How can the behaviour of specialized heuristic solvers assist constraint solvers for optimization problems

A lookahead approach for Chuffed that emulates the behaviour of heuristic solvers

by

Angelos Zoumis

Student Name	Student Number
Angelos Zoumis	4921771

Thesis advisor: N. Yorke-Smith
Daily Supervisor: E. Demirović
Project Duration: November 2022 - August 2023
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

TU Delft

Abstract

Constraint programming solvers provide a generalizable approach to finding solutions for optimization problems. However, when comparing the performance of constraint programming solvers to the performance of a heuristic solver for an optimization problem such as cluster editing, the heuristic solver is able to find near-optimal and optimal solutions much faster. The goal of this research is to understand how the behaviour of such a heuristic solver can assist the performance of constraint programming solvers in optimization problems. In order to achieve this, first Chuffed [8], a state-of-the-art constraint programming solver was combined with a heuristic approach to cluster editing, with the goal of emulating the performance of the heuristic algorithm, in particular, being able to find near-optimal solutions faster. Continuing, the goal was to generalize the behaviour observed by the modified solver, by emulating the performance observed without the need for the specialized heuristic solver. The generalized approach is tested on a wide variety of different tests. An approach to value selection was developed that performs lookahead propagations for the two values of a boolean variable and selects the value that has the most optimal solution within the domain after performing the lookahead propagation. This approach added a significant time overhead that increased the overall solving time for many problems, with the lookahead configuration having a median increase of 8.7% over the default configuration for the optimization problems of the MiniZinc Challenge 2022 [24]. However, it was able to successfully emulate the performance of the heuristic solver, finding near-optimal solutions significantly faster than the default value selection. In particular, for the optimization problems of the MiniZinc Challenge 2022 [24], on average, the lookahead configuration had a definite integral for the time vs objective graph 54.7% lower than the default Chuffed configuration.

Contents

1 Introduction

Constraint programming (CP) is a powerful approach for solving a wide range of problems involving constraints. It deals with the modelling of problems with variables that are subject to constraints, and solving them by providing a set of assignments to the variables, such that all constraints are satisfied. Furthermore, for optimization problems, the variables also have to maximize an objective function [27]. One of the main advantages of constraint programming is that it only needs a model of the problem to work, instead of a tailor-made algorithm, making them extremely versatile and generalizable.

Constraint programming has been applied to a wide range of real-world problems, including scheduling, planning, resource allocation and configuration, and design problems, for many different fields, such as bioinformatics, finance, telecommunications, engineering, and transportation [31, 29]. For example, in scheduling, a model can be created that takes into account a wide range of constraints, such as resource and personnel availability, and temporal constraints, while having an objective function, like minimizing costs. The use of such a model can help better clarify what the conditions that need to be met are, and allows for expandability, as adding additional variables and constraints does not require modifying other parts of the model. A constraint programming solver can take as an input the said model and can be used to find optimal, or near-optimal schedules that satisfy all constraints. Therefore, solving, but also finding near-optimal answers for CP problems as quickly as possible is of great interest to many companies that rely on CP solvers.

1.1. Background

The need for more efficient CP solvers has led to a race where different solvers constantly improve year over year [24], with the goal of creating more efficient solvers. In particular, most solvers attempt to reduce the search space as much as possible and use various heuristics to better guide the solver through the search space.

Recently, CP has grown closer to propositional satisfiability (SAT), another NP-Hard problem. SAT problems take as input a propositional formula, which is constructed with boolean variables and basic logic operations, in particular AND, OR, and NOT. The goal of SAT solvers is to find an assignment of all variables that make the given propositional formula true.

Lazy Clause Generation (LCG) provides a hybrid of CP-SAT, by converting the CP variables, for example, integers, into sets of boolean variables and associated clauses. Two state-of-the-art solvers that utilize such a hybrid CP-SAT approach are Chuffed [8] and OR-Tools [15], the latter having won gold in the MiniZinc challenge every year since 2013. In addition, many heuristics have been used, such as variable state independent decaying sum (VSIDS) and Solution Based Phase Saving (SBPS), with the goal of further improving the performance of the solvers.

1.2. Motivation

Despite many improvements, CP solvers still struggle to find optimal, or near-optimal solutions at a reasonable time for certain problems, in particular, when compared to heuristic solvers. Figure 1.1 showcases this performance disparity. For the problem of cluster editing, which will be the main focus

of this research. When attempting to solve the same instance, Chuffed is able to find the optimal solution of 42 in 56 seconds. In comparison, the heuristic solver finds the same solution in 6 milliseconds.

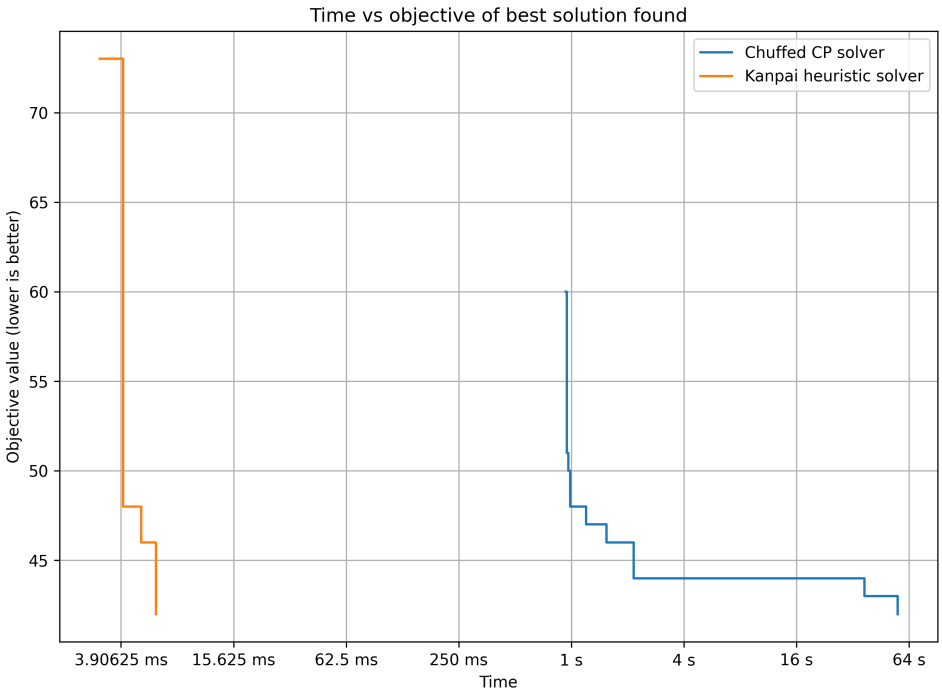


Figure 1.1: Performance disparity between the Kanpai heuristic algorithm[10] and Chuffed CP solver[8] for the instance of the cluster editing problem.

This performance disparity can be partially explained by the fact that a heuristic solver is specialized for a specific problem, and therefore, has additional knowledge of the structure of the problem, and how to achieve a near-optimal solution. Furthermore, looking at some heuristics used by CP solvers such as Chuffed, in particular VSIDS and SBPS, these heuristics provide little assistance early on and only have a significant effect on the solver after finding some solutions and conflicts.

Although approaches exist that utilize heuristic solvers, such as using them for warm start[14], this only provides an initial solution to the CP solver and still keeps the CP solver separate from the heuristic solver. Being able to learn what heuristic solvers does differently, and emulating this behaviour to a CP solver could help decrease the gap between CP and heuristic solvers, especially in the early stages of computation.

1.3. Research goal

CP-SAT solvers rely on various heuristics to make decisions during branching that will improve the overall performance of the solver. In particular, the main steps in branching are selecting a variable, and deciding on a value. The goal of this research is to modify the value selection of the VSIDS branching heuristic for Chuffed, in an attempt to improve the performance of the solver, particularly early on.

In order to achieve this, the Cluster Editing problem will be used as a case study. The goal of cluster editing is for a given undirected graph, to find the minimum number of edge edits, which includes edge addition or removals, that transform the graph into a cluster graph. A cluster graph is a graph where every connected component forms a clique [3].

Finding the optimal solution for this problem is NP-hard, however, just finding a solution that forms a cluster graph is trivial. Simply put the nodes in any arrangement of clusters, and record the edits that need to occur in order to achieve the desired cluster graph. As a result, heuristic algorithms are able to find a solution for cluster editing relatively quickly, and then attempt to find near-optimal or optimal

solutions through many different methods [3], like local searching, or using a set of heuristics.

Therefore, the first step of the research will be modifying the Chuffed solver, so that the value selection will be determined by a heuristic algorithm specialized for solving cluster editing instances. Two approaches are implemented for this step. The first approach fully solves the problem and returns the value assigned to the selected variable. This approach is similar to a warm start, but it keeps providing assistance throughout the entire progress of the solver, instead of just the very start. The second approach will only perform one step of the heuristic algorithm. This approach should show how well can the CP solver utilize intermediate solutions by the solver. Based on the previous steps' findings, the goal will become to emulate the performance noticed, without the need for the heuristic solver. First, the new modified solver will be tested on the Cluster editing problem, and continuing, will be tested on different optimization problems, in order to discover how generalizable are the findings of this research.

1.4. Research questions

Based on the aforementioned research goals, the main research question is **How can the behaviour of specialized heuristic solvers assist constraint programming solvers in optimization problems?** This research question can be divided into the following subquestions:

1. **Can Chuffed be combined with a cluster editing heuristic algorithm to improve the performance of the CP solver on cluster editing instances?**
2. **How can the performance of the combined algorithm be emulated without the need for the heuristic algorithm?**
3. **How does the modified solver perform in different optimization problems?**

1.5. Roadmap

The thesis will be organized as follows: In chapter 2, a description of the related work and state-of-the-art in the relevant topics is provided. Next, chapter 3 provides a preliminary section of the problems relevant to the research, in particular, cluster editing and constraint programming, detailing the concept of the problems, as well as relevant notations and step-by-step examples for concepts that will be of high importance during the research.

Continuing, the following two chapters address the first research sub-question. First, in chapter 4, a description of how Chuffed can be combined with a heuristic solver is provided. Continuing, in chapter 5, the experimental setup to test the performance of the combined solver on cluster editing is detailed, along with the results of the experiments and an analysis of the performance and generalizability of these findings.

Next, the following two chapters address the second and third sub-questions. In particular, an approach to emulate the performance of the combined algorithm is presented in chapter 6.

Chapter 7 presents the test results of the lookahead approach. First, in section 7.2, the cluster editing performance of the new approach is tested, using an identical experimental setup as the previous tests, in order to see how closely this approach emulates the previous approaches, and how it affects the overall performance of the solver on cluster editing. Sections 7.3 and 7.4 address the last sub-question, with the lookahead approach being tested using a set of different problems and instances. Lastly, chapter 8 discusses the main conclusion, along with future improvements.

2

Related Work

This chapter introduces the main concepts, along with related work in the fields of constraint programming and cluster editing heuristic solvers.

2.1. Constraint programming

Before even the emergence of modern computers, the general concept of constraint satisfaction and optimization already existed. The real-world uses for constraint satisfaction and optimization, like scheduling or routing have always been problems in need of a solution. In fact, backtrack searching, which is the concept of building a solution iteratively, by branching through a search tree and removing solutions that cannot satisfy all constraints was used in recreational Mathematics in the nineteenth century [22], as cited in chapter 2 of the handbook of constraint programming by F. Rossi et al. [27].

In general, the field of constraint programming can be split into two sub-fields. The first field focuses on the languages used to model constraint problems and the second one has an interest in constraint-solving algorithms.

2.1.1. Modelling

The core of constraint programming languages is based on variable relations and formal logic. One of the first uses of constraints in a programming language comes from M.V. Wilkes, where programming language statements are included, which are not explicitly assignment statements, but relations between variables that need to be met. Prolog [9] was one of the first declarative programming languages based on formal logic statements, hence the name PROgramming LOGic. Prolog is therefore considered an early constraint programming language [27].

Nowadays, one of the state-of-the-art constraint modelling languages is MiniZinc [23], a high-level modelling language for constraint satisfaction and optimization problems. MiniZinc is solver-independent, by compiling into a low-level language, FlatZinc, which is understood by a wide range of solvers. Due to its availability, efficiency and ease of use with any solver, MiniZinc will be used as the modelling language throughout the research.

2.1.2. CP Solvers

The algorithms focused on solving constraint problems are generally split into two strategies: searching or branching, and inference or propagation [27].

Searching is the task of navigating through the search tree of a problem, in order to find a solution satisfying all constraints. Backtracking is the fundamental search method for CP problems, as it guarantees to find a satisfactory solution if one exists [27]. Backtracking builds up a partial solution, by selecting values for variables until reaching a solution, or a conflict, at which point, it backtracks to a previous decision where a different choice can be made, and makes a different decision. This approach potentially visits all feasible partial solutions and hence guarantees to find any existing satisfying solutions. This approach is better than brute force, as it checks after each decision if the constraints are met, instead of doing that only until a full solution is found. Hence, discovering that a partial

solution cannot satisfy all constraints prunes the subtree of that partial solution, resulting in multiple full solutions that do not satisfy all constraints being removed from consideration.

