Erasmus University Rotterdam

Erasmus School of Economics

Bachelor Thesis Econometrics and Operations Research

Quantitative Logistics

Evaluating and Extending Adaptive Kernel Search for General Mixed Integer Linear Programming

Abstract

This thesis analyzes the Adaptive Kernel Search (AKS) heuristic for finding solutions to general Mixed Integer linear Programs (MIPs) as proposed by Guastaroba, Savelsbergh, and Speranza (2017). The performance of this heuristic is evaluated by solving a set of problem instances and comparing the results to the results of the commercial solver CPLEX using the same time limit. AKS creates a kernel of promising variables and adapts the kernel size based on the difficulty of the problem instance. A weak point regarding the decrease of the kernel size for hard instances is identified, and an extension is suggested to improve the heuristic. This extension includes an iterative process of increasing the kernel size again after it has been decreased. On average, this extended version of AKS finds better solutions in a specific group of instances for which the original version of AKS used less than the available time.

Author:
Martijn Bij de Vaate (476061mv)

Supervisor:
R. Hoogervorst
Second assessor:
Prof. Dr. D. Huisman

Date final version: 5 July 2020

The views stated in this thesis are those of the author and not necessarily those of the supervisor, second assessor, Erasmus School of Economics or Erasmus University Rotterdam.

Contents

1 Introduction **2**

2 Literature **3**

2.1 Mixed Integer Linear Programming 3

2.2 Primal Heuristics 3

2.3 Improvement Heuristics 3

3 Methodology **4**

3.1 Kernel Search 4

3.2 Adaptive Kernel Search 5

3.3 Extension: Unfixing Kernel Variables 7

4 Results **9**

4.1 Problem Instances 9

4.2 Results Adaptive Kernel Search 10

4.3 Results Extension 15

5 Conclusion **19**

1 Introduction

Many practical problems from different disciplines and industries can be formulated as a Mixed Integer linear Program (MIP). This formulation offers a way to model a decision making process with restrictions in a mathematical way. Finding the best solution to such a problem boils down to making the decisions in an optimal way, such that the objective value is optimized. Such an objective can be defined as the total costs associated with the decisions, the total distance traveled, etc.

The objectives of MIPs are often directly or indirectly related to money, for example production costs or total profit. This makes it clear that finding a better solution is valuable for the individual or company facing the problem. Algorithms exist to solve such problems to optimality, but as the problem instances increase in size, the computation time to find the optimal solution using such an algorithm also increases. It is therefore necessary to develop heuristics which are able to find a relatively good solution within a prespecified amount of time.

An example of such a heuristic is *Adaptive Kernel Search* (AKS), presented by Guastaroba, Savelsbergh, and Speranza (2017). AKS is an extension of *Kernel Search* (KS), which was developed by Angelelli, Mansini, and Speranza (2010). KS is a heuristic which divides the non-continuous variables into a set of promising variables and non-promising variables. Initially, a feasible solution to the problem is found using only the set of promising variables, referred to as the *kernel*. Afterwards, this solution is improved in an iterative process. The algorithm repetitively calculates whether including a subset of the non-promising variables improves the solution, trying a different subset in each iteration. One of the extensions of AKS as compared to KS is to identify the difficulty of a problem instance as *easy*, *normal* or *hard*, and to use this in the remainder of the algorithm by increasing or decreasing the kernel size. Secondly, AKS uses a better method to find a feasible solution in the initial phase of the algorithm.

Based on a set of 137 problem instances, Guastaroba et al. (2017) conclude that when a fixed computation time is used, AKS yields better results than the commercial solver CPLEX. Besides, by changing the parameters of AKS, the user can trade quality of the solutions against computation time. This thesis analyzes the performance of AKS in comparison to CPLEX. It also identifies a weak point of AKS. For instances that are considered hard, decreasing the kernel size sometimes is too drastic and leads to short computation times and relatively bad solutions. An extended version of AKS is suggested which deals with this problem by iteratively increasing the kernel size again. On average, this extension finds better solutions in case an instance did not use all of the available time when being solved by the original version of AKS. In case the original version of AKS used all of the available time to solve an instance, the extension usually found the same or a worse solution than the original version of AKS.

The remainder of this thesis is structured as follows. Section 2 contains a description of MIPs and some well-performing heuristics to solve them. Section 3 describes KS as well as AKS and the extension of AKS. Section 4 starts with a description of the problem instances used by Guastaroba et al. (2017) and how these instances will be used in this thesis. It then proceeds to the results of this thesis, which are the analysis of AKS and its extended version. Lastly, Section 5 provides a conclusion.

2 Literature

2.1 Mixed Integer Linear Programming

MIPs consist of a number of decision variables $x_1, ..., x_n$. Different types of decision variables exist. Some variables are continuous, which means that they can attain any value from the real numbers. Integer variables are restricted to the set of non-negative integers, and binary variables to 0 and 1. The MIP contains linear restrictions on the variables, as well as a linear objective function in terms of the variables. The aim of the model is to find values for the decision variables which minimize the objective function while satisfying all of the constraints. An MIP without integer variables is a special case called 0-1 MIP.

A lot of research has been done on heuristical methods to solve MIPs. Guastaroba et al. (2017) indicate that those heuristics can be divided into two groups. The first group consists of primal heuristics. Those heuristics aim to find an initial feasible solution to a problem instance. The other group consists of improvement heuristics, which try to improve a feasible solution. The remainder of this section describes some examples of primal heuristics and improvement heuristics.

2.2 Primal Heuristics

A well-performing primal heuristic is the Feasibility Pump (FP) heuristic, which has been proposed by Fischetti, Glover, and Lodi (2005). As a starting point, an infeasible solution $\tilde{x}$ is used. Defining P as the feasible region of the LP relaxation of the MIP, the following step is to find the point $x^* \in P$ closest to $\tilde{x}$. This will generally not be a solution to the MIP, since the integer and binary variables may now attain continuous values. So the next step is to round the integer and binary variables of x^* such that the integrality constraints are satisfied again. This new point might be infeasible again. It replaces $\tilde{x}$, after which the process repeats until the distance between two points is zero. When this situation is reached, it means that a point has been found satisfying both the integrality constraints and all the other constraints. Such a point is a feasible solution to the MIP.

Several researches have been done on how to extend the FP heuristic, for example by Fischetti and Lodi (2008), who combine the FP heuristic with Local Branching (LB) from Fischetti and Lodi (2003). Their algorithm starts by finding a promising, but infeasible solution. Next, LB is performed to find a feasible solution. This idea of combining a heuristic to find an initial, although infeasible, solution with an improvement heuristic has also been applied to construct the Kernel Search heuristic, which we will describe in Section 3.

2.3 Improvement Heuristics

Lazić, Hanafi, Mladenović, and Urošević (2010) defined the Variable Neighborhood Decomposition Search (VNDS) heuristic, which has proven to be among the best heuristics to solve 0-1 MIPs. An initial feasible solution and the optimal solution to the LP relaxation are used as input for the heuristic. First, the distance between the values in these two solutions is calculated for all binary variables. The binary variables are ordered based on these distances, starting from the variable with the smallest distance. Next, the first k variables according to this order are fixed to their value in the best feasible solution found so far. Initially, most variables will be fixed. Using this restriction, the MIP is solved. If the objective value has not been improved, fewer variables will be fixed, and the MIP will be solved again. This is repeated until a solution is found with a better objective value than the objective value of the best solution found so far. Variable

Neighbourhood Descent (VND) will be applied to this new solution, and the resulting solution will take the role of the initial feasible solution, after which the whole process starts over. This is repeated until the time limit has been reached.

3 Methodology

This Section describes the Kernel Search (KS) heuristic as proposed by Angelelli et al. (2010), as well as the Adaptive Kernel Search (AKS) heuristic, proposed by Guastaroba et al. (2017), which is an extension of KS.

3.1 Kernel Search

KS is a heuristic for finding solutions to 0-1 MIPs. It uses a set of *promising* binary decision variables, called the kernel. Those variables are promising in the sense that they are likely to be equal to one in the optimal solution. The heuristic searches for an initial feasible solution, after which it tries to improve this solution. To this end, the kernel is changed by adding promising variables to it and removing non-promising variables from it. Comparing this to the division of heuristics into two groups as described in Section 2, we can see that KS combines a primal heuristic with an improvement heuristic. The idea of KS has been successfully used by among others Kirschstein and Meisel (2019), who constructed a heuristic based on KS to solve a problem regarding the production process in the chemical industry.

