes significantly more risky to fail obtaining a feasible solution when it comes to optimization problems of very delicate applications: for example, in a facilities location problem during a humanitarian disaster, failing to obtain a solution within a rigid time limit may cost in human lives. In such dangerous situations, can be more useful to operate with a heuristic algorithm that can provide a good-quality solution in few seconds, if the optimization problem is not huge.

Chapter 5

Conclusions and future works

This thesis work aimed on studying, implementing and applying heuristic algorithms for the solution of discrete optimization problems. The Kernel Search heuristic framework has proven to be an efficient and effective solving method to tackle Mixed-Integer Linear Programming problems, thanks to a clever problem restriction strategy and the huge improvements of commercial solvers. In particular, the Adaptive version of the heuristic can successfully tackle problems of different dimensions, being more efficient and flexible than its basic version and exact Branch and Bound. Experimental results have been quite positive, proving that Adaptive Kernel Search performs, on average, better than Branch and Bound, although the latter still constitutes a very effective solving method, being able to obtain feasible solutions in the worst cases where AKS failed. The chosen case study has proven to be extremely challenging, since the dimension of generated instances required a large amount of available computing time, but the heuristic managed to find the optimal solution in way less time than Branch and Bound when facing easier problems.

Our results suggest that the Kernel Search framework is remarkably well-performing when time limits are very strict, *e.g.* when optimization models are related to emergency problems: in such cases, obtaining a good solution within a time limit of, for example, ten minutes can be game-changing. A possible future work is to experiment a specific variant of the Kernel Search heuristic for humanitarian disasters, taking as example the work of Turkeš et al. [2021].

Another future work is to explore other automatic features of the Adaptive Kernel Search heuristic, thanks to its highly customizable structure. For example, it would be interesting to investigate a more dynamic time allocation policy. Obtained results suggest that the amount of available time for each restricted problem is crucial: a even more adaptive version of the algorithm could decide to allocate a different amount of time depending on the number of variables or constraints. In fact, the latter is not considered by the heuristic, and it may provide useful informations about the problem difficulty. A further adaptive feature can be the automatic setting of the most sensitive parameters, like w and q, exploiting the assessed difficulty of the given instance.

Appendix A

Kernel Search implementation

Here we present the Python code of the implementation of our Kernel Search version. The entire code listing is divided into three sections, one for each phase of the heuristic.

A.1 *Initialization* phase code

```
import numpy as np
import pandas as pd
import math
import time
import sys
import more_itertools as mit
import gurobipy as gb

# Functions
def prepareModel(model, var, kernel, bucket=None, unused=None):
    varRemove = set(var).difference(set(kernel))

    # Restrict model variables to kernel
    if len(varRemove) > 0:
        maxVar = model.addVar(ub=0.0, name="maxVar")
        model.addGenConstrMax(maxVar, varRemove, name="maxConstr")
        model.update()

    # Consider at least one variable from bucket
    if bucket is not None and unused is None:
        orVar = model.addVar(lb=1.0, name="orVar")
        model.addGenConstrOr(orVar, bucket, name="orConstr")
        model.update()

   # Consider at least one variable from bucket and unused variables
   if bucket is not None and unused is not None:
        enlarged_bucket = bucket + unused
        orVar = model.addVar(lb=1.0, name="orVar")
        model.addGenConstrOr(orVar, enlarged_bucket, name="orConstr")
        model.update()


def resetModel(model, var, kernel, bucket=None):
    varRemove = set(var).difference(set(kernel))

    # Remove previously added constraints
    if bucket is not None:
        model.remove(model.getVarByName("orVar"))
        model.remove(model.getGenConstrs()[-1])
```

```
        model.update()

    if len(varRemove) > 0:
        model.remove(model.getVarByName("maxVar"))
        model.remove(model.getGenConstrs()[-1])
        model.update()

# Parameters
NBmax = 20
Tmax = 600
Teasy = Tmax/1800
q = 0.30
w = 0.35
eps = 10^-5
file = "MPS/file.mps"

# Read model
model = gb.read(file)
model.Params.OutputFlag = 0
N = model.NumVars
m = model.NumConstrs
var = model.getVars()
index = dict(zip(var, range(0, N)))

# LP-relaxation
relaxed = model.relax()
relaxed.optimize()
Tavailable = max(0, Tmax - relaxed.Runtime)
if relaxed.Status == 3:
    sys.exit("LP-relaxation is infeasible!")

# Kernel and buckets initialization
kernel = [v for (x, v) in zip(relaxed.X, var) if x>0 or v.VType=='C']
C = len(kernel)

# Sorting reduced costs of LP-relaxation
reduced_cost = relaxed.RC
for v in kernel:
    reduced_cost[index[v]] = math.inf
reduced_cost = np.argsort(reduced_cost)
pre_bucket = [var[i] for i in reduced_cost[:N-C]]
```