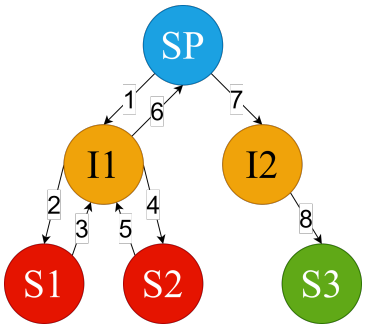


Figure 2.1: An example of backtracking. The starting point is shown in blue. The orange nodes represent an intermediate solution, the red nodes represent a solution with conflicts, and the greed nodes represent a solution satisfying all constraints. The arrows indicate decisions and backjumps

Figure 2.1 shows an example of backtracking. The solver starts with an empty solution and first makes a decision that moves its state to an intermediate solution (I1). Continuing, the solver makes a second decision, which leads to a solution with conflicts (S1). After backtracking, back to I1, the solver now makes a different decision, which also leads to a solution with conflicts. Backjumping back to the starting point, the solver now makes a decision that results in a different intermediate solution than I1. From I2, the solver makes one last decision that results in the solution satisfying all constraints. This process has explored all possible solutions, before finding a solution.

Propagation is a vital step in CP solvers, reducing the search space of candidate solutions, and enabling solvers to work more efficiently, especially as the problem size increases. By default, using backtracking to solve constraint satisfaction problems almost always leads to thrashing behaviours [27, 6]. Thrashing is the repeated exploration of sets of subtree modules that are failing due to the same assignments and only differ in assignments that are not related to the cause of failure. As the size of the problem increases, and hence, the size of the search space increases exponentially, the effect of thrashing in the total runtime also increases exponentially. Propagation can significantly reduce thrashing, by making implicit constraints explicit, and removing values from the domains of variables that are not consistent, meaning that these values produce a non-satisfiable solution.

2.1.3. Hybrid CP-SAT approach

Another field of study similar to constraint programming is boolean satisfiability (SAT), which attempts to satisfy formulas consisting of boolean variables and boolean operators. These two fields of study have grown increasingly closer, in particular after researchers discovered that CP and SAT instances can be mapped to each other [30, 7]. Due to SAT solvers being extremely efficient, encoding certain CP problems as SAT problems could lead to great performance gains.

P.J. Stuckey [28] presents a hybrid CP-SAT approach where CP variables and constraints are mapped to variables and clauses in an SAT solver. This approach utilized a technique called Lazy Clause Generation(LCG). For example, an integer variable with a finite domain $[l..u]$ would be represented by a set of boolean variables like $x \leq d, l \leq d < u$ and $x == d, l \leq d < u$.

The LCG technique combines finite domain propagation of CP solvers with the highly efficient inference graph of SAT solving that records nogoods path and prevents similar failing parts of the problem from being explored. Solvers such as Chuffed[8], utilize these techniques in order to create highly efficient solvers.

2.1.4. CP Heuristics

One of the main ways of improving the branching performance was through the use of heuristics, which better guide the solver through the search tree. One of the earlier concepts of heuristics was using a lookahead procedure called forward checking, which employs the **most likely to fail** principle, which branches on decisions that are more likely to fail [17]. This heuristic was shown to perform better than standard backtracking.

For LCG solvers such as Chuffed, using SAT branching heuristics, such as the Variable State Independent Decaying Sum (VSIDS) heuristic, can further improve the variable selection[21]. VSIDS works by assigning an activity score to each variable and increasing said score based on how many clauses that variable is involved in. During branching, the variable with the highest activity is selected. Through the use of additive bumping and multiplicative decay, a bias towards variables that have a greater presence in recently learnt clauses is created. Due to VSIDS being independent of the current state of assignments, backtracking does not require any changes to the activity score, making this heuristic incredibly efficient.

Solution-based phase saving (SBPS) emulates local search, by attempting to search through the neighbourhood around the current best solution [11]. This is done by setting the default polarity for each variable to the value of the best solution found so far. This approach is useful for optimization problems, finding more optimal solutions faster than the default value selection for certain problems.

Additional approaches that improve the performance of Chuffed using Machine learning exist, such as initializing an activity score for VSIDS [12] or predicting unsatisfiable cores [13].

Overall, many techniques have been used to make CP solvers more efficient for both satisfaction and optimization problems. New techniques are constantly being implemented that further improve the performance of solvers in certain areas. However, for optimization problems, better guiding the SAT value selection for Chuffed, especially in the early stages could potentially lead to finding near-optimal solutions faster.

2.2. Heuristic approaches to Cluster editing

Cluster editing is the problem of finding a set of edge modifications that transform a graph into a cluster graph. A more formal description of this problem is provided in section 3.1.

Finding a solution to the Cluster editing problem is NP-hard, with a time complexity of $O(2.27^k * |V|^3)$ using a branch and bound strategy [16].

From the PACE 2021 challenge [19], the top 9 heuristic algorithms that competed were all able to achieve a score above 99/100 on the 200 instances, with sizes (nodes + edges) of up to five million. The score is calculated by $100 * s_{min}/s$, with s_{min} being the best solution known for a problem, and s the solution produced by the heuristic solver. This indicated that the heuristic solvers were able to find optimal, or near-optimal solutions for most instances. In particular, the top-scoring solver for both the exact and heuristic track, KaPoCE [5] was able to achieve a score of 99.9989/100.

The main algorithm used in this research for cluster editing is the Kanpai approach [10], which was the 5th best solver in the heuristic track of PACE 2021, with a score of 99.9786/100 [19]. This algorithm uses a bottom-up approach to solving this problem. At first, each node is in its own cluster. Next, the algorithm iteratively combines nodes into clusters in a greedy fashion. In particular, if connecting two nodes would decrease the number of nodes, then the two nodes are added to the same cluster. Once a local minimum is reached, nodes in the same cluster a replaced by super-nodes, and the process continues. In order to ensure that the program does not terminate after reaching a local minimum early, it randomly makes sideways decisions if it can not find an improving move. This algorithm is able to make big reductions in the minimum number of edits in the first few seconds of running, even from some of the larger instances of the PACE 2021 heuristic track [19]. Due to its simplicity, yet high performance, this algorithm is a perfect candidate for an attempt to emulate its behaviour with Chuffed.

3

Preliminary

In this chapter, a problem description and notations in relation to cluster editing and constraint programming, as well as step-by-step examples of the Kanpai cluster editing algorithm and a barebones CP solver are presented.

3.1. Cluster editing

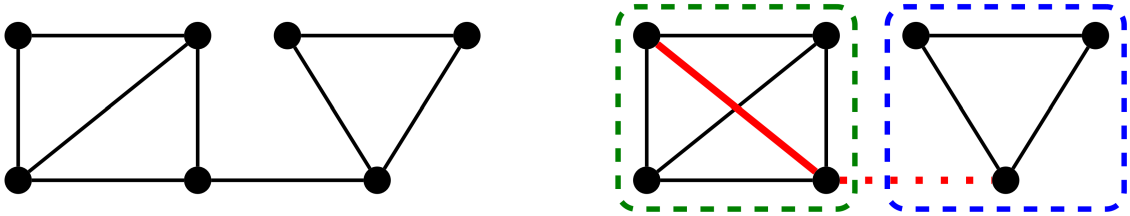


Figure 3.1: Left: An input graph. Right: A cluster graph is obtained by making two edge modifications in the input graph. The thick red line is an added edge, and the dashed red line is a removed edge. Figure sourced from PACE 2021 [19].

Cluster editing, also known as correlation clustering, is a problem with many applications in bioinformatics [4], data mining [1], and psychology [18]. It is a form of graph-based clustering, that takes as an input a graph $G = (V, E)$ that attempts to find the minimum set of edge modifications, which includes edge insertions or deletions, that transforms a given graph into a cluster graph, a graph where every connected component is a complete graph (clique). There are different variations of the cluster editing problem, with different weights per edge, to represent values such as node similarities [2]. For this research, the variation of cluster editing used will be the one used for PACE 2021 [19], which uses graphs with unweighted edges. Figure 3.1 shows an example of cluster editing, with the left graph representing an example of an input graph, and the right graph showing the optimal solution for cluster editing for this example.

Unlike other common clustering techniques, cluster editing finds a cluster graph that is the most similar to the input graph, without requiring the final number of clusters to be provided as input.

Finding an assignment of edge modifications that transforms a graph into a cluster graph can be done in polynomial time. One example of a non-optimal cluster editing solution is simply removing all edges. This results in a cluster graph, as each node is not connected to any other node, and hence, forms a clique by itself. However, finding the set with the minimum number of edge modifications and proving optimality is NP-Hard [19].

3.1.1. Cluster editing notation

Let G be an undirected and unweighted graph. V_G represents the set of vertices of a graph G and E_G represents the set of edges. $\{u, v\} \in E_G$ represents an existing edge in G between $u \in V_G$ and $v \in V_G$.

Two nodes $u, v \in V_G$ are adjacent, or neighbors, if an edge $\{u, v\} \in E_G$ between them exists. The neighbourhood of a node $u \in V_G$ is the set of all neighbours of u and is denoted as $N_G(v)$. The closed

neighbourhood of a node u includes the neighbourhood of the node plus itself, and is defined as such: $N_G[u] = N_G(u) \cup u$

The adjacency matrix of a graph G is a $[|V_G|, |V_G|]$ matrix, where each row and column represents each node, and each element, being 1 or 0 (true or false) represents whether the two nodes are adjacent. In the following sections, the input graph will be denoted as G, and the output graph as g.

3.1.2. Step by step Kanpai algorithm

In this section, a more formal explanation of the Kansai algorithm [10] will be explained, through the use of an example, along with important concepts of the algorithm, in particular, the super-nodes, which will be relevant in the following sections.

$$G = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

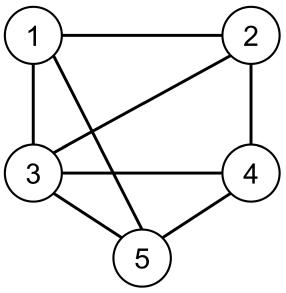


Figure 3.2: Input graph for the kanpai algorithm.

For this example, the input graph is shown in figure 3.2. This graph has 5 nodes and 8 edges, with the minimum number of edits being two, achieved by adding an edge between 1 and 4, and 5 and 2, essentially transforming the graph into a complete graph. Initially, the Kanpai algorithm will start with each node being in its own cluster, and hence, the number of edits will be 8.

In each iteration, the algorithm will go through each node, and move it to a cluster that reduces the number of edits the most. This is referred to as the best local move. If no moves exist that improve the number of edits, the algorithm might randomly perform a sideways move if one exists, a move that does not change the number of edits.

$$g = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

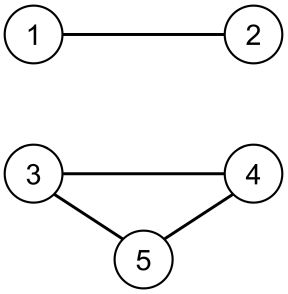


Figure 3.3: An intermediate solution produced by the kanpai algorithm. At this state, any move will be at best a sideways move.

Figure 3.3 shows the potential state of the algorithm after one iteration. In this state, the number of edits is 4. There are no moves that can improve the number of edits, and at best, only sideways moves can be made. Let's say an attempt is made to move edge 1 to the cluster with 3,4 and 5, an edge will have to be removed between 1 and 2, leading to 4 edits.

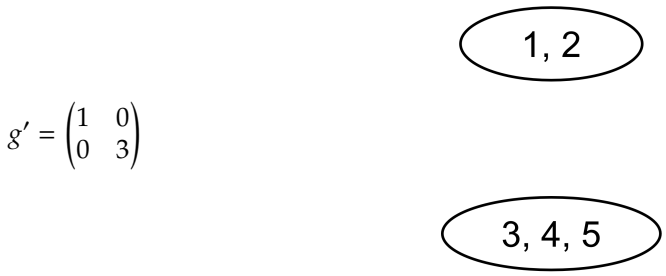


Figure 3.4: The algorithm adds the existing clusters into super-nodes.

After not being able to improve the number of edits, the algorithm will add all current nodes into super-nodes, as shown in figure 3.4. In this state, the previous clusters are treated as nodes. Furthermore, the graph now acts as a weighted graph, in order to account for the fact that multiple edges are needed to connect the two super-nodes.

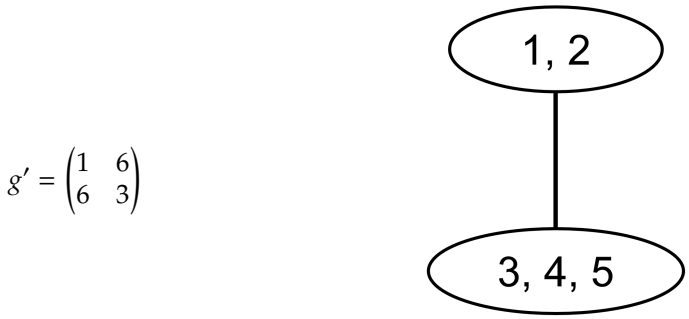


Figure 3.5: The final result of the kanpai algorithm, after connecting the two super nodes.

In this iteration, connecting the two super nodes, as shown in figure 3.5, improves the number of edits to 2. This also happens to be the best solution, although this algorithm does not guarantee to find the optimal answer.