As a first step towards a feasible solution, an initial kernel K needs to be created. This is done by solving the Linear Programming (LP) Relaxation, and letting the kernel consist of all binary variables with a value greater than zero in the LP solution. The remaining binary variables are sorted by non-increasing reduced cost. Using this order, they are put into so called buckets $B_1, B_2, ..., B_{N_b}$ of certain size. For example, this size could be equal to the kernel size. This would mean that a problem instance with 35 binary decision variables and a kernel size of 10 would need $N_b = 3$ buckets. The first two buckets would contain 10 variables each, whereas the last bucket would only contain 5 variables. The 0-1 MIP may also contain continuous variables. They are neither in the kernel nor in the buckets.

The next part of the heuristic is to find a feasible solution to the problem. This is done by solving a restricted MIP, denoted by MIP(K), which is a version of the original MIP with extra constraints. In this restricted MIP, all binary decision variables, except those in K, are restricted to zero. If the set of binary decision variables is denoted by B, this restriction would be given by $x_j = 0, \quad j \in B \setminus K$. Any continuous variables are not restricted in the restricted MIP.

The final part of the KS algorithm tries to improve the initial feasible solution. Such a solution may not have been found yet, in which case this part of the algorithm could find the first feasible solution. This *improvement phase* consists of a number of iterations, defined by the number of buckets N_b, or a prespecified maximum number of iterations $\bar{N}_b$, if $\bar{N}_b < N_b$. Iteration i corresponds to bucket B_i. The variables of this bucket will not be restricted to zero anymore. That is, MIP($K \cup B_i$) is solved. To this restricted MIP, two more constraints are added. The first constraint is that the objective value should be less than (or, if the programming environment requires this, equal to) the objective value of the best solution found so far, since we are looking for improvement of this solution. The other constraint is given by $\sum_{j \in B_i} x_j \geq 1$. It implies that at least one of the variables in the bucket should be equal to one. Both constraints reduce the search space, therefore they speed up the process of finding better solutions. When this restricted MIP has been solved, the kernel needs to be updated in case a feasible or optimal solution has been found. Such a solution will be better or exactly as good as the best solution found before that. Of all variables in bucket B_i, those

with a value of one in the solution of MIP($K \cup B_i$) are added to the kernel. Next, we remove those variables from the kernel that have been equal to zero for p iterations, in order to prevent the size of the kernel from becoming too large. Here, p is a prespecified parameter.

When all iterations have been performed, the best solution found so far is the final solution. However, it is not guaranteed that a feasible solution has been found.

3.2 Adaptive Kernel Search

One problem with KS is the kernel size and how it depends on the problem instance. A larger kernel size generally means better solutions, but also longer computation times. In case KS can be applied to a problem instance in a very short time, it would be possible to increase the kernel size, without creating extremely long computation times. On the other hand, if it takes a very long time to apply KS to a certain problem instance, it would be desirable to decrease the size of the kernel.

The possibility to adapt the kernel based on the difficulty of the problem instance is one of the changes Guastaroba et al. (2017) made to KS when they proposed AKS. This change and the other changes have been described in the following paragraphs. Algorithm 1 provides pseudocode for AKS.

LP relaxation

In KS, the LP relaxation of the problem instance is solved. This is done by relaxing all the integrality constraints. In AKS, this has been changed to the LP relaxation as solved by CPLEX when the first node of the search tree is evaluated. In its basic form, this equals the LP relaxation of the complete problem instance. However, CPLEX adds some functionality such as preprocessing the problem and creating cuts. This means that a higher bound is obtained, which in turn generally results in a better initial kernel.

Initial feasible solution

In case MIP(K) is an infeasible problem, there are too many constraints. Some constraints need to be relaxed to find a feasible solution, which is done by adding variables to the kernel. Repeatedly, the variables from w buckets are added to the kernel, starting from the first bucket. If there exists any solution to the overall problem, and there is no time limit for the new restricted MIPs that are created, this procedure should find a feasible solution. Once it has been found, the remaining non-kernel binary variables are again divided into buckets. If the bucket length is taken to be equal to the kernel size, there will now be fewer buckets, but with more variables per bucket.

Adaptation of the kernel size

After having found an initial feasible solution to the problem instance, the kernel size is adapted based on the computation time of the last restricted MIP that has been solved, and which found the initial solution. This computation time will be denoted by t_{init} and compared to certain prespecified thresholds: if this time is less than or equal to t_{easy}, the problem instance is supposed to be *easy*; if this time is greater than or equal to t_{hard}, the problem instance is supposed to be *hard*. Otherwise, the problem instance is supposed to be *normal*. This information is used to adapt the kernel accordingly. For normal instances, the kernel will not be adapted. Instead, the algorithm immediately proceeds to the improvement phase.

The adaption of the kernel for easy instances is to add the variables of the first q buckets to the kernel and solve MIP(K) again, now using the adapted kernel. To find a better solution, at least one of the new

Algorithm 1 Adaptive Kernel Search

```
    Initialization phase
 1: Solve the root node relaxation in CPLEX
 2: Create a kernel K and buckets B_1, ..., B_{N_b}
 3: Solve MIP(K)
 4: if a feasible solution has not been found then
 5:     while a feasible solution has not been found do
 6:         Add bucket variables to kernel K
 7:         Solve MIP(K)
 8:     end while
 9:     Create buckets B_1, ..., B_{N_b}
10: end if

    Adaptation phase
11: Classify the difficulty of the problem instance as easy, normal or hard
12: if the problem instance is easy then
13:     while the problem instance is easy and the kernel K can be extended do
14:         Add bucket variables to kernel K
15:         Solve MIP(K)
16:     end while
17:     if the kernel K can be extended then
18:         Create buckets B_1, ..., B_{N_b}
19:     else
20:         return
21:     end if
22: else if the problem instance is hard then
23:     for every variable x_j in K do
24:         if the value of variable x_j at the root node relaxation differs at most ϵ from the closest integer
    then
25:             Add constraint x_j = 1 to the problem instance
26:         end if
27:     end for
28: end if

    Improvement phase
29: for i = 1, 2, ..., min{N_b, N̄_b} do
30:     Solve MIP(K ∪ B_i)
31:     Adjust the kernel K based on the solution
32: end for
33: return
```

variables should equal one, since they were all equal to zero in the best solution found so far. Therefore, to cut off a part of the search space, the constraint $\sum_{j \in K^+} x_j \geq 1$ can be added to the restricted MIP. Here, K^+ denotes the set of new kernel variables. As long as the time to solve this problem does not exceed t_{easy}, the variables of another q buckets are added, following their initial order of reduced cost. It could occur that all the binary variables have been added to the kernel and the solution can still be found, in which case the optimal solution to the overall problem has been found and the algorithm stops. In case the algorithm does not stop, the process of creating buckets of variables is repeated with the remaining non-kernel binary variables. AKS then proceeds to the improvement phase.

Hard instances are adapted by restricting some of the kernel variables to one for the remainder of the algorithm. Those variables are the binary decision variables with a value greater than $1 - \epsilon$ in the LP relaxation at the root node. Here, ϵ is a prespecified value. After this adaptation, the improvement phase is carried out.

MIPs with integer decision variables

Whereas KS has only been defined for 0-1 MIPs, so MIPs without integer decision variables, AKS also works for general MIPs. To create the kernel and the buckets, the binary and integer variables are handled separately. Non-zero variables are put into the kernel, while the remaining variables are ordered based on their reduced cost. This results in two kernels and two sets of buckets, which are then combined for the remainder of the algorithm. In case either the binary or the integer variables have more buckets, the last couple of combined buckets consist only of one type of variables.

When adjusting the kernel size by fixing variables in the case of a hard instance, integer variables with an absolute difference of at most ϵ to the nearest integer will be fixed to this integer.