A.2 *Adaptation* phase code

```
print("Getting feasible solution for MILP(K)...")
hardness = 'N'
get_feasible = round(w * C)
first_time = Tavailable/(1+math.ceil(len(pre_bucket)/C))
failure = 1
flag = 0

# Searching for a feasible solution of MILP
while failure == 1:

    # Assess time limit
    if flag == 0:
        model.Params.timeLimit = first_time
        flag = 1
    elif first_time * 2 < Tavailable:
        model.Params.timeLimit = first_time * 2
    else:
        model.Params.timeLimit = Tavailable
    prepareModel(model, var, kernel)
```

```
    model.optimize()
    Tavailable = max(0, Tavailable - model.Runtime)

    # Check solution
    if model.Status == 2 or model.Status == 13:
        best_sol = model.X
        best_objval = model.ObjVal
        model.Params.Cutoff = best_objval
        failure = 0

        # Recognize hard instances
        if model.Runtime >= model.Params.timeLimit and \
            model.Status == 13:
            hardness = 'H'

        resetModel(model, var, kernel)

        if len(kernel) == len(var):
            print("Solution found: ", best_objval)
            print("Elapsed time: ", round(time.perf_counter()))
            sys.exit("Kernel contains all variables.")

    # Enlarging kernel if not feasible
    else:
        resetModel(model, var, kernel)
        kernel.extend(pre_bucket[:get_feasible])
        pre_bucket = pre_bucket[get_feasible:]

    # Check remaining time
    if Tavailable == 0:
        sys.exit("Timeout: no feasible solution found.")

# Perform adaptive step
print("Performing adaptive step...")

# Hard instance: fix binary and integer variables to nearest integer
if hardness == 'H':

    # Integer variables
    for var in [v for (x, v) in zip(relaxed.X, var) \
                if (int(round(x))-eps) <= x <= (int(round(x))+eps) \
                and v.VType == 'I' and v in pre_bucket]:
        model.addConstr(var == int(round(relaxed.X[index[var]])))
        model.update()

    # Binary variables
    for var in [v for (x, v) in zip(relaxed.X, var) \
                if x > 1 - eps and v.VType == 'B' and v in pre_bucket]:
        model.addConstr(var == 1)
        model.update()

# Easy instance
instance_easy = round(q * C)

while model.Runtime <= Teasy and hardness != 'H':
    hardness = 'E'

    bucket = pre_bucket[:instance_easy]
    kernel.extend(bucket)
    pre_bucket = pre_bucket[instance_easy:]

    prepareModel(model, var, kernel, bucket)
    model.Params.timeLimit = Teasy
    model.optimize()

    Tavailable = max(0, Tavailable - model.Runtime)
```

```
    # Check solution
    if model.Status == 2 or model.Status == 13:
        best_sol = model.X
        best_objval = model.ObjVal
        model.Params.Cutoff = best_objval
        if len(kernel) == len(var):
            print("Solution found: ", best_objval)
            print("Elapsed time: ", round(time.perf_counter()))
            sys.exit("Kernel contains all variables.")

    resetModel(model, var, kernel, bucket)

    # Check remaining time
    if Tavailable == 0:
        print("Solution found: ", best_objval)
        print("Elapsed time: ", round(time.perf_counter()))
        sys.exit("Timeout: feasible solution found.")
```