3.2. Constraint programming

A constraint programming solver takes as an input a model of a problem and attempts to produce a solution, based on the model's parameters. A constraint optimization model consists of 4 parts:

- **Variables:** The main part of a CP model is the set of variables that should be assigned a value.
- **Domain:** The domain of possible values that the variable can be assigned. Any assignment of values within the domain of each variable is a candidate solution.
- **Constraints:** Each constraint is a condition that needs to be satisfied.
- **Objective function:** Optionally, a model can have an objective function that needs to be maximized or minimized based on the problem. The goal of the research is to improve the performance of CP solvers in optimization problems. Therefore, each used model will always have an objective function.

The goal of a CP solver is to find a solution within the search space of the model, the search space including all candidate solutions, that satisfies all constraints, and optimizes the objective function. In general, to achieve this, a typical CP solver roughly performs the following two steps in each iteration: propagation and branching. Propagation updates the domains of all variables, removing values that would violate the constraints of the problem. Branching decides which part of the tree to explore next. Many strategies and heuristics are used to better guide the CP solver during this phase.

For cluster editing, a CP model could be created as such. The variables are the adjacency matrix of the output cluster graph. Essentially, the edges in the output graph are the variables. The constraints indicate that if two nodes are connected in the output graph, then they must have the same neighbours,

i.e. be connected to the exact same nodes, else, they must share no neighbours. Lastly, the objective function is to minimize the number of edits. For this model, the goal of a CP solver would be to find an assignment of variables for the adjacency matrix that satisfies the constraint and prove that no other assignment of the adjacency matrix produces a lower amount of edits.

3.2.1. CP solver steps

For this section, a simplified view of the steps taken by a CP solver, like Chuffed, will be analyzed and explained. The focus will mainly be on the branching decisions of the solver, while also keeping in mind how variables are propagated.

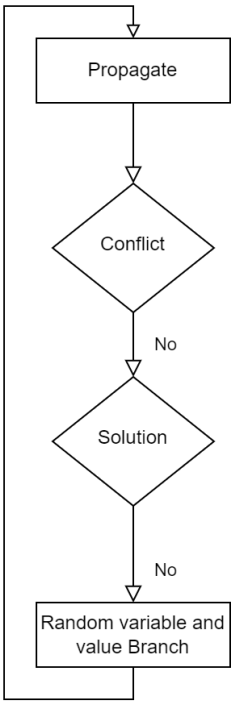


Figure 3.6: An example of the main components of a simple constraint programming solver.

Figure 3.6 shows the barebones solver that will be used for this step-by-step example. At first, the solver propagates all variables, updating their domain to only include values that satisfy the constraints. Assuming there are no conflicts or a full solution, the solver takes the current state after the propagation, and branches on a new variable and value. For this example, it is assumed that branching is random.

$$g = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

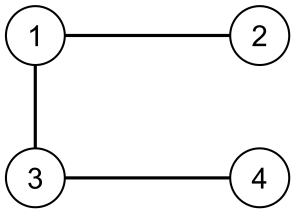


Figure 3.7: Input graph for Chuffed.

A step-by-step showcases how these steps are relevant to cluster editing. Figure 3.7 shows an input graph with 4 nodes and 3 edges. The optimal solution requires 1 edit, where the edge between 1 and 3 is removed.

$$G = \begin{pmatrix} 0 & 0..1 & 0..1 & 0..1 \\ 0..1 & 0 & 0..1 & 0..1 \\ 0..1 & 0..1 & 0 & 0..1 \\ 0..1 & 0..1 & 0..1 & 0 \end{pmatrix}$$

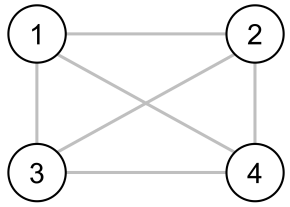


Figure 3.8: Initial state of the solver.

At first, the solver will start with all variables empty,as shown in figure 3.8. At this point, the domain for the number of edits is from 0 to 6. If edge $1, 2 \in E_g$ is selected as the first variable, and assigned it 1. After propagation, the domain for the number of edits is updated to 0..5.

$$G = \begin{pmatrix} 0 & 1 & 1 & 0..1 \\ 1 & 0 & 1 & 0..1 \\ 1 & 1 & 0 & 0..1 \\ 0..1 & 0..1 & 0..1 & 0 \end{pmatrix}$$

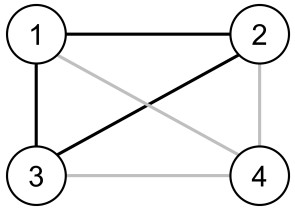


Figure 3.9: Intermediate state of the solver.

In the second iteration, selecting variable $1, 3 \in g$ and assigning 1 will propagate edge $1, 2 \in E_g$ to 1, due to the constraints of cluster editing. Next, the number of edits is propagated to 1..4. The resulting state is shown in figure 3.9. From this point, the solver will at best find a solution with 2 edits first before finding the optimal solution. This is because if node 4 is connected to the existing cluster of 1, 2 and 3, then the resulting solution will be 3 edits, whereas if 4 is not added to the cluster, the resulting solution will be 2. Chuffed will restart the process and reattempt to find a solution, having learned additional clauses that will assist in finding the optimal solution. However if in step 2, a different decision was made, in particular, selecting value 0 for edge $1, 3 \in g$, the solver would find the optimal solution faster.

4

Hybrid CP-Heuristic solver approach

This chapter will detail the first steps of the main contributions of the research. in particular, the cluster editing model used, continued by the Chuffed modifications done to integrate the heuristic approach to cluster editing with the CP solver.

4.1. Cluster editing Constraint model

In this section, the MiniZinc constraint model for cluster editing is presented, which will be used for testing the performance of the modified chuffed solver.

The model is based on a viewpoint of the adjacency matrix of the output cluster graph g. As an input, the model requires the number of nodes V_G, the number of edges in the original graph E_G, and a list of size $|E_G|$ that contains each edge $\{u, v\} \in E_G$. The variables of this model include the adjacency matrix of the final cluster graph g. The adjacency matrix is a $[|N|, |N|]$ matrix, with each variable being a Boolean. Next, the objective function is minimizing the number of edits. The main constraint of this model says that if edge $\{u, v\} \in E_g$ exists, then $N_g[u] = N_g[v]$, else, $N_g[u] \cap N_g[v] = \emptyset$. In other words, this constraint ensures that if two nodes are connected, then they must have the exact same neighbours, and if they are not connected, they must not share any neighbours.

In order to improve the performance of the model, additional optional constraints that limit the search space are included. These constraints are based on the lemmas presented in the paper Cluster Editing with Diamond-free Vertices by D. Rhebergen [26], and rely on the closed neighbourhood of the original graph. In particular, they ensure the following two properties:

1. If two nodes $u, v \in V_G$ are not adjacent in G, and $|N_G(u) \cup N_G(v)| \leq 1$ then the two nodes will not share an edge in the final cluster graph.

2. If two nodes $u, v \in V_G$ are not adjacent in G, $u, v \in E_g$ will exist only if $|(N_g(u) \cap N_g(v)) \cap (N_G(u) \cap N_G(v))| \geq 2$.

These constraints significantly reduce the search space for many instances, resulting in significantly faster performance. For certain instances with many possible optimal solutions, some optimal solutions are also excluded from the search space, however, there is always at least one solution in the search space that is optimal.

4.2. Chuffed solver modifications

This section describes the modifications done to the Chuffed solver in order to combine the solver with the heuristic algorithm.

4.2.1. Value selection modifications

The following two modifications affect the $SAT :: branch()$ function of Chuffed. In particular, in this function, if a variable of the adjacency matrix of g, let's say $u, v \in E_g$ has been selected, a heuristic

approach is used to determine the value of this variable. In other words, if u and v belong in the same cluster, and hence, share an edge in the final cluster graph.

First is a simple heuristic value selection. After selecting variable $u, v \in E_g$, this heuristic approach calculates how the number of edits changes if u is added to the cluster of v. If the number of edits decreases then the value selection returns $true$, adding an edge between the two nodes. Essentially, this performs one step of the Kanpai algorithm. This approach will be referred to as the simple heuristic value selection.

Continuing to the second approach, during value selection, the Kanpai algorithm solves the problem and returns the value that variable $u, v \in E_g$ has in the solved state of the heuristic algorithm. This approach will be referred to as the Kanpai value selection.

For both approaches, the current state of the adjacency matrix of g is passed to the heuristic algorithms. Before starting the computations, the state of the heuristic algorithm copies the assigned values of the passed variables, in order to reflect the state of the solver. Continuing, it is important to assume that the state of the assigned variables in the solver will remain the same. Therefore, the clusters that already exist in the current state of the solver are added into super-nodes, ensuring that the heuristic algorithm keeps the state of the solver intact at the end of its computation.

4.2.2. Initial activity for VSIDS

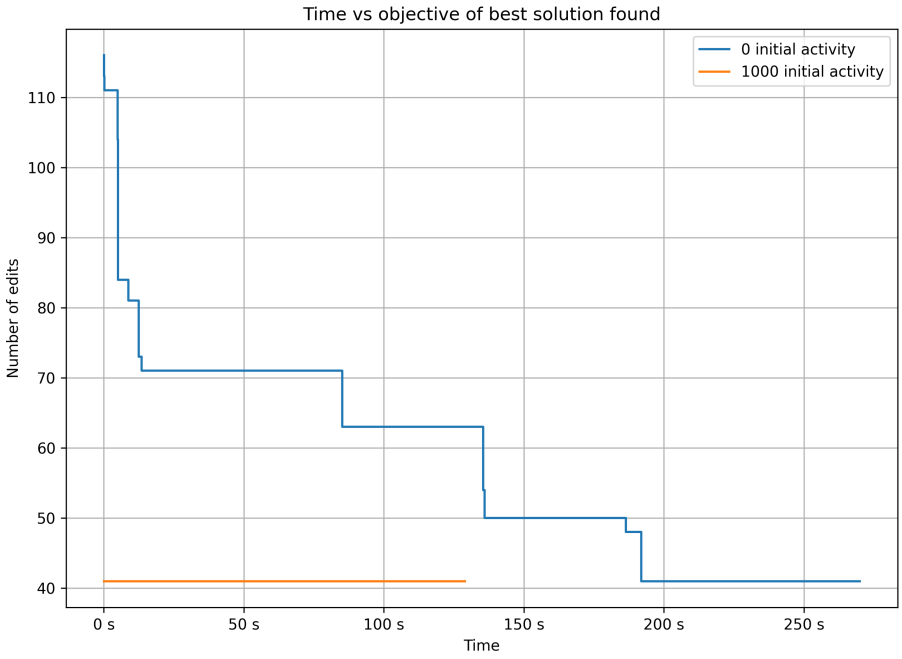


Figure 4.1: This graph shows the time against the best solution found on a cluster editing instance, for a solver using 0 initial activity, and one using 1000 initial activity. Both configurations use the Kanpai value selection

VSIDS starts with the same activity for all variables. As there are many variables that are not part of the adjacency matrix but propagate the adjacency matrix variables, this resulted in the Chuffed solver rarely using the heuristic approaches for value selection, leading to no significant performance changes. As a result, the initial activity for the adjacency matrix variables was increased, in order to force the solver to use the heuristic approach for value selection, especially in the earlier stages. This has a significant effect on performance, particularly in finding the best solution earlier, as shown in figure 4.1. Using the Kanpai value selection, a configuration with 0 initial activity is able to find an optimal solution in 192 seconds, while a configuration with 1000 initial activity for the variables of the adjacency matrix, g reaches the optimal solution in 0.03 seconds.

4.2.3. Warm start

During early testing, the modified solvers, and in particular, the Kanpai value selection was able to find the optimal solution almost immediately and spent the rest of the solve time simply proving optimality. Therefore, another approach is designed. This approach starts by using the Kanpai approach, but during variable selection, it only picks variables of the adjacency matrix first. When a solution is found the solver reverts the normal variable and value selection.

4.2.4. Solution based phase saving [11]

An additional version of the two heuristic value selection algorithms is also implemented that uses an approach similar to solution-based phase saving (SBPS) [11], which uses the best solution found so far during value selection.

This approach alongside the current state of the adjacency matrix of g, also passes the adjacency matrix of g of the best solution found so far. After copying the current state of the adjacency matrix of g and creating super nodes, the heuristic algorithm copies the adjacency matrix of the best solution found so far by Chuffed, while ensuring that no conflicts are created. However, these new clusters are not added into super-nodes, meaning that the solver does not have to keep the state of these clusters intact. For variables not in the adjacency matrix of g, SBPS is used.

The following chapter will test the performance of these modifications. In particular, the metrics used will represent the to total time taken to fully solve cluster editing instances and the quality of solutions produced at each time point.

5

Initial Experimental Results

In this chapter, the experimental setup is described. Continuing, the initial results are presented, comparing the performance of Chuffed using a cluster editing heuristic solver for value selection. First, the data collected during the experiment are displayed, and next, an analysis detailing the performance of each configuration, as well as how their benefits can be generalized is detailed. The full results are shown in appendix A.

5.1. Experimental Setup

This section describes the overall setup used through the experimental process, including the selection of instances, the changes between solver configurations, and the metrics used to evaluate performance.

Overall, 20 different graph instances were used to compare the performance of 8 different solver configurations. The model presented in section 4.1 was used, in combination with MiniZinc Python to run the tests.