Regarding the improvement phase, KS uses an extra constraint which ensures to have at least one of the binary variables of the buckets equal to one. In AKS, this changes to having at least one bucket variable which is greater than zero. The mathematical formulation of this constraint remains unchanged: $\sum_{j \in B_i} x_j \geq 1$.

3.3 Extension: Unfixing Kernel Variables

In order to reduce the complexity of the hard instances, AKS fixes some of the kernel variables at a promising value before starting the improvement phase. This may turn the remaining problem into a very easy one, such that the improvement phase can be performed in a small amount of time. However, this small computation time may come at a price of a bad solution. In such cases, we might want to leave out the fixation of variables. This will result in a longer computation time and a better solution for some instances.

The AKS heuristic is extended by implementing the idea described above. For hard instances, each iteration of the improvement phase is split into several sub-iterations. The first of these sub-iterations equals the usual iteration from AKS. In the next sub-iteration, some of the variables that were fixed to a certain value, will not be fixed to this value anymore. In other words, some of the fixation constraints are removed from the problem. In the next iteration, even more fixation constraints are removed, until the last sub-iteration does not contain any fixation constraints anymore. This means that the last sub-iteration is equal to a usual iteration if the instance were treated as a *normal* instance, in which case no variables are fixed. The order in which the variables are unfixed is determined by their difference with the nearest integer, or the value to which they were fixed. The variable with the largest difference is unfixed first, and the variables which is closest to the nearest integer is unfixed last.

The number of sub-iterations per iteration is controlled by a pre-specified parameter $f = 1, 2, ...$ If the number of variables that are fixed, N_f, is less than f, f is adjusted to N_f. During each sub-iteration, $1/f$ of the fixed variables is unfixed. This results in a total of $f + 1$ sub-iterations per iteration. The time limit of each sub-iteration is determined by dividing the remaining time for the iteration as a whole by the remaining number of sub-iterations. This implies that if a sub-iteration is finished before the time that was allocated to it, the remaining time is shared equally among the next sub-iterations of the current iteration. Algorithm 2 contains the pseudocode for this extended version of AKS.

This extension to AKS will likely perform well for hard instances for which the computation time of AKS is below the time limit. This generally means that the iterations were solved in a small amount of time. Now, the time that is left will be used to unfix some of the variables and search for a better solution in the search space that was added to the problem. However, if an iteration needed all or a substantial part of the available time, this extension might also result in worse solutions. That is, the original iteration, now the first sub-iteration, will now be aborted earlier, to allow for the other sub-iterations to be solved. At this point, the first sub-iteration may not have found the solution it could have found with some more time. Also, the next sub-iterations may not find better solution than the first sub-iteration because of the increased complexity of the problem.

It follows from this argument that better solutions can be expected for some of the hard instances, but a better or equal solution is not guaranteed. Section 4.3 provides the results of this extension and a discussion thereof.

Algorithm 2 Extended version of Adaptive Kernel Search

```
    Initialization phase as in Adaptive Kernel Search

    Adaptation phase as in Adaptive Kernel Search

    Improvement phase
 1: for i = 1, 2, ..., min{N_b, N̄_b} do
 2:     if the problem instance is hard and the number of fixed variables, N_f, is greater than 0 then
 3:         Solve MIP(K ∪ B_i)
 4:         Adjust the kernel K based on the solution
 5:         for k = 1, 2, ..., min{f, N_f} do
 6:             Unfix a total of k/min{f, N_f} kernel variables
 7:             Solve MIP(K ∪ B_i)
 8:             Adjust the kernel K based on the solution
 9:         end for
10:     else
11:         Solve MIP(K ∪ B_i)
12:         Adjust the kernel K based on the solution
13:     end if
14: end for
15: return
```

4 Results

The AKS algorithm as described in Section 3 has been implemented in Java using version 12.10.0 of the commercial solver CPLEX. Appendix A explains the set-up of this code. This section starts with a description of the problem instances from Guastaroba et al. (2017), and which of those will be used in this thesis. Two changes have been made to the original algorithm, which will be discussed in Section 4.2. Afterwards, the settings of the AKS heuristic will be given, as well as the results. These results will be compared and discussed in detail, which also shows the need of an extension. Finally, the results of the extension of AKS will be given and discussed.

4.1 Problem Instances

Guastaroba et al. (2017) used a set of 137 problem instances. 29 of those are called the *Lazić* instances, since they were used by Lazić et al. (2010) for evaluating the results of VNDS. Earlier, they were also used by Fischetti and Lodi (2003), who composed this set of problem instances from different disciplines. The other 108 problem instances are taken from the *MIPLIB2010*, a library of MIPs. Both the Lazić and MIPLIB2010 instances are taken from different applications, such as crew scheduling, network design and rolling stock planning.

This thesis will extend the AKS heuristic, after which its results will be compared to the results of the original version of AKS. In order to be able to compare our results to the results of Guastaroba et al. (2017), the same instances as described before will be solved. Due to limited time over the course of this thesis, we will need to use a subset of these instances. This subset will contain a variety of instances, e.g. instances that are classified as *easy* and *hard* by AKS, instances with relatively few and many decision variables, etc. Tabel 2 shows that 17 out of 29 Lazić instances can be solved to optimality within 1 hour using CPLEX 12.10.0 on a Dell laptop with Windows 10 as its operating system and with an Intel core i7 CPU (2.6 GHz) and 16 GB of RAM. The other 12 instances need more than 1 hour to be solved to optimality. This means that the instances have different difficulty. The number and types of variables also varies among the Lazić instances, which is why we let them be part of our set of problem instances.

The MIPLIB2010 instances are generally harder than the Lazić instances. Guastaroba et al. (2017) divide them into 53 instances to which the optimal solution was known at the time of their research, and 55 instances to which the optimal solution was unknown at the time of their research. These two sets of problem instances are respectively referred to as *MIPLIB2010-OS* and *MIPLIB2010-NOS*. We will add ten problem instances from MIPLIB2010-OS and ten from MIPLIB2010-NOS to the set of problem instances used in this thesis. Because the Lazić instances already contain some easier instances, but also because the extension to AKS in this thesis focuses on hard instances, these will be the first ten problem instances which are identified as hard by AKS when ordered alphabetically. They include instances from different fields and with different characteristics, such as different numbers of variables. Table 1 provides an overview of the problem instances for this thesis, ordered by their difficulty as assessed by AKS (see Table 2).

Table 1: Problem Instances

Lazić easy	Lazić normal	Lazić hard	MIPLIB2010-OS hard	MIPLIB2010-NOS hard
arki001	A1C1S1	B1C1S1	bg512142	circ10-3
danoint	A2C1S1	B2C1S1	dc1c	dano3mip
glass4	net12	biella1	dg012142	dc1l
markshare1	seymour	mkc	germanrr	ex1010-pi
markshare2	sp97ar	NSR8K	germany50-DBM	lectsched-1-obj
nsrand_ipx	UMTS	rail2586c	go19	n3700
rail2536c		rail4284c	ivu52	n3705
rail507		rail4872c	maxgasflow	n370a
roll3000		swath	n3-3	neos-937815
sp97ic		van	neos-948126	ns1854840
sp98ar				
sp98ic				
tr12-30				

4.2 Results Adaptive Kernel Search

Changes to the algorithm

Firstly, the LP relaxation at the root node of the problem as solved by CPLEX has been used for creating the kernel, but not for ranking the bucket variables. This has to do with the fact that CPLEX does not offer a straightforward way to retrieve the reduced costs of variables while the problem is being solved. Instead, the LP relaxation of the complete problem has been used to find the reduced costs and to rank the variables to create the buckets.

Secondly, the constraint to use at least one of the newly added variables when adapting the kernel for easy instances has been left out. Its function is to reduce the search space by preventing the heuristic from searching for solutions that have already been found during the previous iteration. These solutions had been found within the time limit t_{easy}, which means there is not much time to be gained by adding this constraint. Moreover, it increases the complexity of the algorithm when implementing it in a programming environment, because of the need to keep track of the previous solution during the procedure to adapt the kernel of easy instances. We therefore chose to leave out this constraint.