A.3 *Extension* phase code

```
# Generate buckets
lbuck = C
buckets = list(mit.chunked(pre_bucket, lbuck))
NB = len(buckets)

# Iterate over buckets
print("Iterating over buckets...")
for i in range(0, min(NB, NBmax)):

    new_kernel = kernel + buckets[i]

    # Force also unused kernel variables, may improve current solution
    if best_sol is not None:
        unused = [v for v in kernel if best_sol[index[v]] == 0]
        prepareModel(model, var, new_kernel, buckets[i], unused)
    else:
        prepareModel(model, var, new_kernel, buckets[i])

    model.Params.timeLimit = Tavailable / (NB - i)
    model.optimize()
    Tavailable = max(0, Tavailable - model.Runtime)

    # Check solution
    if model.Status == 2 or model.Status == 13:
        best_sol = model.X
        best_objval = model.ObjVal
        model.Params.Cutoff = best_objval
        kernel.extend([v for v in buckets[i] if best_sol[index[v]] > 0])

    resetModel(model, var, new_kernel, buckets[i])

    # Check remaining time
    if (Tavailable == 0):
        print("Solution found: ", best_objval)
        print("Elapsed time: ", round(time.perf_counter()))
        sys.exit("Timeout: feasible solution found.")

# Ending of Kernel Search
kernel_time = round(time.perf_counter())
print("Best value obtained (KS):", best_objval)
print("Elapsed time (KS):", kernel_time)
```

Appendix B

Experimental results

In this appendix we report all the entire results obtained during computational experiments.

B.1 Adaptive Kernel Search vs Branch and Bound

The following tables contain the computational comparison between Adaptive Kernel Search and pure Brand and Bound algorithm, with $t_{Max} = 60$ and $t_{Easy} = 10$. The AKS parameters were set as in Table 4.2, while Gurobi settings were set to the values specified in Table 4.3, in order to get an almost pure Branch and Bound algorithm.

Table B.1: Experimental results with $t_{Max} = 60$ and $t_{Easy} = 10$ seconds (1)

Instance name	z^{AKS}	CPU^{AKS}	z^{BB}	CPU^{BB}	*Difficulty*	Gap^{AKS}	Gap^{BB}
30n20b8	302	36,48	302	56,21	E	**0.0%**	0.0%
50v-10	4618,18	60	3475,89	60,01	E	39.47%	4.97%
academictimetablesmall	Inf	N/A	Inf	60,03	U	Fails	Fails