5.1.1. Selection of Instances

For the selection of instances, in order to better represent the overall performance of the new modifications for cluster editing, a wide range of instances had to be selected, with different sizes. Therefore 20 different graphs were used, 10 graphs that are solvable by the default chuffed solver in under 10 minutes, and 10 larger graphs with a timeout of 10 minutes. The 20 graphs are either taken directly from instances of graphs used for the exact track of PACE 2021 or generated based on the tools used to generate said graphs [19].

For the small graphs, it was deemed important to select instances that are large enough for any differences between runs to be because of the different configurations. Furthermore, it was also necessary to select instances that are solvable multiple times in a reasonable amount of time. Therefore, instances that are solvable in between 30 seconds to 10 minutes by default chuffed were selected.

In total 10 different small instances were used. In particular, the instances exact003 and exact005 were used from pace2021 [19]. The rest of the instances were generated based on the tools provided by pace2021 [19], section 3.2, which transform weighted graphs into unweighted ones. In particular, 4 instances were generated using the real-world biological dataset [20]. Another 4 instances were generated using randomly created action sequences. Table 5.1, presents details about the 10 small graphs.

Continuing, 10 additional larger instances of increasing size were used. The goal of these instances is to compare the early performance of the different Chuffed configurations and observe how each configuration scales as the size of the graph increases. The graphs were all taken from the exact track of pace 2021[19]. For each large graph, a time limit of 10 minutes is set. Table 5.2 presents the large graphs used.

Table 5.1: Small graphs

Graph	Nodes	Edges	Optimal
exact003	20	73	42
exact005	20	97	46
instance_nr_11-csv-thres-0-40	22	141	39
instance_nr_313-csv-thres-0-45	20	107	42
instance_nr_1243-csv-thres-0-50	26	97	46
instance_nr_1679-csv-thres-0-45	24	154	50
pace_actionseq_21_10	21	69	44
pace_actionseq_22_2	22	133	33
pace_actionseq_23_10	23	79	41
pace_actionseq_26_2	26	195	41

Table 5.2: large graphs

Graph	Nodes	Edges	Optimal
exact015	40	360	164
exact020	50	707	101
exact025	60	1053	439
exact030	70	1522	277
exact035	80	1831	385
exact040	90	2182	492
exact045	100	1981	1085
exact050	113	3848	1440
exact055	120	3723	1410
exact060	124	2288	1491

5.1.2. Solver configurations

Based on the modifications discussed in chapter 4, 8 different value selection configurations are compared. In particular, those configurations are:

1. **Default chuffed**: Chuffed without any modifications. This will be used as a baseline, in order to judge the performance of the algorithms.
2. **Chuffed with SBPS**: This version will be using SBPS, as described by E. Demirović et al.[11].
3. **Simple heuristic value selection**: This version uses the simple heuristic for value selection.
4. **Simple heuristic value selection with SBPS**: This version uses the simple heuristic for value selection, in combination with SBPS as described in section 4.2.4.
5. **Kanpai value selection**: This version uses Kanpai for value selection.
6. **Kanpai value selection with SBPS**: This version uses Kanpai for value selection, in combination with SBPS as described in section 4.2.4.
7. **Warm start**: This version uses warm start, as described in section 4.2.3.
8. **Warm start with SBPS**: This version uses warm start, in combination with SBPS, as described by E. Demirović et al.[11].

The above configurations are based on version 0.10.4 of Chuffed[8]. Since all these value selection configurations affect the VSIDS branching, VSIDS will be constantly enabled for all configurations, in order to better observe any differences the value selection heuristics might have. Furthermore, for the 4 value selection configurations using kanpai or the simple heuristic value selection (3-6), the initial activity for the variables that are part of the adjacency matrix is set to 1000. All other solver parameters are not affected between configurations and are set to their default values, in order to ensure that any difference observed between runs is caused by the different configurations.

5.1.3. Measurements

In this experiment, the following variables will be recorded for all tests using the 10 small graphs:

1. **Nodes**: The total number of search space nodes explored by Chuffed during solving. This metric will show any performance difference between configurations, without accounting for any additional overhead caused by different value selection configurations.

2. **Solve time**: The time it takes for Chuffed to fully solve a problem, hence the time it takes to find the optimal solution and prove optimality. This metric will show which configurations are overall the fastest at fully solving a problem.

3. **Optimal time**: The time it takes for Chuffed to find the optimal solution, without proving optimality. This will always be shorter than the solve time.

From these metrics, the expected outcome is to mainly see an improvement in the time taken to find the optimal answer using the configurations with heuristic value selection, as the heuristic algorithm is able to find the optimal answer almost instantly for most of these graphs. However, there is also expected to be a small positive effect on the number of nodes, due to the heuristic approaches having an effect on generating new clauses and as such, adding a stricter upper bound. However, due to the additional overhead of the algorithms, the effect on solve time should be smaller, or there could even be an overall increase in the solve time.

For the 10 large graphs, the best solution found by Chuffed by the end of each test will be recorded. This metric will show how each configuration affects the solution quality after the same amount of time. This scenario should present the main benefit for the heuristic approach, as the heuristic algorithm is able to find near-optimal solutions almost instantly for even larger graphs, while the CP solver is only able to find solutions with a high number of edits as the complexity of the problem increases.

Lastly, for all tests, the graph showing the time vs the number of edits will be generated. An example of such a graph is shown in figure 1.1 and figure 4.1. Before a solver is able to find a solution, it is assumed that the solution is the number of edges. Furthermore, the 0-point of the x-axis is the optimal solution. This allows this metric to better represent how quickly a solver is able to find near-optimal or optimal solutions. A good solver configuration should be able to find near-optimal or optimal solutions early on. In order to compare how each configuration ranks in this aspect, the integral of this graph will be calculated. Due to the behaviour of the heuristic algorithm being able to find near-optimal or optimal solutions significantly faster than the CP solver, we can expect the configurations utilizing the heuristic value selection to have the most significant advantage in this metric.

In order to better compare the results between all the different instances, all the previously mentioned metrics, except for the best objective found, are normalized to the result of default chuffed. Therefore, for example, a result of 100% would indicate the same performance as the default configuration. For mean calculation of the normalized results, the geometric mean is preferred over the arithmetic mean, as it provides correct results when normalizing in relation to other results [25]. The formula of the geometric mean is as follows:

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$$

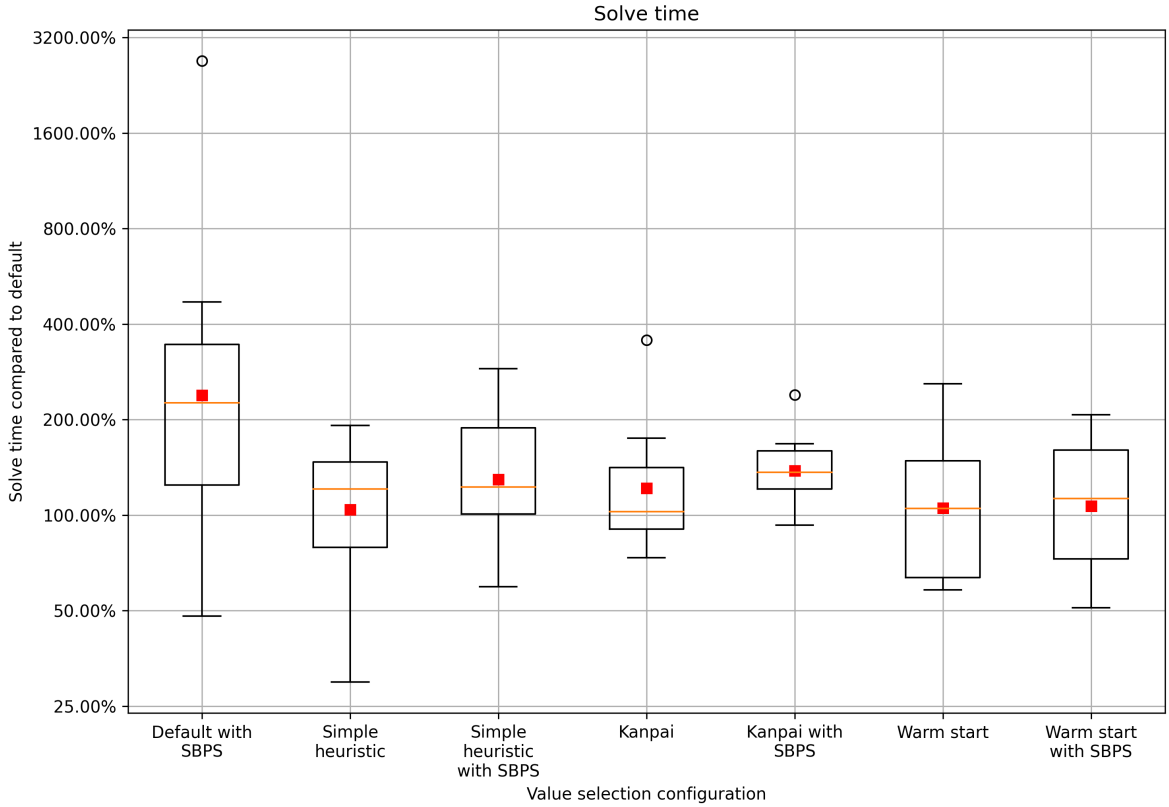
For averaging the objective, instead of normalizing in comparison to the default configuration solution, instead, a lower and upper bound is used for each instance, with the lower bound being 0 and the upper bound being 1. The lower bound is the optimal number of edits, which is calculated using the KaPoCE solver [5], the number one ranking solver from PACE 2021 for both the heuristic and exact track [19]. the upper bound is the number of edges. The objective normalization is also used for the y-axis of the integral before the main normalization, as it sets the 0-point of the y-axis to the optimal solution and the default value when no solution has been found to 1.

Similar to calculating the integral, the lower bound is the optimal solution, and the upper bound is the number of edges. The arithmetic mean is preferred, as the results are not normalized in relation to another solution, but an upper and lower bound, which surrounds the domain of possible answers. The main reason for the different normalization of the objective is that since some of the results could be 0, using the geometric mean should be avoided, as the geometric mean multiplies the results instead of adding them.

Last, the t-score and the p-value of a statistical significance test for each configuration will be recorded. The null hypothesis is that a configuration follows the same distribution as the default configuration. The significance level will be at $p < 5\%$.

5.2. Results

The results presented here are averaged over the 20 different graphs for the integral, over the 10 smaller graphs for the number of total nodes, solve time, and time taken to find the optimal answer, and over the 10 larger graphs for the best value found for the objective. The plots present the boxplot of the results, alongside the geometric or arithmetic mean for the normalized results of each value selection configuration.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	2.385518	0.480567	1.244321	2.255587	3.447963	4.696202	2.516545	0.021553
Simple heuristic	1.040712	0.29784	0.790282	1.209851	1.46837	1.917982	0.225667	0.824003
Simple heuristic with SBPS	1.294866	0.595094	1.00839	1.226919	1.880836	2.891944	1.69725	0.106872
Kanpai	1.213998	0.73275	0.903951	1.025249	1.409997	1.748143	1.327822	0.200831
Kanpai with SBPS	1.378026	0.929949	1.206769	1.363693	1.595798	1.678569	3.696709	0.001651
Warm start	1.050639	0.581071	0.634487	1.049977	1.481506	2.595636	0.300204	0.767461
Warm start with SBPS	1.067259	0.510918	0.72846	1.129032	1.600441	2.071551	0.423892	0.676667

Figure 5.1: Average solve time for each configuration.

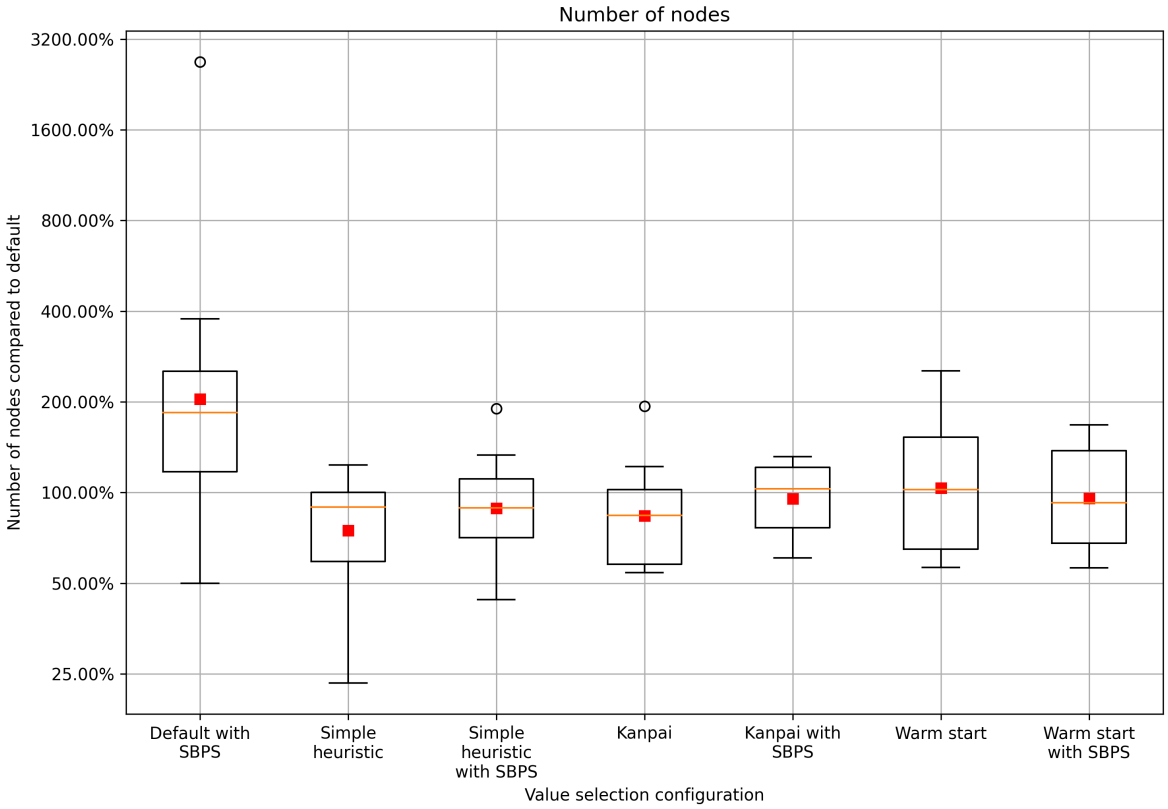
Figure 5.1 presents the average normalized solve time taken for each configuration. That is the time taken to find and prove the optimal solution. A value above 100% indicates an increased solve time over the default configuration, while a value under 100% indicates that the configuration had a faster solve time over the default configuration.