Settings

The problem instances have been solved with CPLEX and by applying AKS, using the PC described in Section 4.1. Solving the instances using CPLEX has been done with the default settings and a time limit of 1 hour (3600 s). This has been denoted by CPLEX(1h). AKS also has a time limit of 1 hour, which is implemented as follows: each LP relaxation or restricted MIP is limited to the remaining time. Some restricted MIPs are limited to an even shorter time. Those are MIP(K) from line 3 of Algorithm 1 with a time limit of $\bar{t}$, which is $1/(1+N_b)$ multiplied by the remaining time, and the other restricted MIPs solved to find a feasible solution (Algorithm 1, line 7), which have a time limit of $2 \cdot \bar{t}$. Finally, we have the restricted MIPs during the improvement phase (Algorithm 1, line 30), which all get an equal share of the remaining time. For example, if the remaining time is 2700 seconds and iteration 5 out of 13 is about to start, it will be limited to a computation time of $2700/9 = 300$ seconds.

The other parameters are taken as follows: the bucket length is equal to the kernel size and there is no

maximum number of buckets ($\bar{N}_b = \infty$). Variables remain in the kernel, even if they have been equal to zero for several iterations ($p = \infty$). Next, $w = 0.3$, $q = 0.35$, $\epsilon = 10^{-5}$, $t_{easy} = 10$ seconds and t_{hard} is equal to the time limit of the last restricted MIP, so either $\bar{t}$ or $2 \cdot \bar{t}$. This version of the AKS heuristic is referred to as AKS(1h).

Finally, the LP relaxations and MIPs in AKS were solved by CPLEX. The default settings were used, except for the following changes. All restricted MIPs (lines 3, 7, 15 and 30 of Algorithm 1) are solved with bound strengthening (parameter `Preprocessing.Boundstrength` = 1). In addition, finding feasible solutions is emphasized in the restricted MIPs of the improvement phase (line 30 of Algorithm 1), by setting parameter `Emphasis.MIP` to `HiddenFeas` and the Feasibility Pump heuristic parameter (`MIP.Strategy.FPHeur`) to 1.

Table 2: Results of CPLEX(1h) and AKS(1h)

Instance	Decision variables				CPLEX(1h)		AKS(1h)		
	Cont.	Bin.	Int.	Tot.	Objective	CPU (s)	Category	Gap (%)	CPU (s)
arki001	850	415	123	1388	7581043.7	8	easy	0.00	53
danoint	465	56	0	521	65.7	463	easy	0.00	786
glass4	20	302	0	322	1200012600.0	150	easy	0.00	135
markshare1	12	50	0	62	4.0	3600	easy	0.00	3600
markshare2	14	60	0	74	12.0	3600	easy	-8.33	3600
nsrand_ipx	1	6620	0	6621	51200.0	73	easy	0.00	451
rail2536c	9	15284	0	15293	689.0	35	easy	0.00	110
rail507	10	63009	0	63019	174.0	80	easy	0.00	1745
roll3000	428	246	492	1166	12890.0	15	easy	0.00	54
sp97ic	0	12497	0	12497	427684487.7	3109	easy	0.59	3600
sp98ar	0	15085	0	15085	529740623.2	408	easy	0.12	3600
sp98ic	0	10894	0	10894	449144758.4	123	easy	0.00	1368
tr12-30	720	360	0	1080	130596.0	126	easy	0.00	166
A1C1S1	3456	192	0	3648	11503.4	1298	normal	0.00	737
A2C1S1	3456	192	0	3648	10889.1	870	normal	0.00	1285
net12	12512	1603	0	14115	214.0	139	normal	0.00	127
seymour	0	1372	0	1372	423.0	3602	normal	0.00	3602
sp97ar	0	14101	0	14101	660729183.0	2934	normal	0.53	3600
UMTS	73	2802	72	2947	30090583.0	277	normal	0.00	1263
B1C1S1	3584	288	0	3872	24544.2	3600	hard	0.00	3600
B2C1S1	3584	288	0	3872	25812.0	3600	hard	0.22	3600
biella1	1218	6110	0	7328	3065037.5	125	hard	0.00	1402
mkc	2	5323	0	5325	-563.8	3606	hard	6.49	266
NSR8K	6316	32040	0	38356	857676860.7	3600	hard	-6.65	3600
rail2586c	11	13215	0	13226	953.0	3600	hard	0.31	3600
rail4284c	9	21705	0	21714	1072.0	3600	hard	0.09	3600
rail4872c	11	24645	0	24656	1550.0	3600	hard	-0.39	3600
swath	81	6724	0	6805	467.4	3603	-	27.15	3600
van	12289	192	0	12481	4.8	3600	hard	0.00	3600
bg512142	552	240	0	792	188862.2	3601	hard	0.90	3601
dc1c	1659	8380	0	10039	1782289.6	3600	hard	0.11	1880
dg012142	1440	640	0	2080	2596277.0	3600	hard	-10.69	2162
germanrr	239	5288	5286	10813	47095871.2	2054	hard	0.36	256
germany50-DBM	8101	0	88	8189	474980.0	3600	hard	0.09	2162
go19	0	441	0	441	84.0	3603	hard	0.00	3603
ivu52	0	157591	0	157591	490.1	3601	hard	-1.71	2181
maxgasflow	4981	2456	0	7437	-44565689.0	3600	hard	0.25	1943
n3-3	8662	0	366	9028	16072.0	3600	hard	0.49	1880
neos-948126	2586	6965	0	9551	2609.0	3600	hard	3.64	1806
circ10-3	0	2700	0	2700	408.0	3600	hard	-5.39	3601
dano3mip	13321	552	0	13873	691.8	3600	hard	0.02	3600
dc1l	1659	35638	0	37297	1782855.4	3600	hard	5.86	3600
ex1010-pi	0	25200	0	25200	241.0	3600	hard	0.00	3600
lectsched-1-obj	0	28236	482	28718	83.0	3600	hard	0.00	2408
n370a	5000	5000	0	10000	1277711.0	3600	hard	0.82	3600
n3700	5000	5000	0	10000	1280361.0	3600	hard	-1.75	3600
n3705	5000	5000	0	10000	1281088.0	3600	hard	-1.16	3600
neos-937815	2770	8876	0	11646	2859.0	3600	hard	1.19	1450
ns1854840	0	135280	474	135754	2242000.0	3600	hard	-91.17	3600
Average	**2247**	**14064**	**151**	**16462**		**2529**		**-1.59**	**2336**

Comparison CPLEX(1h) and AKS(1h)

Table 2 shows the results of applying CPLEX(1h) and AKS(1h) to the 49 problem instances. For each instance, the number of continuous, binary and integer variables is given, as well as the total number of

variables. The next two columns provide the objective value obtained by CPLEX(1h), and the actual CPU time, which had a limit of 1 hour (3600 seconds). A CPU time of less than 3600 seconds implies that the optimal solution was found. However, the objective value given in the table may be slightly greater than the actual optimal value due to the default value of the MIP Gap Tolerance parameter in CPLEX, which is set to 1E-4. For AKS(1h), the category of the instance is given, as well as the gap between the objective values of AKS(1h) and CPLEX(1h). This gap has been calculated as $\frac{z_A - z_C}{|z_C|} \cdot 100\%$, where z_C and z_A denote the objective values found by CPLEX(1h) and AKS(1h), respectively. Thus, a gap of 0% means that AKS(1h) found the same value as CPLEX(1h). A positive gap means that it found a worse solution, and a negative gap means that the solution of AKS(1h) was better than the solution of CPLEX(1h). Lastly, the CPU time of AKS(1h) is given in the right-most column.

The problem instances in Table 2 have been grouped into 5 blocks of rows. The first three blocks correspond to the easy, normal and hard Lazić instances. The difficulty of instance `swath` could not be identified, because the part of the heuristic in which the difficulty is assessed was not reached before the end of the time limit. The fourth block corresponds to the MIPLIB2010-OS instances, and the MIPLIB2010-NOS instances are presented in the fifth block.

This ordering of the problem instances makes it clear that CPLEX(1h) was able to solve to optimality 11 out of 13 easy instances, 5 out of 6 normal instances and only 2 out of 29 hard instances. On average, the computation time was 2529 seconds. As expected, the computation time increased with the difficulty of the problem instances, which shows that AKS(1h) generally succeeds in assessing the difficulty of a problem instance.