air05	26374	60	26374	10,83	E	0.0%	0.0%
app1-1	-3	4,82	-3	3,2	E	0.0%	0.0%
app1-2	Inf	N/A	Inf	60,05	U	Fails	Fails
assign1-5-8	Inf	N/A	212	60	U	Fails	0.0%
atlanta-ip	94,01	57,06	Inf	60,04	E	**4.44%**	Fails
b1c1s1	Inf	N/A	26208,82	60,04	U	Fails	6.78%
bab2	Inf	N/A	Inf	62,41	U	Fails	Fails
bab6	Inf	N/A	Inf	60,07	U	Fails	Fails
beasleyC3	754	8,57	829	60,03	E	**0.0%**	9.95%
binkar10_1	6742,2	5,69	6746,76	60,03	E	**0.0%**	0.07%
blp-ar98	6361,5	60	6592,55	60,04	E	**2.52%**	6.24%
blp-ic98	4508,9	60	4942,96	60,03	E	**0.39%**	10.05%
bnatt400	Inf	N/A	Inf	60,03	U	Fails	Fails
bnatt500	Inf	N/A	Inf	60,03	U	N/A	N/A
bppc4-08	57	60	55	60	E	7.55%	3.77%
brazil3	Inf	N/A	Inf	60,01	U	Fails	Fails
buildingenergy	Inf	N/A	Inf	60,11	U	Fails	Fails
cbs-cta	0	0,88	0	0,78	E	0.0%	0.0%
chromaticindex1024-7	4	16,11	Inf	60,01	E	**0.0%**	Fails
chromaticindex512-7	4	13,18	Inf	60,07	E	**0.0%**	Fails
cmflsp50-24-8-8	Inf	N/A	Inf	60,03	U	Fails	Fails
CMS750_4	252	19,15	252	7,52	N	0.0%	0.0%
co-100	5962512,73	60	6064254,52	60,49	E	**125.86%**	129.71%
cod105	Inf	N/A	Inf	60,06	U	Fails	Fails
comp07-2idx	Inf	N/A	13	60,04	U	Fails	116.67%
comp21-2idx	Inf	N/A	121	60,04	U	Fails	63.51%
cost266-UUE	Inf	N/A	25280712,46	60,04	U	Fails	0.52%
cryptanalysiskb128n5obj14	Inf	N/A	Inf	60,02	U	N/A	N/A
cryptanalysiskb128n5obj16	Inf	N/A	Inf	60	U	Fails	Fails
csched007	Inf	N/A	374	60,01	U	Fails	6.55%
csched008	173	43,72	174	60,03	E	**0.0%**	0.58%
cvs16r128-89	Inf	N/A	-92	60	U	Fails	5.15%
dano3_3	576,34	29,65	576,34	36,78	N	**0.0%**	0.0%
dano3_5	Inf	N/A	576,99	60,01	U	Fails	0.01%
decomp2	-160	60	-160	2,66	E	0.0%	0.0%
drayage-100-23	103333,87	60	103333,87	1,01	E	0.0%	0.0%
drayage-25-23	101282,65	60	101282,65	60,04	E	0.0%	0.0%
dws008-01	Inf	N/A	41559,42	60,04	U	Fails	11.08%
eil33-2	Inf	N/A	934,01	3,95	U	Fails	0.0%
eilA101-2	Inf	N/A	Inf	60,09	U	Fails	Fails
enlight_hard	37	60	37	2,06	E	0.0%	0.0%
ex10	Inf	N/A	Inf	60,06	U	Fails	Fails
ex9	81	60	Inf	60,02	E	**0.0%**	Fails
exp-1-500-5-5	65887	11,29	68671	60,01	E	**0.0%**	4.23%
fast0507	Inf	N/A	Inf	60,03	U	Fails	Fails
fastxgemm-n2r6s0t2	2327	11,63	527	60,04	E	911.74%	129.13%
fhnw-binpack4-4	Inf	N/A	Inf	60	U	N/A	N/A
fhnw-binpack4-48	0	40,51	0	7,52	N	0.0%	0.0%
fiball	138	60	Inf	60,03	E	**0.0%**	Fails