From these results, we observe that SBPS with the current solver configuration does not perform well in these instances, with a mean solve time of 238.55% of the solve time of the default Chuffed results. This could potentially be caused due to SBPS getting stuck in local minima.

For the configurations relying on heuristic value selection, there is a slight increase in solve time. In particular, it is observed that the simple heuristic has the solve time closest to the default configuration,

having a solve time of 104.07% of the solve time of the default configuration. The other configurations, especially the ones that are relying on SBPS are even slower.

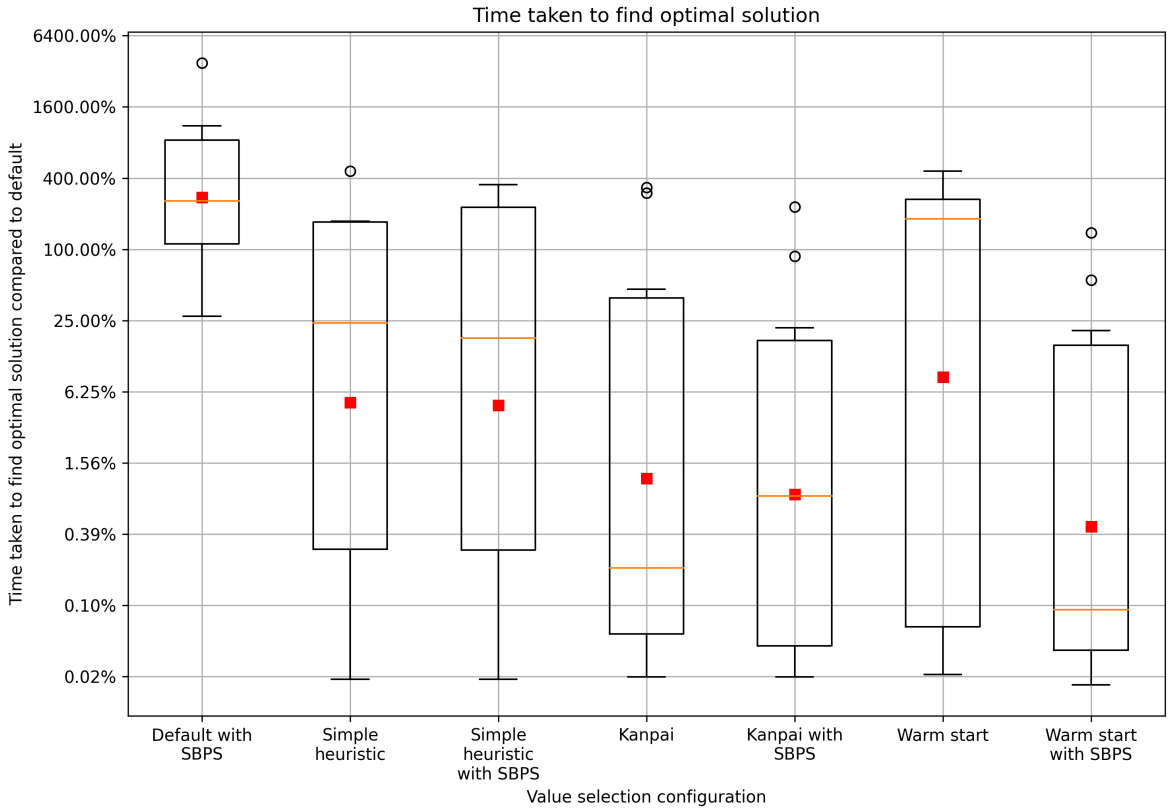
Looking at the p-value, for most configurations, the null hypothesis cannot be rejected, and hence, there is not a significant enough change from the default configuration. However, with a p-value of 0.02 for the default configuration with SBPS, and 0.001 for the Kanpai with SBPS, which are below the significance level of 0.05, the null hypothesis can be rejected. In particular, these 2 configurations have significant evidence to state that their solve time is worse than the default configuration.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	2.045148	0.499285	1.173432	1.840574	2.529081	3.770233	2.092815	0.0508
Simple heuristic	0.747186	0.233207	0.589325	0.893452	0.99955	1.235428	-1.82643	0.084419
Simple heuristic with SBPS	0.885931	0.441158	0.70788	0.889867	1.108339	1.332441	-0.88356	0.388585
Kanpai	0.836554	0.541408	0.578376	0.839778	1.022355	1.220068	-1.35249	0.192968
Kanpai with SBPS	0.95231	0.607011	0.763386	1.02822	1.210889	1.316946	-0.53212	0.601149
Warm start	1.035598	0.563719	0.647511	1.022958	1.528113	2.532	0.215584	0.831736
Warm start with SBPS	0.957103	0.562359	0.677889	0.924078	1.377928	1.67752	-0.3351	0.741426

Figure 5.2: Average nodes for each configuration.

Next, figure 5.2 shows the nodes created during searching in Chuffed. Unlike figure 5.1, this graph appears to show a slight improvement for most of the heuristic value selection configurations. Although the number of nodes is highly correlated to the solve time, as they represent the iterations taken on an instance, the additional overhead of the heuristic value selection configurations contributes to an increased computation time per node, and hence, a higher solve time. Despite the apparent improvements, for all heuristic value selection configurations, there is not enough evidence to reject the null hypothesis.



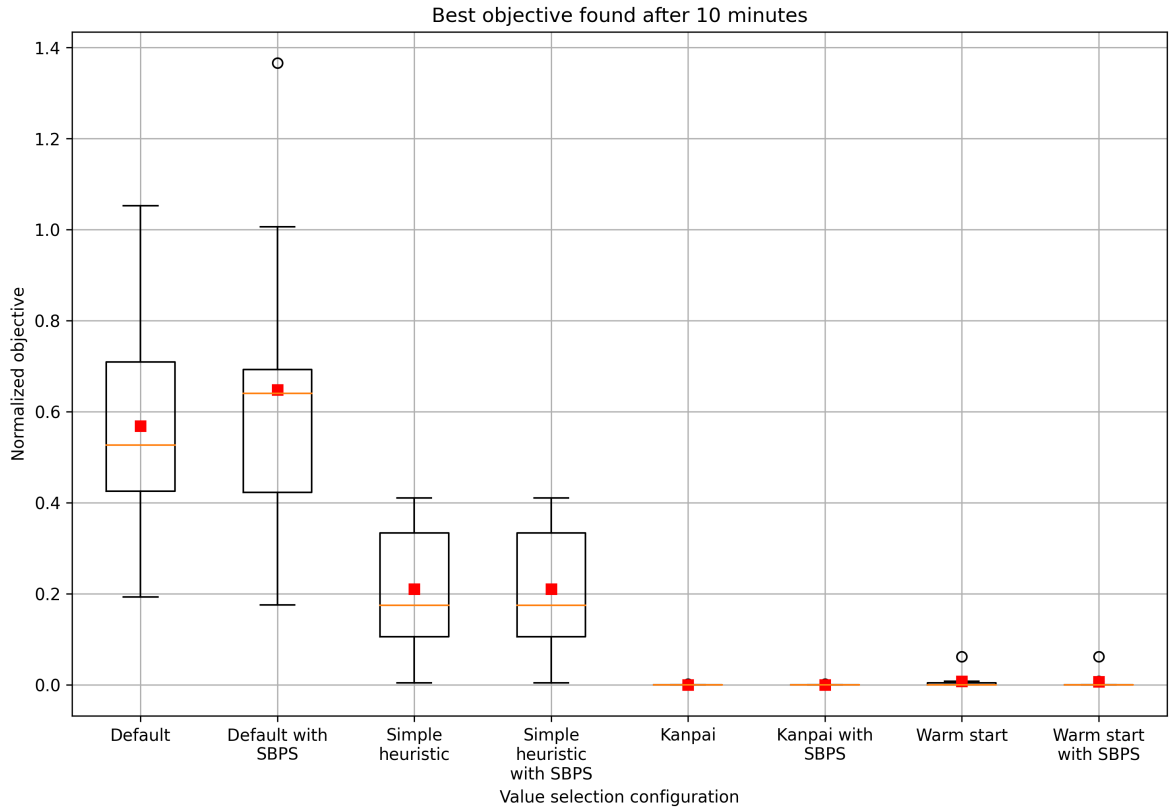
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	2.757653	0.271369	1.111295	2.561506	8.383388	11.05695	2.154184	0.045023
Simple heuristic	0.05062	0.000233	0.002914	0.237547	1.70658	1.727959	-2.46653	0.023912
Simple heuristic with SBPS	0.048017	0.000233	0.002873	0.177154	2.260137	3.51126	-2.51456	0.021642
Kanpai	0.011571	0.000243	0.000562	0.002027	0.387268	0.460839	-3.58082	0.002136
Kanpai with SBPS	0.008498	0.000243	0.000443	0.0082	0.169554	0.217301	-4.26861	0.000462
Warm start	0.082939	0.000254	0.000643	1.812864	2.65082	4.599752	-1.73705	0.09946
Warm start with SBPS	0.004538	0.000209	0.000407	0.000901	0.154549	0.205195	-5.01515	8.98E-05

Figure 5.3: Average optimal time for each configuration.

Continuing, figure 5.3 shows the time taken by the solver to find the optimal solution. Unlike the solve time, this metric does not count the time taken to prove optimality.

Since the heuristic is able to find the optimal answer almost instantly, in comparison to Chuffed, it is expected for the optimal answer to similarly be found extremely fast when using the Kanpai approach for value selection. The results using the Kanpai results have an optimal time with a mean of 1.16%, indicating that the answer compared to the default Chuffed configuration was found almost instantly. Furthermore, the other configurations using a heuristic value selection also perform significantly better, with the simple heuristic having a mean of 5.06%, and the warm start having a mean of 8.29%. Notably combing the heuristic approaches with SBPS appears to have a mostly positive impact on the final results, most apparent with the warm start with SBPS configuration, which achieved the lowest overall mean of 0.45%.

For the optimal time, there is enough evidence to reject the null hypothesis for all configurations with high confidence. Therefore, also considering the t-value it can be stated that all heuristic value selection configurations result in a decrease of the total solve time.

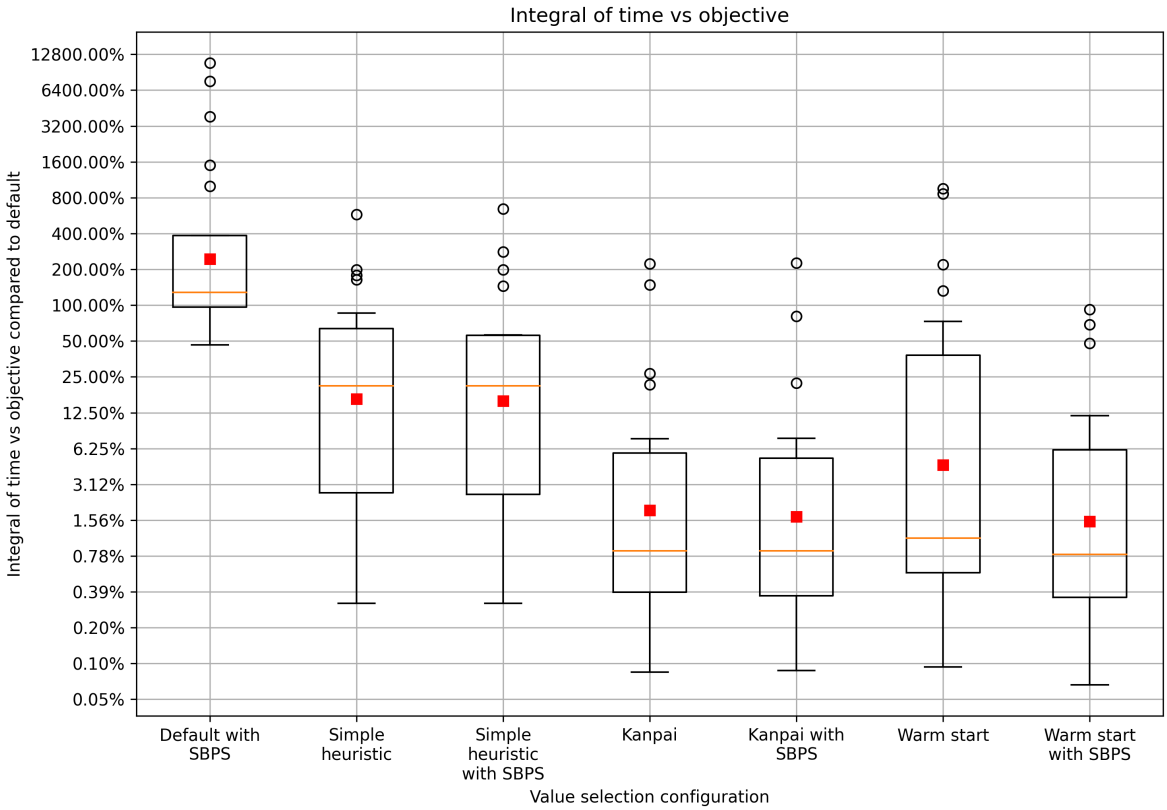


Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.568313	0.193106	0.425003	0.526528	0.709114	1.052455	0	1
Default with SBPS	0.648187	0.175249	0.422239	0.640206	0.692708	1.006274	0.606735	0.551604
Simple heuristic	0.210083	0.004016	0.10578	0.174122	0.333707	0.410289	-4.0064	0.000828
Simple heuristic with SBPS	0.210083	0.004016	0.10578	0.174122	0.333707	0.410289	-4.0064	0.000828
Kanpai	0.000125	0	0	0	0	0	-7.35483	7.94E-07
Kanpai with SBPS	0.000125	0	0	0	0	0	-7.35483	7.94E-07
Warm start	0.007489	0	0	0	0.003827	0.007528	-7.23667	9.91E-07
Warm start with SBPS	0.006979	0	0	0	0	0	-7.24293	9.80E-07

Figure 5.4: Average objective after 10 minutes for each configuration.