Especially for the easy and normal instances, the gap between AKS(1h) and CPLEX(1h) is often equal to 0%. This happens 13 times. In 3 cases, a better solution was found. This includes problem instances `arki001` and `UMTS`, for which the gaps are slightly negative, but equal to 0.00% when rounded to two decimals. In the other 3 cases, a worse solution was found. Without outliers of more than 5% in absolute values, the average gap equals 0.07%.

Among the hard instances, the results of AKS(1h) are more often different from CPLEX(1h), both better and worse. In 5 cases, the same solution was found, but in 9 cases a better solution is found and in 15 cases a worse solution is found. On average, again after discarding outliers with gaps of more than 5% in absolute values, the gap between AKS(1h) and CPLEX(1h) equals 0.15% for hard instances. For the problem instances used in this thesis, AKS(1h) seems to perform slightly worse than CPLEX(1h), but further research with a larger set of instances will have to show whether this difference is significant.

Explaining the performance of AKS(1h)

We will now attempt to explain the gaps between AKS(1h) and CPLEX(1h) in order to gain some more insight into the robustness of AKS. The upper graph of Figure 1 shows the relationship between the proportion of binary variables and the gap. Gaps with an absolute value of 5% or more have been removed. It seems that positive and negative gaps are spread evenly along the horizontal axis of the graph, which means that the proportion of binary variables does not explain the gap.

Regarding the proportion of integer variables, there is not much we can say about its effect on the performance of AKS(1h), since there were only a few instances with integer variables among our problem instances. These did not show any remarkable behavior. Appendix B shows the graph of the relationship between the proportion of integer variables and the gap. The fact that most instances did not contain any integer variables means that the graph which shows the relationship between the proportion of continuous

variables and the gap resembles the upper graph of Figure 1, but with a reversed horizontal axis. The proportion of continuous variables therefore does not explain the gaps either. Appendix B also contains this graph.

The lower graph of Figure 1 show the relationship between the total number of variables of a problem instance and the gap between AKS(1h) and CPLEX(1h). The total number of variables is given on a logarithmic scale. Most instances have around 10,000 variables, which explains why many of the positive gaps occur in the same area. There does not seem to be a trend between the number of variables and the gap.

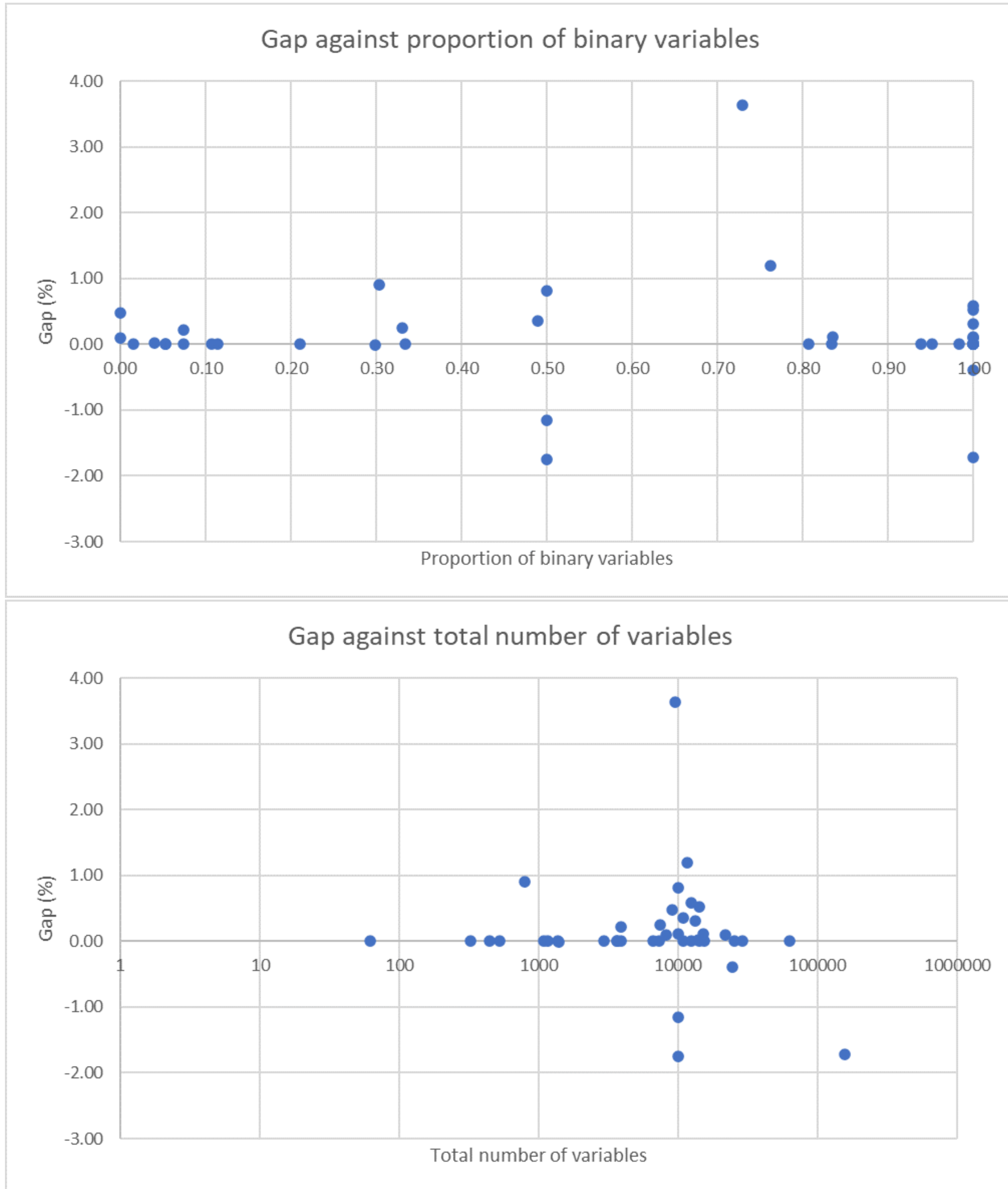

Figure 1: Performance of AKS(1h) for different instance characteristics (without outliers)

Table 3 provides the average gaps for the different categories of instances as identified by AKS(1h). After discarding the gaps which are greater than 5% in absolute values, the averages are 0.06%, 0.09% and 0.15%, so none of the categories stands out in terms of performance of AKS(1h). We might say that there seems to be an upward trend of the gap as the difficulty increases. However, the scale of the gaps makes it difficult to compare the average values. For example, if we remove the hard instance with the highest gap (3.64%), the average gap of the hard instances drops to -0.01%.

Table 3: Performance of AKS(1h) for different categories

Category	**Frequency**	**Av. Gap (%)**
easy	11	0.06
normal	6	0.09
hard	23	0.15

To conclude, we can say that neither the type or number of variables nor the difficulty of a problem instance affects the performance of AKS(1h), as compared to CPLEX(1h). This means that AKS is a robust technique to find solutions to general MIPs, in the sense that many problem characteristics do not influence the results. In other words, there are no groups of instances based on the characteristics discussed above for which AKS is likely to find very good or bad results.

Comparison with Guastaroba et al. (2017)

In order to be able to make a comparison, all the settings of CPLEX(1h) and AKS(1h) have been kept equal to the settings of CPLEX(5h) and AKS(5h) in Guastaroba et al. (2017), except for the time limit, which has been changed from 5 hours to 1 hour due to limited time over the course of this thesis. However, because of a newer version of CPLEX and the use of an average but relatively new PC, this difference did not become very evident in the results. This can be seen by examining the results of applying CPLEX to the instances. Guastaroba et al. (2017) solved 18 out of 49 instances within their 5 hour time limit. The same number of instances is solved within in the time limit of 1 hour in this thesis. When we look at the other 31 instances, Guastaroba et al. (2017) found the better solution in most cases.