Table B.2: Experimental results with $t_{Max} = 60$ and $t_{Easy} = 10$ seconds (2)

Instance name	z^{AKS}	CPU^{AKS}	z^{BB}	CPU^{BB}	*Difficulty*	Gap^{AKS}	Gap^{BB}
gen-ip002	-4762,94	60	-4772,4	60	E	0.43%	0.24%
gen-ip054	6840,97	21,85	6847,25	60,01	E	**0.0%**	0.09%
germanrr	Inf	N/A	Inf	60,02	U	Fails	Fails
gfd-schedulen180f7d50m30k18	Inf	N/A	Inf	60,04	U	Fails	Fails
glass-sc	Inf	N/A	23	60,03	U	Fails	0.0%
glass4	1200012600	35,61	1600014267	60	E	**0.0%**	33.33%
gmu-35-40	-2406613,22	28,2	-2406157,69	60,01	E	**0.0%**	0.02%
gmu-35-50	-2607797,05	39,62	-2605748,52	60,01	E	**0.01%**	0.08%
graph20-20-1rand	-7	60	-8	60,02	E	22.22%	11.11%
graphdraw-domain	24141	60	19857	60,01	E	22.63%	0.87%
h80x6320d	6437,53	52,78	7343,01	60,05	E	**0.87%**	15.06%
highschool1-aigio	Inf	N/A	Inf	60,06	U	Fails	Fails
hypothyroid-k1	-2851	27,74	-2851	29,74	N	**0.0%**	0.0%
ic97_potential	Inf	N/A	3953	60,02	U	Fails	0.28%
icir97_tension	6376	60	6386	60,01	N	**0.02%**	0.17%
irish-electricity	Inf	N/A	Inf	60,08	U	Fails	Fails
irp	Inf	N/A	12159,49	0,81	U	Fails	0.0%
istanbul-no-cutoff	Inf	N/A	204,08	60,04	U	Fails	0.0%
k1mushroom	Inf	N/A	Inf	60,07	U	Fails	Fails
lectsched-5-obj	Inf	N/A	39	60,06	U	Fails	62.5%
leo1	411539933,3	60	471859930,1	60,02	E	**1.81%**	16.73%
leo2	411520245,6	60	Inf	60,05	E	**1.84%**	Fails
lotsize	Inf	N/A	1760543	60,01	U	Fails	18.94%
mad	0,34	60	0,09	60,01	E	1033.33%	200.0%
map10	Inf	N/A	Inf	60,12	U	Fails	Fails
map16715-04	Inf	N/A	Inf	60,05	U	Fails	Fails
markshare2	90	60	22	60,01	E	8900.0%	2100.0%
markshare_4_0	1	60	1	7,39	E	0.0%	0.0%
mas74	11877,69	30,07	11801,19	60,01	E	0.65%	0.0%
mas76	40005,05	60	40005,05	12,17	E	0.0%	0.0%
mc11	11702	25,96	12848	60,05	E	**0.11%**	9.92%
mcsched	214786	16,67	211950	60,02	E	1.36%	0.02%
mik-250-20-75-4	-43679	60	-52287	60,02	E	16.49%	0.03%
milo-v12-6-r2-40-1	328802,32	10,19	Inf	60,02	E	**0.71%**	Fails
momentum1	Inf	N/A	Inf	60,03	U	Fails	Fails
mushroom-best	0,06	26,25	Inf	60,03	E	**0.0%**	Fails
mzzv11	-21608	32,63	-21638	60,04	E	0.51%	0.37%
mzzv42z	-20500	60	-20540	34,88	E	0.19%	0.0%
n2seq36q	52600	60	52200	11,92	E	0.77%	0.0%
n3div36	140800	60	Inf	60,04	E	**7.65%**	Fails
n5-3	8105	5,97	8465	60,03	E	**0.0%**	4.44%
neos-1122047	161	60	161	5,61	E	0.0%	0.0%
neos-1171448	-309	4,54	-309	2,87	E	0.0%	0.0%
neos-1171737	-195	21,38	-195	31,44	E	**0.0%**	0.0%
neos-1354092	Inf	N/A	Inf	60,02	U	Fails	Fails
neos-1445765	0	10,12	-17783	23,75	E	100.0%	0.0%
neos-1456979	177	40,31	177	60,02	E	0.57%	0.57%
neos-1582420	91	60	91	51,55	E	0.0%	0.0%
neos-2075418-temuka	Inf	N/A	Inf	60,29	U	N/A	N/A
neos-2657525-crna	Inf	N/A	Inf	60,01	U	Fails	Fails
neos-2746589-doon	Inf	N/A	Inf	60,11	U	Fails	Fails

Table B.3: Experimental results with $t_{Max} = 60$ and $t_{Easy} = 10$ seconds (3)