Moving to the results from the large graphs, figure 5.4 presents the normalized number of edits found. In this scenario, The approaches utilizing Kanpai are able to find near-optimal or optimal solutions, with all four approaches utilizing the full Kanpai algorithm achieving a median of 0, indicating that in more than half the graphs, they were able to find the optimal solution. In particular, warm start was able to find the optimal solution for 7 out of the 10 instances, while warm start with SBPS and Kanpai were able to find the optimal solution for 8 out of the 10 instances. Last, Kanpai with SBPS was able to find the optimal solution in 9 out of the 10 instances. In comparison, the other configurations did not able to find the optimal solution for any of the large instances. For these 4 configurations, the p-value is lower than 10^{-6} for all of them, indicating that the null hypothesis can be rejected with extremely high confidence.

From these results, the limitations of the simple heuristic, in comparison to the full Kanpai value selection approach become more clear. Nonetheless, the simple heuristic is still able to find significantly better solutions when compared to the default configuration, with a mean of 0.21, in comparison to the mean of the default configuration of 0.57. Taking into account the p-value of 0.0082, there is enough evidence to reject the null hypothesis, and hence prove that even a simple approach to heuristic assistance in value selection can have great benefits.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	2.452597	0.464468	0.95999	1.284535	3.848816	3.848816	2.413331	0.020737
Simple heuristic	0.163475	0.00312	0.026534	0.210655	0.636215	0.858832	-3.75844	0.000574
Simple heuristic with SBPS	0.15704	0.00312	0.025785	0.210675	0.560174	0.56202	-3.7919	0.000521
Kanpai	0.01891	0.000827	0.003871	0.008653	0.057491	0.075826	-8.06616	9.33E-10
Kanpai with SBPS	0.016676	0.000852	0.003625	0.008646	0.051646	0.07634	-8.86974	8.57E-11
Warm start	0.045558	0.000912	0.005654	0.011026	0.380108	0.732328	-4.86532	2.02E-05
Warm start with SBPS	0.015202	0.000648	0.003517	0.008086	0.060682	0.118091	-8.86472	8.70E-11

Figure 5.5: Average integral value for each configuration.

The last metric presented in figure 5.5 shows the normalized to default Chuffed results of the definite integral of the graph plotting the number of edits vs time, for all 20 graphs. These results encapsulate some of the information presented in figure 5.3 and 5.4.

For all configurations, enough evidence exist to reject the null hypothesis. In particular, due to the increased sample size of the integral results, using all 20 graphs, the null hypothesis can be rejected with higher confidence for each configuration, when compared to the results for the optimal time and the best objective found after 10 minutes.

Due to the Kanpai approach being able to find the optimal solution extremely fast even for larger problems, the definite integral, hence the area under the graph has a mean of 1.89% of the area of default Chuffed. Nonetheless, the simple heuristic approaches also have a significant effect on the integral, recording a value of 16.35% and 15.70% for the versions without and with SBPS respectively.

Overall, for the 6 value selection configurations assisted by a heuristic algorithm, not enough evidence exist to suggest that they have an effect on the overall solve time of the solver. However, they do have, with above 95% confidence, a significant positive impact when it comes to finding near-optimal solutions early on, as shown in the previous results.

5.3. Comparing performance differences

In this section, the performance of each configuration will be discussed. In particular, the advantages and drawbacks of each configuration will be presented, and next, how the findings of these results can be generalized to other problems.

5.3.1. Simple heuristic value selection

The simple heuristic value selection, if the main goal is to get good solutions early on, performs in between the default implementation and the approaches utilizing Kanpai. This approach, especially in comparison to the Kanpai approach is more greedy, looking only one move forward. However, it still provides great guidance to the CP solver that assists in finding good solutions faster.

The main benefit of this approach is that it can be more easily generalizable. It shows that at least for the cluster editing problem, using the same simple greedy heuristics used by heuristic algorithms can lead to finding good and optimal solutions faster, and also prove optimality faster.

In particular, This approach to value selection is also similar to the following lookahead process. Try selecting both possible values for a variable and then propagate the rest of the variables for both selected values. Continuing, compare how the objective function changes, and select the value that increases the least the lower bound of the objective function in the case we want to minimize. This is an approach that can be generalized to any optimization problem, and as a result, would not require combining the solver with specialized code.

For example, an example problem has a set of variables V and an objective function f with domain $lb..ub$ that needs to be minimized. After selecting a boolean variable $x \in V$, first, false is prematurely selected for x and the other variables and the objective function are propagated. After which, let's say the domain of f becomes $lb + 2..ub$. Next prematurely selecting true for x, after propagation, the domain of f becomes $lb + 1..ub - 1$. In this scenario, selecting true for x results in a domain for f where the lower bound is lower than if selecting false. Hence, this approach would ultimately select true.

5.3.2. Kanpai value selection

Using the Kanpai value selection allowed the solver to find in most cases the optimal solution significantly faster. One potential flaw of this solver is additional time overhead. Although the number of iterations is limited when using the Kanpai approach, due to having to still fully solve the graph heuristically, instead of only performing one step in the simple heuristic value selection, there is a greater time overhead, as a result, slowing down the performance of the algorithm.

The Kanpai value selection is furthermore not easily generalizable to other problems. For the simple heuristic value selection, a fully generalizable method, without needing modifications to the solver between problems could be created. This approach does not provide any knowledge that can be generalized, other than combining heuristic solvers with Chuffed could improve performance. Furthermore, since most of the performance benefits of this approach come mostly from the early stages of computation, using the heuristic result as warm start with solvers that support such an option allows for a similar performance benefit without needing to modify the solver.

5.3.3. Warm Start

The warm start implementation is able to find a solution on average faster than default chuffed, however, it still appears to be slower in some cases when compared to the other approaches using heuristic value selection. One reason for this is that with a warm start, the optimal solution might not be found at the first step, resulting in the algorithm having to take extra steps to find the optimal solution without any heuristic assistance.

SBPS appears to have a positive effect on warm start. Using SBPS with default Chuffed, Chuffed would often find solutions with a high number of edits first, and SBPS would only be able to find local minima around these high solutions. Warm start instead provides an initial solution closer to the optimal answer, hence, the effect of SBPS is more significant, being able to better guide the solver toward the optimal solution.

This approach is more generalizable than the Kanpai approach. Many solvers allow for a warm start input that is used in a similar way, meaning that no solver modification is required. Other research has also shown this approach to have significant improvements in performance for other problems [14].

5.3.4. SBPS with heuristic value selection

When using SBPS with the non-warm start heuristic value selection, as introduced in section 4.2.4, there are at most only small performance advantages, especially when compared with the performance advantage SBPS offers to warm start. Similar to the default SBPS configuration, it is possible that the solver is getting stuck in local minima. Furthermore, the heuristic solver has to determine how to deal with conflicts between the current state and the best solution found so far, and by default, it ignores the edges from the best solution that cause conflicts, leading to a resulting interpretation of the graph inside the heuristic solver that only lightly resembles the previous best solution in later stages.

5.4. Generalizing findings

Overall, using a heuristic solver for value selection has an insignificant effect on the total solve time. However, this approach benefits more early on, as it allows the CP solver to emulate the performance of the heuristic solver, by finding better solutions earlier.

A way of using lookahead propagations could potentially emulate the performance of the simple heuristic value selection, generalizing this approach. From these results, it is also clear that the heuristic algorithm value selection approach provides its main benefits before finding the optimal solution. Therefore, a way to stop the solver earlier would be beneficial.

6

Implementing a generalizable approach

Based on the results of chapter 5, the main takeaways are that for a simple heuristic approach like Kanpai, simply performing one step of the iteration instead of the entire algorithm is able to provide significant performance improvements. Furthermore, from the warm start results, it is clear that the major benefits of this approach are caused by the early steps of the iterations. However, simply stopping after a solution that satisfies the constraints does not guarantee that a near-optimal solution has been found. Therefore, based on this information, a new approach is developed that should be able to emulate these findings, without the need for the cluster editing algorithm, and therefore, ideally be generalizable to other problems. This chapter discusses this new lookahead approach, starting with the main concept of the lookahead value selection, and continuing with additions to the algorithm that attempt to address certain edge cases and improve performance.

6.1. Lookahead value selection

Based on the simple heuristic value selection, a generalized algorithm can be created that in theory should make similar decisions. The simple heuristic value selection takes as an input the variable of one of the edges of the adjacency matrix g, and the current state of the matrix, and calculates how the number of edits changes based on if the boolean variable is true or false, while ensuring that all constraints are satisfied, before returning an answer. In Chuffed, propagation updates the domain of all variables, including the optimization function, through simplifying clauses after a value has been assigned. This means that after each propagation, the domain for the possible values the optimization function can now take is updated, becoming more strict. Therefore, the simple heuristic should be similar to essentially propagating both possible values for a variable, before returning a final value that has a better domain for the optimization function. This is the process that the lookahead approach attempts to do.

The core process of the lookahead approach is shown in figure 6.1. After VSIDS selects an SAT variable, first, a lookahead propagation is performed for one possible value for the variable, and the domain of the optimization function after the lookahead propagation is recorded, if there are no conflicts. Next, similarly, the lookahead propagation for the other possible value is performed, and again, assuming no conflicts, the domain of the optimization function is recorded. After both lookahead propagations are performed, the recorded optimization function domains for the two possible values are compared. Finally, the value that has the domain that has a more optimal value is ultimately selected. For example in cluster editing, since it is a minimization problem, the value selected is the value that has the lower lowest bound for the domain of the optimization function.

In the case of a tie, the value that results in the smaller domain for the optimal value is selected. This is based on the fail first strategy [17]. If one of the lookaheads ends in a conflict, the lookahead performs an SAT analysis of the conflict and selects the other value. This allows the solver to quickly learn additional clauses and nogoods. If a tie has not been resolved, the value is selected based on the default behaviour of Chuffed.

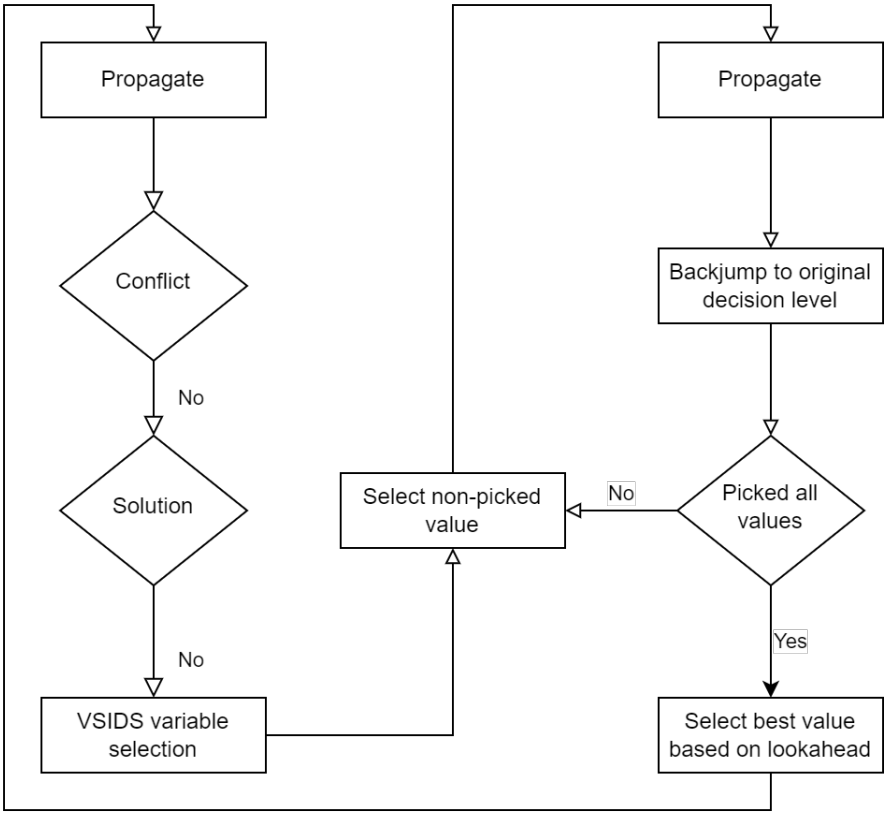


Figure 6.1: An example of the main components of the lookahead constraint programming solver.