Comparing AKS(5h) with AKS(1h) from this thesis, we see that the average gap with the corresponding version of CPLEX equals 0.11% for this thesis, and 0.57% for Guastaroba et al. (2017). Both average values have been calculated after discarding the gaps that were greater than 5% in absolute values. This result seems to show that the version from this thesis performs relatively better. However, we must note that AKS(1h) has many outliers, both positive and negative ones, whereas AKS(5h) only has two negative outliers. This fact and the scope of the gaps makes it hard to compare the performance. If further research would find any significant differences in performance, this could be due to the different time limits, the changes to the algorithm as described at the beginning of this section, or the difference in software and hardware. Appendix C provides the details of this comparison.

4.3 Results Extension

In Section 4.2, we have seen that the performance of AKS(1h) can not be explained by instance characteristics. Instead of trying to find patterns, we will now look closely at a specific instance with a large gap, and how

the extension of AKS proposed in Section 3.3 could be used to find better solutions in this case and similar cases.

Lazić instance `mkc` has a gap of 6.49% between the objective values of AKS(1h) and CPLEX(1h). It is a hard instance, with a kernel of 4563 variables. 4452 (97.6%) of those kernel variables are fixed to 1 because they are close to 1 in the LP relaxation at the root node. This adaptation of the kernel rules out a lot of solutions, and it means that the improvement phase can be performed in 0.015 seconds. Therefore, the total computation time is very small, even though we are dealing with a hard instance. A way to deal with this problem is to completely remove the adaptation of hard instances, which is essentially what KS does. It will eliminate problems such as with instance `mkc`, but it might also result in unsolvable restricted MIPs during the improvement phase of other problem instances.

The extension from Section 3.3 is a compromising option. For each iteration, the allocated time is used to solve different versions of the restricted MIP, with all, some or none of the original fixed variables. This extended version of AKS has been performed using $f = 4$, meaning that a quarter of all fixed variables was unfixed during each sub-iteration. All other settings are equal to AKS(1h). This new version of AKS will be denoted by E-AKS(1h). Table 4 shows the results of AKS(1h) and E-AKS(1h) on all 29 hard instances. Again, the gaps are calculated as compared to the solution of CPLEX(1h). The best gap is given in bold. The CPU times are also given.

Table 4: Results of AKS(1h) and E-AKS(1h)

Instance	**AKS(1h)**		**E-AKS(1h)**	
	Gap (%)	**CPU (s)**	**Gap (%)**	**CPU (s)**
B1C1S1	**0.00**	3600	**0.00**	3600
B2C1S1	**0.22**	3600	**0.22**	3600
biella1	**0.00**	1402	**0.00**	3600
mkc	6.49	266	**0.00**	3600
NSR8K	**-6.65**	3600	**-6.65**	3600
rail2586c	**0.31**	3600	1.47	3600
rail4284c	**0.09**	3600	1.31	3600
rail4872c	**-0.39**	3600	0.84	3600
van	**0.00**	3600	**0.00**	3600
bg512142	0.90	3601	**0.60**	3600
dc1c	0.11	1880	**-0.01**	3600
dg012142	**-10.69**	2162	3.24	3600
germanrr	0.36	256	**0.00**	3501
germany50-DBM	**0.09**	2162	**0.09**	3600
go19	**0.00**	3603	**0.00**	3603
ivu52	**-1.71**	2181	**-1.71**	3600
maxgasflow	0.25	1943	**0.04**	3600
n3-3	**0.49**	1880	**0.49**	3600
neos-948126	**3.64**	1806	3.68	3600
circ10-3	**-5.39**	3601	**-5.39**	3600
dano3mip	**0.02**	3600	**0.02**	3600
dc1l	**5.86**	3600	**5.86**	3600
ex1010-pi	**0.00**	3600	2.49	3600
lectsched-1-obj	**0.00**	2408	**0.00**	3600
n370a	0.82	3600	**0.53**	3600
n3700	**-1.75**	3600	-1.44	3600
n3705	**-1.16**	3600	-0.42	3600
neos-937815	1.19	1450	**-0.14**	3600
ns1854840	**-91.17**	3600	**-91.17**	3600
Average	**-3.38**	**2793**	**-2.97**	**3597**

For instance `mkc`, which only has one iteration in the improvement phase, the first three sub-iterations are solved within 1 second. The fourth iteration takes 10 seconds, and it brings down the gap to 4.74%. The fifth and last sub-iteration runs until the time limit is reached and decreases the gap even more to 0.00%, which means the E-AKS(1h) found a solution with the same objective value as CPLEX(1h).

On average, the gap of E-AKS(1h) is -2.97%, so a bit worse than the gap of AKS(1h), which is -3.38% for the set of hard instances. Therefore, the extension does not seem to be an improvement at first sight. However, if we split the instances into instances for which the time limit was reached in AKS(1h), and instances for which there was time left after performing AKS(1h), we see some interesting results. For the first group, there are 2 cases in which E-AKS(1h) improved the solution, 9 in which the solution stayed the same, and 6 in which the solution of E-AKS(1h) was worse than the solution of AKS(1h). For the instances which have a CPU time less than 3600 seconds for AKS(1h), there were 5 improved cases, 5 equal cases, and only 2 worse cases. This behavior could be expected, since the extension was specifically designed to improve the cases in which the improvement phase could be solved in a very small amount of time due to the fixation

of variables. On the other hand, instances which already needed all of the allocated time, might not benefit from expanding the search space.

These results point towards a strategy in which the extended version of AKS is used only for those instances which can be expected to need a small amount of time. It is a topic for further research to examine how these instances can be identified without running AKS(1h) first. Appendix D contains more details on the results of AKS(1h) and E-AKS(1h).

5 Conclusion

This thesis has analyzed Adaptive Kernel Search (AKS), a heuristic for finding solutions to MIPs. AKS creates a kernel of promising variables, after which the difficulty of the problem instance is identified to be easy, normal or hard. Based on this classification, variables are added to or removed from the kernel, in order to use the available time more efficiently for finding good solutions. A diverse set of 49 problem instances has been used to evaluate the performance of this heuristic and to compare it to the commercial solver CPLEX.

Two minor changes have been made to the original version of AKS as described by Guastaroba et al. (2017) because of programming reasons. Using a time limit of 1 hour, AKS has been compared to CPLEX. On average, AKS seemed to perform slightly worse than CPLEX, but further research should be performed to judge the significance of this difference.

Another result of this thesis is that AKS is a robust technique for finding solutions to general MIPs with different characteristics. This could be concluded after analyzing the relationship between different characteristics of instances and the performance of AKS, which did not show any trends.

This thesis also found a weak point of the AKS heuristic. When the kernel size is decreased for instances that are classified as hard, the problems that are solved during improvement phase are sometimes very easy, such that they can be solved in a small amount of time compared to the total amount of available time. This also yields relatively bad results. An extension of AKS has been proposed to overcome such problematic behavior. After decreasing the kernel size, variables are iteratively added back to the kernel to increase the search space and find better solutions.

This extension has proven to be successful for instances which were solved by the original version of AKS within the time limit. This indicates that AKS could have used the leftover time to find better solutions, which is indeed what the extended version of AKS did. On the other hand, this extended version usually did not find a better solution to the other hard instances. This makes the extension an interesting topic for further research. In a new version of AKS, the algorithm could be extended to assess whether or not the extension should be used.

References

[1] Enrico Angelelli, Renata Mansini, and M. Grazia Speranza. "Kernel search: A general heuristic for the multi-dimensional knapsack problem". In: *Computers & Operations Research* 37.11 (2010), pp. 2017–2026.

[2] Matteo Fischetti, Fred Glover, and Andrea Lodi. "The feasibility pump". In: *Mathematical Programming* 104 (2005), pp. 91–104.

[3] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical Programming* Ser. B 98 (2003), pp. 23–47.

[4] Matteo Fischetti and Andrea Lodi. "Repairing MIP infeasibility through local branching". In: *Computers & Operations Research* 35.5 (2008), pp. 1436–1445.

[5] Gianfranco Guastaroba, Martin Savelsbergh, and M. Grazia Speranza. "Adaptive Kernel Search: A Heuristic for Solving Mixed Integer Linear Programs". In: *European Journal of Operational Research* 263 (2017), pp. 789–804.

[6] Thomas Kirschstein and Frank Meisel. "A multi-period multi-commodity lot-sizing problem with supplier selection, storage selection and discounts for the process industry". In: *European Journal of Operational Research* 279.2 (2019), pp. 393–406.