Instance name	z^{AKS}	CPU^{AKS}	z^{BB}	CPU^{BB}	*Difficulty*	Gap^{AKS}	Gap^{BB}
neos-2987310-joes	-607702988,3	4,72	-607702988,3	3,61	E	0.0%	0.0%
neos-3004026-krka	Inf	N/A	0	9,28	U	Fails	0.0%
neos-3024952-loue	Inf	N/A	29266	60,04	U	Fails	9.38%
neos-3046615-murg	1600	60	1612	60,01	E	**0.0%**	0.75%
neos-3083819-nubu	6308067	60	6307996	14,99	E	0.0%	0.0%
neos-3216931-puriri	Inf	N/A	Inf	60,03	U	Fails	Fails
neos-3381206-awhea	453	60	Inf	60,01	E	**0.0%**	Fails
neos-3402294-bobin	0,33	60	Inf	60,27	E	**371.43%**	Fails
neos-3402454-bohle	Inf	N/A	Inf	60,21	U	N/A	N/A
neos-3555904-turama	Inf	N/A	Inf	60,04	U	Fails	Fails
neos-3627168-kasai	988585,62	10,43	989725,33	60,02	E	**0.0%**	0.12%
neos-3656078-kumeu	Inf	N/A	Inf	60,04	U	Fails	Fails
neos-3754480-nidda	Inf	N/A	13128,62	60	U	Fails	1.44%
neos-3988577-wolgan	Inf	N/A	Inf	60,07	U	N/A	N/A
neos-4300652-rahue	2,14	60	3,28	60,07	N	**0.0%**	53.27%
neos-4338804-snowy	1720	60	1484	60,07	E	16.93%	0.88%
neos-4387871-tavua	36,56	60	147,95	60,04	E	**9.53%**	343.23%
neos-4413714-turia	48,93	60	Inf	60,14	E	**7.85%**	Fails
neos-4532248-waihi	Inf	N/A	Inf	60,05	U	Fails	Fails
neos-4647030-tutaki	Inf	N/A	Inf	60,25	U	Fails	Fails
neos-4722843-widden	25532,13	16,44	Inf	60,01	E	**2.09%**	Fails
neos-4738912-atrato	383978456,9	10,3	283627956,6	48,35	E	35.38%	0.0%
neos-4763324-toguru	1689,59	60	Inf	60,05	E	**4.75%**	Fails
neos-4954672-berkel	2896853	17,1	2775590	60,02	E	10.88%	6.23%
neos-5049753-cuanza	Inf	N/A	Inf	60,39	U	Fails	Fails
neos-5052403-cygnet	182	60	Inf	60,17	E	**0.0%**	Fails
neos-5093327-huahum	Inf	N/A	6714	60,12	U	Fails	7.25%
neos-5104907-jarama	Inf	N/A	Inf	60,02	U	Fails	Fails
neos-5107597-kakapo	27243	60	3681	60,05	H	647.41%	0.99%
neos-5114902-kasavu	Inf	N/A	Inf	65,1	U	Fails	Fails
neos-5188808-nattai	Inf	N/A	0,12	60,04	U	Fails	9.09%
neos-5195221-niemur	Inf	N/A	0,02	60,06	U	Fails	N/A
neos-631710	289	60	Inf	60,19	E	**42.36%**	Fails
neos-662469	Inf	N/A	264827,5	60,09	U	Fails	43.63%
neos-787933	32	60	Inf	60,16	E	**6.67%**	Fails
neos-827175	112	5,1	112	2,67	E	0.0%	0.0%
neos-848589	Inf	N/A	2264174,31	60,47	U	Fails	96190.48%
neos-860300	3313	51,48	3201	51,92	E	3.5%	0.0%
neos-873061	146,18	60	Inf	60,02	E	**28.61%**	Fails
neos-911970	56,36	60	56,51	60,01	E	**2.92%**	3.2%
neos-933966	318	60	318	6,42	E	0.0%	0.0%
neos-950242	4	58,96	4	60,07	E	**0.0%**	0.0%
neos-957323	-229,77	60	-237,76	14,59	E	3.36%	0.0%
neos-960392	-236	60	-238	11,11	E	0.84%	0.0%
neos17	0,15	60	0,15	15,02	E	0.0%	0.0%
neos5	15	52,97	15	60,01	E	**0.0%**	0.0%
neos8	-3627	60	-3719	5,38	E	2.47%	0.0%
net12	214	41,52	255	60,09	H	**0.0%**	19.16%
netdiversion	Inf	N/A	242	24,64	U	Fails	0.0%
nexp-150-20-8-5	365	60	Inf	60,04	E	**58.01%**	Fails
ns1116954	Inf	N/A	Inf	60,02	U	Fails	Fails

Table B.4: Experimental results with $t_{Max} = 60$ and $t_{Easy} = 10$ seconds (4)