The lookahead propagation performs the following steps. It takes as input the selected variable and the candidate value, and first, create a temporary new decision level. Then, it updates the value of the selected variable and enqueues the updated clauses for propagation. After fully propagating, the updated optimization function domain is recorded, and if there has been a conflict, a sat analysis is performed. After, a backjump is performed to the previous decision level, essentially undoing all changes. Last, the recorded domain, and if there has been a conflict is returned. Algorithm 1 shows the pseudocode of this function.

Algorithm 1 Lookahead Propagation

```
function LOOKAHEADPROPAGATE(variable, value)
    NEWDECISIONLEVEL
    assign[variable] ← value
    SAT.ENQUEUE(variable, value)
    PROPAGATE
    domain ← optVal.domain
    conflict ← sat.conflict
    if conflict then
        SAT.ANALYZE
    end if
    BACKJUMP(previousDecisionLevel)
    return domain, conflict
end function
```

6.2. Lookahead approach additions

Now that the main core of the lookahead approach has been designed, further additions are made to the algorithm, in order to improve performance in specific scenarios. These improvements include a slightly different variable selection when many variables have the same activity score, preferring full solutions, even if the domain could be further improved, a way to decrease unnecessary lookahead propagations, and fully stopping lookaheads after a certain amount of conflicts.

First, from early testing, it was noted that the algorithm would often repeatedly pick SAT variables generated from the same integer variable, and iterate toward one direction of its domain. One notable example of the cluster editing problem was the algorithm selecting SAT variables generated for the number of edits, and simply selecting values that would slowly decrease the upper bound of the domain of edits. This led to a significant slowdown. Therefore, in order to avoid such a scenario, when multiple variables have the same activity score, boolean variables are prioritized. As the solver starts assigning different activity scores to each variable the effect of this issue diminishes.

Continuing, in order to produce solutions faster, when one of the two candidate values produces a fixed value for the optimization function, that value is preferred. In case both values lead to a solution, the more optimal solution is preferred. This allows the solver to produce solutions more often and earlier.

The next improvement reduces the number of lookahead propagations in case of conflicts. Based on the current implementation, if one value results in a conflict then there is no need to perform lookahead for both values. Therefore, by always starting the lookahead propagation with the non-default value, if there is a conflict after the propagation, then the default value can be selected without the need for second a lookahead propagation. Based on these additions, algorithm 2 shows the full branch function for the lookahead approach.

Algorithm 2 Lookahead Branch

function LOOKAHEADBRANCH
  $candidates \leftarrow$ variables tied for the highest activity score
  $variable \leftarrow$ an unassigned boolean variable from candidates if possible
  $value \leftarrow$ default value for $variable$
  $domain_0, conflict_0 \leftarrow$ LOOKAHEADPROPAGATE($variable, !value$)
  if $conflict_0$ **then**
    return $variable \leftarrow value$
  end if
  $domain_1, conflict_1 \leftarrow$ LOOKAHEADPROPAGATE($variable, value$)
  if $domain_0$ has a more optimal value than $domain_1$ *or* ($domain_0$ most optimal solution is the same as $domain_1$ *and* $|domain_0| < |domain_1|$) **then**
    return $variable \leftarrow !value$
  end if
  return $variable \leftarrow value$
end function

Last, after finding the optimal, or near-optimal solution, the utility of the lookahead approach diminishes, while the additional overhead of the lookahead propagations makes the algorithm significantly slower. Based on testing on the 20 cluster editing graphs, approximately 40% of the total solve time is spent on lookahead propagation, where a significant number of that time is after the solver has found the best solution. Based on the previous tests it was shown that the warm-start approach performs similarly to the heuristic value selection approaches. However, stopping only after finding the first solution would not be ideal, as there is a high chance the solution is still far from optimal, especially with a more greedy and generalizable approach. Therefore, it was decided to use the number of conflicts caused after finding at least a solution as a stopping criterion for the lookahead approach. This is because after finding a near-optimal or optimal solution, the chance for a conflict to occur after a decision increases, due to the stricter domain bounds.

The code for Chuffed with the lookahead approach can be found in the following repository: `https://github.com/AZoumis/chuffed`

7

Lookahead Results

In this chapter, first, the experimental setup for the lookahead configuration is explained, followed by the results of the lookahead approach on cluster editing, being presented and compared to the results from chapter 5. Continuing, the generalized results of the MiniZinc challenge [24] will be presented, and next, the results for individual problems of the MiniZinc challenge will be discussed. Lastly, an analysis of the performance of the lookahead approach will be made, where the main advantages and disadvantages of the algorithm will be discussed.

A table of the full results for cluster editing can be found in appendix A, and the results for the MiniZinc challenge can be found in appendix B.

7.1. Experimental Setup

To test the performance of the lookahead approach on cluster editing, the experimental setup from section 5.1 will be used, with the addition of lookahead and lookahead with SBPS. The lookahead stopping criterion is set at 100,000 conflicts. In order to better understand the behaviour of the lookahead approach, the value selection of default chuffed, the lookahead approach, the simple heuristic, and Kanpai value selection at the first decision level for each of the adjacency matrix variables will be compared.

To test the generalized performance of the lookahead approach, the MiniZinc challenge 2022 [24] optimization models and instances will be used with the same experimental setup as the cluster editing experiments. In total, there are 19 optimization problems, with 95 instances. In the case of the MiniZinc challenge instances, the solver configurations utilizing the cluster editing solver will not be used, leaving four different configurations for this setup, in particular:

1. **Default Chuffed**
2. **Chuffed with SBPS**
3. **Lookahead**
4. **Lookahead with SBPS**

Furthermore, for all problems in the MiniZinc challenge, a time limit of 10 minutes is added. The same measurements as the cluster editing experiment will be recorded, with the exception of the optimal value and the number of nodes, since there is no guarantee that the optimal value will be found and that the solver will terminate. When a configuration does not terminate within the time limit, the solve time is set to 10 minutes. For the integral and objective, since the minimum and maximum bounds are unknown, instead, the best and worst values found for that instance in all runs will be used instead. In particular, for the worst values, all intermediate full solutions will be considered, not just the final ones. For consistency between minimization and maximization problems, all measurements are converted so lower values indicate a better solution. In case a solution has not been found at a point in time, or by the end of the 10 minutes, the objective is assumed to be twice the range between the lower and upper bound. Therefore, for the objective, if a configuration did not find a single solution satisfying all

constraints, the normalized objective will be 2. Last, an additional measurement of how many instances each configuration is able to fully solve or find a solution for will be recorded.

7.2. Cluster editing results

The first value selection comparison compares the value selection for all the variables of the adjacency matrix g, selected by the default, lookahead, simple heuristic, and Kanpai value selection. More specifically, these results show how similar the value selection would be after the first propagation between the four different configurations.

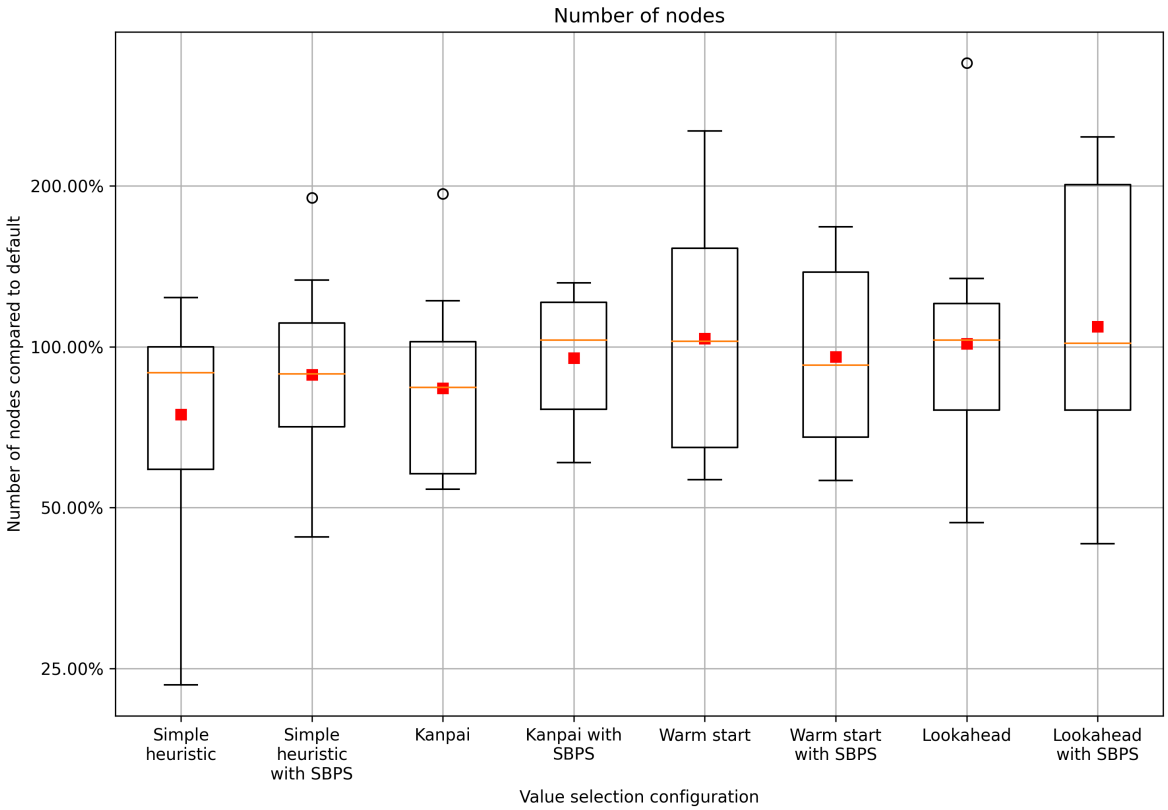
	Default	**Lookahead**	**Simple heuristic**	**Kanpai**
Default	100%	46.683365 %	44.173373 %	56.795098 %
Lookahead	46.683365 %	100%	97.490009 %	80.052239 %
Simple heuristic	44.173373 %	97.490009 %	100%	79.087293 %
Kanpai	56.795098 %	80.052239 %	79.087293 %	100%

Table 7.1: Initial value selection comparison between configurations.

Table 7.1 shows the results from the initial value selection comparison between configurations. These results first show that the value selection initially for default Chuffed is around 50% similar to all other value selection techniques. This is because the default value selection does not have any understanding of the structure of the problem. Hence, since there are only two possible values a boolean variable can take, a random value selection will be 50% similar to another value selection, assuming both values are picked roughly with the same rate.

Continuing, the lookahead approach has an extremely high similarity with the simple heuristic, at 97.49%. This measure, therefore, aligns with the claim that the lookahead approach emulates the simple heuristic. Continuing both the simple heuristic and the lookahead approach have around an 80% similarity with the Kanpai approach. The more greedy approach implemented by the lookahead and simple heuristic lead to these two approaches not being as accurate as the Kanpai approach, however, they are still able to better emulate the value selection of the full Kanpai approach in comparison to default Chuffed.