[7] Jasmina Lazić, Saïd Hanafi, Nenad Mladenović, and Dragan Urošević. "Variable neighbourhood decomposition search for 0–1 mixed integer programs". In: *Computers & Operations Research* 37.6 (2010), pp. 1055–1067.

Appendix A: Summary of the written code

AKS and the extended version of AKS have been implemented using three classes in Java. The main class (see `AKS.java`) contains the algorithm. The first part of the code contains variables which correspond to the parameters of the algorithm, as well as a boolean variable, which can be set to `true` or `false` to switch between the extended and the original version of AKS. After some initializing steps, the pseudocode given in this paper is performed. Some (recurring) parts have been written as functions, such as the part in which the buckets are created and the part in which the root node is solved.

The other two classes represent the buckets and the decision variables. The bucket class (see `Bucket.java`) is mainly a way to store a set of variables. The variable class (see `Variable.java`) has been used, even though CPLEX already has an object associated with variables. However, writing a class for this allows us to store other information alongside of the variable, such as whether it is a kernel variable, what its value during the root node relaxation was, etc. Among others, this class also contains functions to restrict or unfix the variables.

Finally, a simple program consisting of only one class (see `SimpleCPLEX.java`) has been written to apply the basic version of CPLEX to the instances.

Appendix B: Performance of AKS(1h)

Figure 2 contains two additional graphs showing the relationship between the performance of AKS(1h) and different instance characteristics. Outliers with a gap of more than 5% in absolute values have been removed.

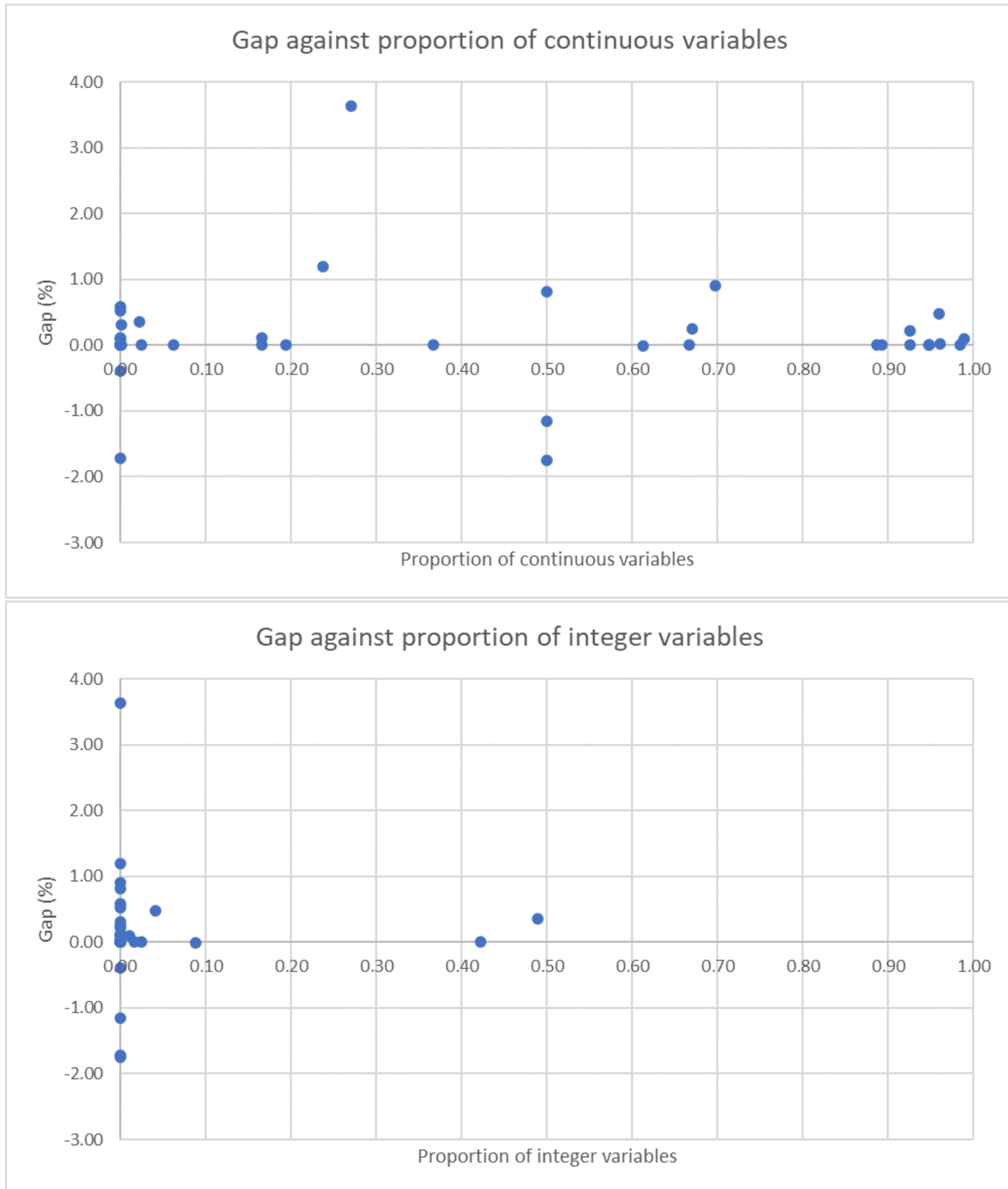

Figure 2: Performance of AKS(1h) for different instance characteristics (without outliers)

Appendix C: Full comparison with Guastaroba, Savelsbergh, and Speranza (2017)

Table 5: Detailed Results of CPLEX(1h) and AKS(1h) as compared to CPLEX(5h) and AKS(5h) from Guastaroba et al. (2017)