Instance name	z^{AKS}	CPU^{AKS}	z^{BB}	CPU^{BB}	*Difficulty*	Gap^{AKS}	Gap^{BB}
ns1208400	2	60	Inf	60,03	E	**0.0%**	Fails
ns1644855	Inf	N/A	Inf	60,74	U	Fails	Fails
ns1760995	Inf	N/A	Inf	60,05	U	Fails	Fails
ns1830653	Inf	N/A	23622	60,02	U	Fails	14.55%
ns1952667	Inf	N/A	0	19,41	U	Fails	0.0%
nu25-pr12	53905	60	53905	4,02	E	0.0%	0.0%
nursesched-medium-hint03	Inf	N/A	Inf	60,04	U	Fails	Fails
nursesched-sprint02	58	60	58	33,39	E	0.0%	0.0%
nw04	Inf	N/A	16862	7,91	U	Fails	0.0%
opm2-z10-s4	Inf	N/A	Inf	60,02	U	Fails	Fails
p200x1188c	15078	60	17133	60,03	E	**0.0%**	13.63%
peg-solitaire-a3	Inf	N/A	Inf	60,03	U	Fails	Fails
pg	-8674,34	60	-8511,74	60	E	**0.0%**	1.87%
pg5_34	-14339,35	34,71	-14339,35	46,12	H	**0.0%**	0.0%
physicianscheduling3-3	Inf	N/A	Inf	60,1	U	Fails	Fails
physicianscheduling6-2	49324	60	Inf	60,11	E	**0.0%**	Fails
piperout-08	125055	60	125055	6,76	E	0.0%	0.0%
piperout-27	8124	60	8124	37,25	E	0.0%	0.0%
pk1	14	15,61	11	11,01	E	27.27%	0.0%
proteindesign121hz512p9	Inf	N/A	Inf	60,8	U	Fails	Fails
proteindesign122trx11p8	Inf	N/A	Inf	60,09	U	Fails	Fails
qap10	340	60	340	24,01	E	0.0%	0.0%
radiationm18-12-05	Inf	N/A	25049	60,07	U	Fails	42.6%
radiationm40-10-02	155341	48,54	Inf	60,29	H	**0.01%**	Fails
rail01	Inf	N/A	Inf	60,01	U	Fails	Fails
rail02	Inf	N/A	Inf	60,02	U	Fails	Fails
rail507	Inf	N/A	Inf	60,04	U	Fails	Fails
ran14x18-disj-8	3735	60	4029	60,01	N	**0.62%**	8.54%
rd-rplusc-21	Inf	N/A	Inf	60,19	U	Fails	Fails
reblock115	Inf	N/A	-36505377,44	60,02	U	Fails	0.8%
rmatr100-p10	423	13,95	423	9,22	N	0.0%	0.0%
rmatr200-p5	Inf	N/A	Inf	60,05	U	Fails	Fails
rocI-4-11	-6020203	60	-5050303	60,05	H	**0.0%**	16.11%
rocII-5-11	Inf	N/A	-0,63	60,06	U	Fails	90.57%
rococoB10-011000	30095	60	21909	60,05	E	54.74%	12.65%
rococoC10-001000	16008	60	11908	60,03	E	39.69%	3.91%
roi2alpha3n4	-55,17	60	-58,51	60,06	E	12.72%	7.44%
roi5alpha10n8	Inf	N/A	-20,3	60,14	U	Fails	61.2%
roll3000	12890	24,03	14021	60	E	**0.0%**	8.77%
s100	Inf	N/A	Inf	60,03	U	Fails	Fails
s250r10	Inf	N/A	Inf	60,03	U	Fails	Fails
satellites2-40	Inf	N/A	31	60,03	U	Fails	263.16%
satellites2-60-fs	Inf	N/A	Inf	60,05	U	Fails	Fails
savsched1	Inf	N/A	Inf	60,05	U	Fails	Fails
sct2	-230,99	44,66	-229,93	60,01	E	**0.0%**	0.46%
seymour	Inf	N/A	426	60	U	Fails	0.71%
seymour1	410,76	60	410,76	60,01	N	0.0%	0.0%
sing326	7794681,99	60	Inf	60,05	E	**0.53%**	Fails
sing44	8151205,18	60	Inf	61,34	E	**0.28%**	Fails
snp-02-004-104	Inf	N/A	Inf	60,29	U	Fails	Fails
sorrell3	Inf	N/A	-5	60,07	U	Fails	68.75%

Table B.5: Experimental results with $t_{Max} = 60$ and $t_{Easy} = 10$ seconds (5)