Instance	CPLEX(5h)		CPLEX(1h)		AKS(1h)				AKS(5h)	
	Objective	CPU (s)	Objective	CPU (s)	Category	Objective	Gap (%)	CPU (s)	Gap (%)	CPU (s)
arki001	7581212.5	8	7581043.7	8	easy	7580813.0	0.00	53	0.00	1369
danoint	65.7	234	65.7	463	easy	65.7	0.00	786	0.00	427
glass4	1200012600.0	459	1200012600.0	150	easy	1200012600.0	0.00	135	0.00	54
markshare1	2.0	18000	4.0	3600	easy	4.0	0.00	3600	-50.00	18000
markshare2	4.0	18000	12.0	3600	easy	11.0	-8.33	3600	0.00	18000
nsrand_ipx	51200.0	34	51200.0	73	easy	51200.0	0.00	451	0.00	578
rail2536c	689.0	18	689.0	35	easy	689.0	0.00	110	0.00	43
rail507	174.0	27	174.0	80	easy	174.0	0.00	1745	0.00	147
roll3000	12890.0	11	12890.0	15	easy	12890.0	0.00	54	0.00	35
sp97ic	427684487.7	3188	427684487.7	3109	easy	430191920.3	0.59	3600	0.32	5673
sp98ar	529740623.2	312	529740623.2	408	easy	530359351.7	0.12	3600	0.03	7619
sp98ic	449144758.4	172	449144758.4	123	easy	449144758.4	0.00	1368	0.02	7646
tr12-30	130596.0	71	130596.0	126	easy	130596.0	0.00	166	0.00	110
A1C1S1	11503.4	528	11503.4	1298	normal	11503.4	0.00	737	0.00	18000
A2C1S1	10889.1	307	10889.1	870	normal	10889.1	0.00	1285	0.00	18000
net12	214.0	69	214.0	139	normal	214.0	0.00	127	0.00	104
seymour	423.0	18000	423.0	3602	normal	423.0	0.00	3602	0.00	18000
sp97ar	661059959.7	18000	660729183.0	2934	normal	664229005.1	0.53	3600	0.32	180
UMTS	30090583.0	625	30090583.0	277	normal	30090420.0	0.00	1263	0.00	427
B1C1S1	24544.3	4888	24544.2	3600	hard	24544.3	0.00	3600	0.00	18000
B2C1S1	25687.9	16861	25812.0	3600	hard	25869.4	0.22	3600	0.09	18000
biella1	3065029.8	130	3065037.5	125	hard	3065005.8	0.00	1402	0.78	6743
mkc	-563.8	18000	-563.8	3606	hard	-527.3	6.49	266	0.00	18000
NSR8K	18148119.0	18000	857676860.7	3600	hard	800664386.3	-6.65	3600	4.78	18000
rail2586c	955.0	18000	953.0	3600	hard	955.0	0.21	3600	0.21	18000
rail4284c	1069.0	18000	1072.0	3600	hard	1073.0	0.09	3600	0.75	18000
rail4872c	1538.0	18000	1550.0	3600	hard	1544.0	-0.39	3600	0.52	18000
swath	467.4	18000	467.4	3603	-	594.3	27.15	3600	3.06	4816
van	4.6	18000	4.8	3600	hard	4.8	0.00	3600	0.00	18000
bg512142	184234.0	18000	188862.2	3601	hard	190569.0	0.90	3601	1.51	18000
dc1c	1778296.5	18000	1782289.6	3600	hard	1784332.4	0.11	1880	1.39	18000
dg012142	2549755.3	18000	2596277.0	3600	hard	2318648.0	-10.69	2162	1.54	18000
germanrr	47095891.1	18000	47095871.2	2054	hard	47266478.9	0.36	256	0.13	18000
germany50-DBM	473840.0	18000	474980.0	3600	hard	475410.0	0.09	2162	0.60	18000
go19	84.0	18000	84.0	3603	hard	84.0	0.00	3603	0.00	18000
ivu52	481.1	18000	490.1	3601	hard	481.7	-1.71	2181	0.12	15740
maxgasflow	-44562045.0	18000	-44565689.0	3600	hard	-44454998.8	0.25	1943	-0.01	18000
n3-3	15921.0	18000	16072.0	3600	hard	16150.0	0.49	1880	0.30	18000
neos-948126	2608.0	18000	2609.0	3600	hard	2704.0	3.64	1806	1.46	3580
circ10-3	382.0	18000	408.0	3600	hard	386.0	-5.39	3601	-2.62	18000
dano3mip	679.8	18000	691.8	3600	hard	692.0	0.02	3600	0.98	18000
dc1l	1781287.1	18000	1782855.4	3600	hard	1887273.6	5.86	3600	3.95	18000
ex1010-pi	240.0	18000	241.0	3600	hard	241.0	0.00	3600	1.25	18000
lectsched-1-obj	76.0	18000	83.0	3600	hard	83.0	0.00	2408	-3.95	18000
n370a	1261899.0	18000	1277711.0	3600	hard	1288180.0	0.82	3600	1.75	18000
n3700	1252509.0	18000	1280361.0	3600	hard	1257932.0	-1.75	3600	3.26	18000
n3705	1254168.0	18000	1281088.0	3600	hard	1266286.0	-1.16	3600	3.59	18000
neos-937815	2845.0	18000	2859.0	3600	hard	2893.0	1.19	1450	0.56	1008
ns1854840	316000.0	18000	2242000.0	3600	hard	198000.0	-91.17	3600	-49.37	7860
Average		**11958**		**2529**			**-1.59**	**2336**	**-1.48**	**11962**

Appendix D: Detailed Results

Table 6: Detailed results of CPLEX(1h), AKS(1h) and E-AKS(1h)

Instance	CPLEX(1h)		AKS(1h)				E-AKS(1h)			
	Objective	CPU (s)	Category	Objective	Gap (%)	CPU (s)	Category	Objective	Gap (%)	CPU (s)
arki001	7581043.7	8	easy	7580813.0	0.00	53				
danoint	65.7	463	easy	65.7	0.00	786				
glass4	1200012600.0	150	easy	1200012600.0	0.00	135				
markshare1	4.0	3600	easy	4.0	0.00	3600				
markshare2	12.0	3600	easy	11.0	-8.33	3600				
nsrand_ipx	51200.0	73	easy	51200.0	0.00	451				
rail2536c	689.0	35	easy	689.0	0.00	110		*equals AKS(1h)*		
rail507	174.0	80	easy	174.0	0.00	1745				
roll3000	12890.0	15	easy	12890.0	0.00	54				
sp97ic	427684487.7	3109	easy	430191920.3	0.59	3600				
sp98ar	529740623.2	408	easy	530359351.7	0.12	3600				
sp98ic	449144758.4	123	easy	449144758.4	0.00	1368				
tr12-30	130596.0	126	easy	130596.0	0.00	166				
A1C1S1	11503.4	1298	normal	11503.4	0.00	737				
A2C1S1	10889.1	870	normal	10889.1	0.00	1285				
net12	214.0	139	normal	214.0	0.00	127		*equals AKS(1h)*		
seymour	423.0	3602	normal	423.0	0.00	3602				
sp97ar	660729183.0	2934	normal	664229005.1	0.53	3600				
UMTS	30090583.0	277	normal	30090420.0	0.00	1263				
B1C1S1	24544.2	3600	hard	24544.3	0.00	3600	hard	24544.3	0.00	3600
B2C1S1	25812.0	3600	hard	25869.4	0.22	3600	hard	25869.4	0.22	3600
biella1	3065037.5	125	hard	3065005.8	0.00	1402	hard	3065005.8	0.00	3600
mkc	-563.8	3606	hard	-527.3	6.49	266	hard	-563.8	0.00	3600
NSR8K	857676860.7	3600	hard	800664386.3	-6.65	3600	hard	800664386.3	-6.65	3600
rail2586c	953.0	3600	hard	956.0	0.31	3600	hard	967.0	1.47	3600
rail4284c	1072.0	3600	hard	1073.0	0.09	3600	hard	1086.0	1.31	3600
rail4872c	1550.0	3600	hard	1544.0	-0.39	3600	hard	1563.0	0.84	3600
swath	467.4	3603	-	594.3	27.15	3600		*equals AKS(1h)*		
van	4.8	3600	hard	4.8	0.00	3600	hard	4.8	0.00	3600
bg512142	188862.2	3601	hard	190569.0	0.90	3601	hard	189999.5	0.60	3600
dc1c	1782289.6	3600	hard	1784332.4	0.11	1880	hard	1782196.2	-0.01	3600
dg012142	2596277.0	3600	hard	2318648.0	-10.69	2162	hard	2680326.0	3.24	3600
germanrr	47095871.2	2054	hard	47266478.9	0.36	256	hard	47095869.6	0.00	3501
germany50-DBM	474980.0	3600	hard	475410.0	0.09	2162	hard	475410.0	0.09	3600
go19	84.0	3603	hard	84.0	0.00	3603	hard	84.0	0.00	3603
ivu52	490.1	3601	hard	481.7	-1.71	2181	hard	481.7	-1.71	3600
maxgasflow	-44565689.0	3600	hard	-44454998.8	0.25	1943	hard	-44546688.4	0.04	3600
n3-3	16072.0	3600	hard	16150.0	0.49	1880	hard	16150.0	0.49	3600
neos-948126	2609.0	3600	hard	2704.0	3.64	1806	hard	2705.0	3.68	3600
circ10-3	408.0	3600	hard	386.0	-5.39	3601	hard	386.0	-5.39	3600
dano3mip	691.8	3600	hard	692.0	0.02	3600	hard	692.0	0.02	3600
dc1l	1782855.4	3600	hard	1887273.6	5.86	3600	hard	1887273.6	5.86	3600
ex1010-pi	241.0	3600	hard	241.0	0.00	3600	hard	247.0	2.49	3600
lectsched-1-obj	83.0	3600	hard	83.0	0.00	2408	hard	83.0	0.00	3600
n370a	1277711.0	3600	hard	1288180.0	0.82	3600	hard	1284461.0	0.53	3600
n3700	1280361.0	3600	hard	1257932.0	-1.75	3600	hard	1261935.0	-1.44	3600
n3705	1281088.0	3600	hard	1266286.0	-1.16	3600	hard	1275716.0	-0.42	3600
neos-937815	2859.0	3600	hard	2893.0	1.19	1450	hard	2855.0	-0.14	3600
ns1854840	2242000.0	3600	hard	198000.0	-91.17	3600	hard	198000.0	-91.17	3600
Average		**2529**			**-1.59**	**2336**			**-2.97**	**3597**