Instance name	z^{AKS}	CPU^{AKS}	z^{BB}	CPU^{BB}	*Difficulty*	Gap^{AKS}	Gap^{BB}
sp150x300d	69	60	69	60	E	0.0%	0.0%
sp97ar	Inf	N/A	Inf	60,02	U	Fails	Fails
sp98ar	534807380,8	54,51	Inf	60,05	E	**0.96%**	Fails
splice1k1	Inf	N/A	0	60,12	U	Fails	100.0%
square41	Inf	N/A	Inf	60,34	U	Fails	Fails
square47	Inf	N/A	Inf	60,61	U	Fails	Fails
supportcase10	Inf	N/A	Inf	60,02	U	Fails	Fails
supportcase12	Inf	N/A	Inf	60,24	U	Fails	Fails
supportcase18	51	60	51	60,04	E	6.25%	6.25%
supportcase19	Inf	N/A	Inf	60,08	U	Fails	Fails
supportcase22	Inf	N/A	Inf	60,15	U	N/A	N/A
supportcase26	Inf	N/A	1768,26	60,02	U	Fails	1.33%
supportcase33	Inf	N/A	-315	60,05	U	Fails	8.7%
supportcase40	25539,18	15,48	24735,96	60,06	E	5.29%	1.98%
supportcase42	8	20,09	8	60,07	N	3.09%	3.09%
supportcase6	Inf	N/A	51906,48	47,8	U	Fails	0.0%
supportcase7	-1132,22	60	-1132,22	42,23	E	0.0%	0.0%
swath1	379,07	60	379,07	7,37	E	0.0%	0.0%
swath3	397,76	17,84	397,76	49,08	N	**0.0%**	0.0%
tbfp-network	Inf	N/A	Inf	60,13	U	Fails	Fails
thor50dday	204179	60	Inf	60,01	E	**405.18%**	Fails
timtab1	Inf	N/A	816413	60	U	Fails	6.75%
tr12-30	Inf	N/A	139929	60,04	U	Fails	7.15%
traininstance2	Inf	N/A	Inf	60,04	U	Fails	Fails
traininstance6	28290	60	30850	60,02	E	**0.0%**	9.05%
trento1	17936029,04	60	Inf	60,03	N	**245.62%**	Fails
triptim1	22,87	60	22,87	41,14	E	0.0%	0.0%
uccase12	12339,49	16,91	11507,76	41,65	E	7.23%	0.0%
uccase9	Inf	N/A	12076,28	61,03	U	Fails	9.85%
uct-subprob	Inf	N/A	327	60,05	U	Fails	4.14%
unitcal_7	19635617,2	28,17	19642524,39	60,05	N	**0.0%**	0.04%
var-smallemery-m6j6	Inf	N/A	-141,12	60,06	U	Fails	5.53%
wachplan	Inf	N/A	-8	60,03	U	Fails	0.0%
					Solved	127(6)	145(3)
					Average	**104.48%**	692.78%
					Worst	**8900.0%**	96190.48%
					Clean	**8.26%**	12.81%

Bibliography

Enrico Angelelli, Renata Mansini, and M. Grazia Speranza. Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, 37(11):2017–2026, 2010. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2010.02.002. Metaheuristics for Logistics and Vehicle Routing.

Enrico Angelelli, Renata Mansini, and M.Grazia Speranza. Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications*, 51:345–361, 01 2012. doi: 10.1007/s10589-010-9326-6.

Paolo Brandimarte. Multi-item capacitated lot-sizing with demand uncertainty. *International Journal of Production Research*, 44(15):2997–3022, 2006. doi: 10.1080/00207540500435116.

Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

Ambros et al. Gleixner. Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 2021. doi: 10.1007/s12532-020-00194-3. URL https://doi.org/10.1007/s12532-020-00194-3.

G. Guastaroba and M.G. Speranza. Kernel search: An application to the index tracking problem. *European Journal of Operational Research*, 217(1):54–68, 2012. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2011.09.004.

G. Guastaroba, M. Savelsbergh, and M.G. Speranza. Adaptive kernel search: A heuristic for solving mixed integer linear programs. *European Journal of Operational Research*, 263(3):789–804, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2017.06.005.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL https://www.gurobi.com.

Leo Liberti, Sonia Cafieri, and Fabien Tarissan. *Reformulations in Mathematical Programming: A Computational Approach*, pages 153–234. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-01085-9. doi: 10.1007/978-3-642-01085-9_7.

Renata Turkeš, Kenneth Sörensen, and Daniel Palhazi Cuervo. A matheuristic for the stochastic facility location problem. *Journal of Heuristics*, 27(4):649–694, August 2021. doi: 10.1007/s10732-021-09468-.