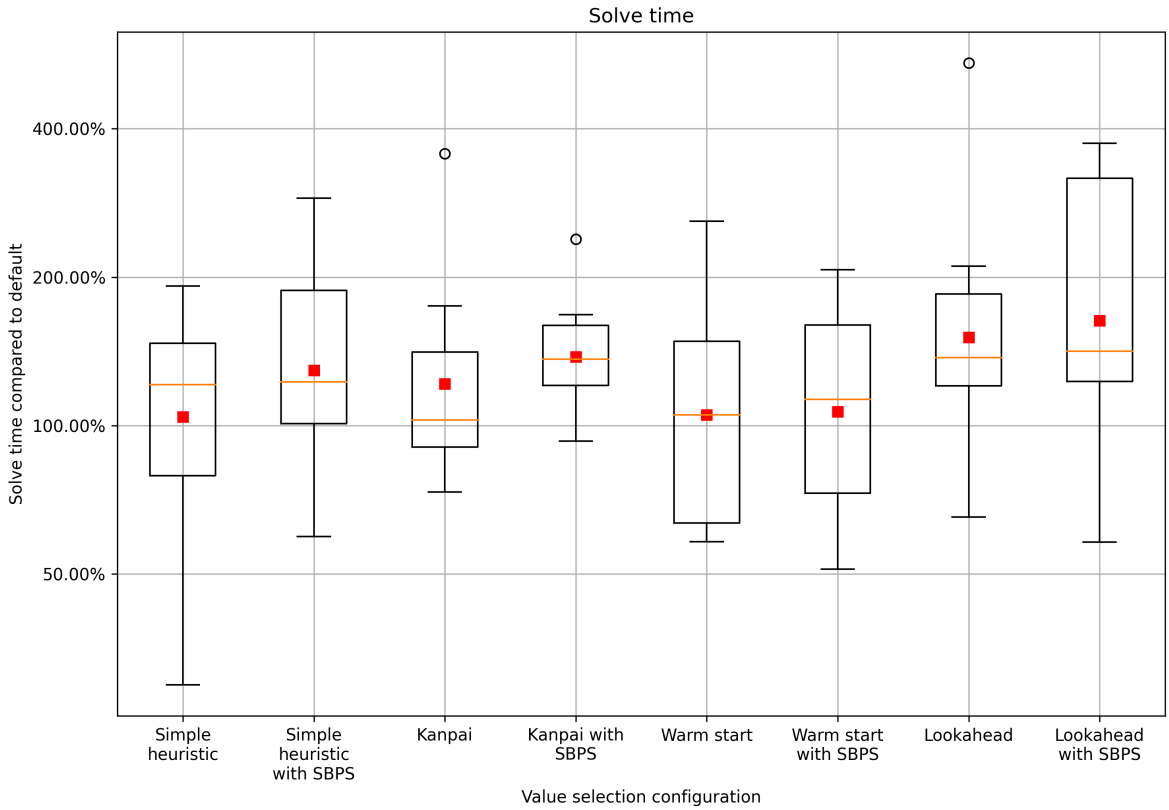
Based on these results, the attempt to emulate the value selection of the simple heuristic appears to be successful, with the lookahead approach having a largely similar behaviour to the simple heuristic. Nonetheless, these results are not fully descriptive of the similarity between the two approaches. In particular, they only show the value selection early on, and they ignore the fact that the simple heuristic approach performs value selection for only variables within the adjacency matrix g, while the generalized lookahead approach performs lookahead for all variables.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Lookahead	1.012878	0.469162	0.760995	1.028948	1.205426	1.341363	0.074967	0.941068
Lookahead with SBPS	1.090036	0.428444	0.760685	1.015147	2.009872	2.468145	0.423271	0.677112

Figure 7.1: Average nodes for each configuration.

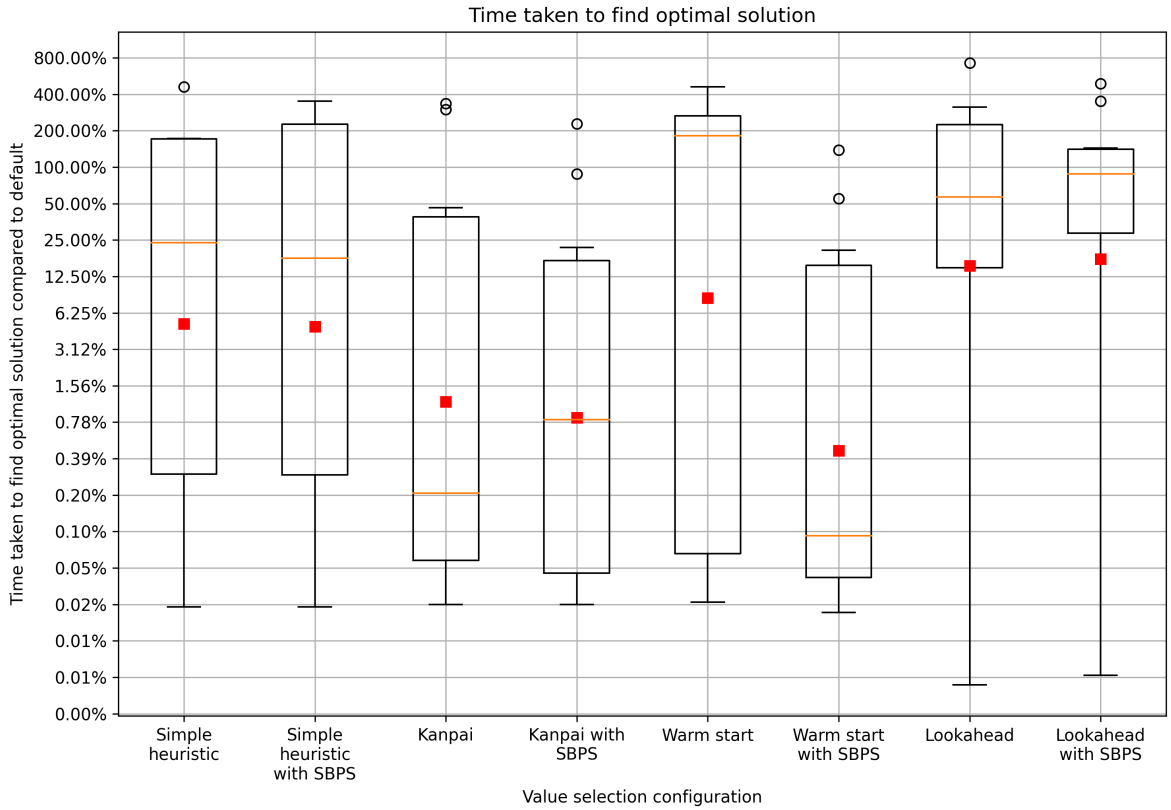
Figure 7.1 present the average nodes used, in comparison to default Chuffed, to fully solve an instance. These results show a minor increase in the number of nodes used, however, with p-values of 0.94 and 0.68, the null hypothesis cannot be rejected. Hence, not enough evidence exists to state that the lookahead approach has a significant effect on the total number of nodes created for cluster editing.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Lookahead	1.510803	0.651806	1.20425	1.37355	1.849818	2.106442	2.289958	0.034318
Lookahead with SBPS	1.63243	0.57996	1.229219	1.414869	3.174621	3.739832	2.383762	0.028355

Figure 7.2: Average solve time for each configuration.

Figure 7.2 presents the average solve time taken by each configuration. The mean solve time taken is 151.08% of the default configuration for the lookahead configuration, and 163.24% for the lookahead with SBPS configuration. With a p-value of 0.034 and 0.028 for the two lookahead configurations, there are enough evidence to reject the null hypothesis and state that the lookahead configuration overall increases the total solve time. Despite not observing a significant difference in the number of nodes between configurations, the additional time overhead caused by the lookahead approach results in significantly higher solve time.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Lookahead	0.152669	5.29E-05	0.148045	0.567645	2.238569	3.122573	-1.46698	0.159636
Lookahead with SBPS	0.174539	6.35E-05	0.285707	0.88106	1.41059	1.43608	-1.3739	0.186343

Figure 7.3: Average optimal time after 10 minutes for each configuration.

Figure 7.3 presents the average time taken to find the optimal time for all configurations including the lookahead approach. These results show that the lookahead approach appears to perform worse than all the approaches utilizing the heuristic algorithm for value selection. In particular, the lookahead configuration has a mean of 15.27%, while the lookahead with SBPS configuration has a mean of 17.45%. Although this is a big decrease when compared to the default configuration, the interquartile range (IQR) ranges between 14.80% and 223.86% for the lookahead configuration, and between 28.57% and 141.06% for the lookahead with SBPS configuration. This shows that the mean is likely skewed by certain well-performing instances, and in actuality, most instances have a worse score than the mean.

Furthermore, with p-values of 0.16 and 0.19, the null hypothesis cannot be rejected with high confidence. It is likely that the more generalized view of the problem and the time overhead are likely the cause for this worsening in performance when compared to the results of chapter 5.

Looking at the performance of the individual instances, shown in figure 5.3, it is observed that certain cases perform significantly worse than the default approach. pace_actionseq_22_2 appears to have the worse optimal time performance.

Figure 7.5 plots the time vs the number of edits for the default and lookahead approach. The major cause for the performance disparity cannot be explained just by the additional time overhead. Instead, it is observed that the lookahead approach reached a specific solution not reached by the default approach, but then was unable to find a better solution before the default configuration.

In particular, a solution with 43 edits was found in 3 milliseconds for the lookahead approach. Despite the default configuration finding a much worse solution with 106 edits first, it reached the optimal solution in 19 seconds, while the lookahead approach took 137 seconds. It is therefore likely that for the lookahead configuration, the solution found and the clauses generated led the solver to repeatedly reach different local minima that did not improve on the final result.

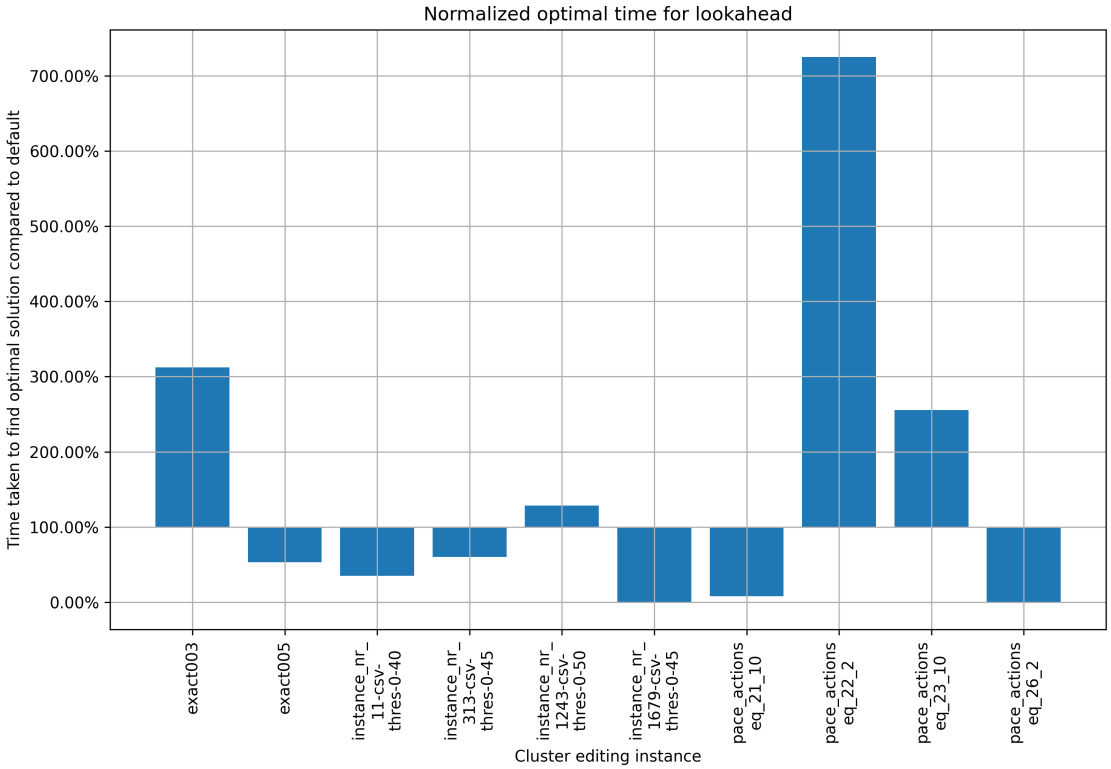


Figure 7.4: Average optimal time for each instance.

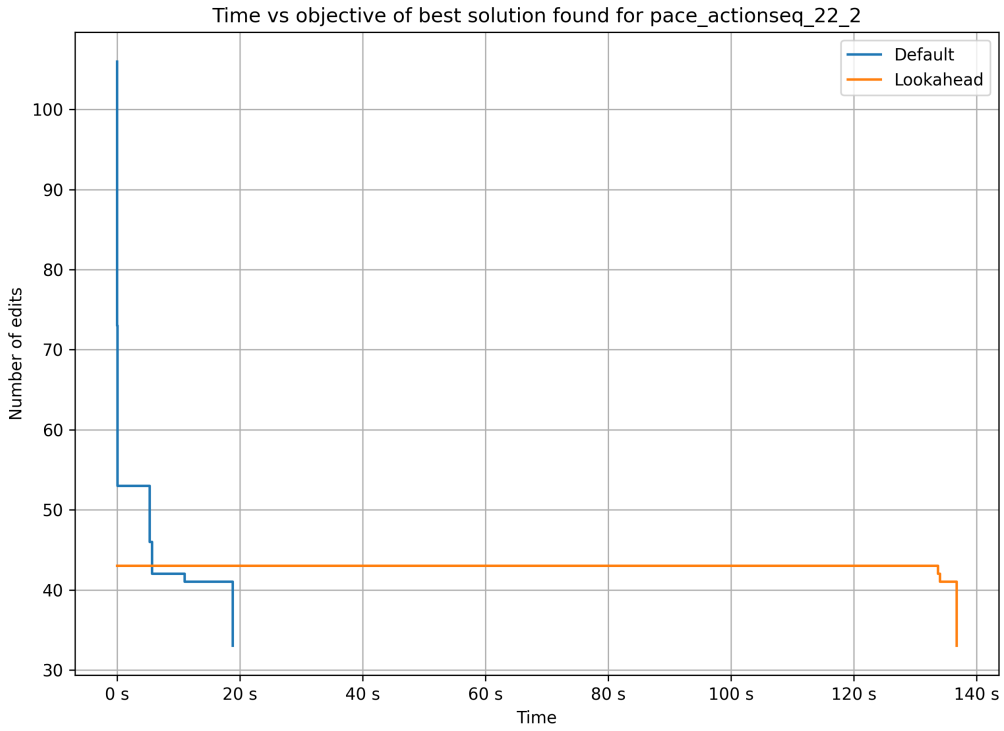
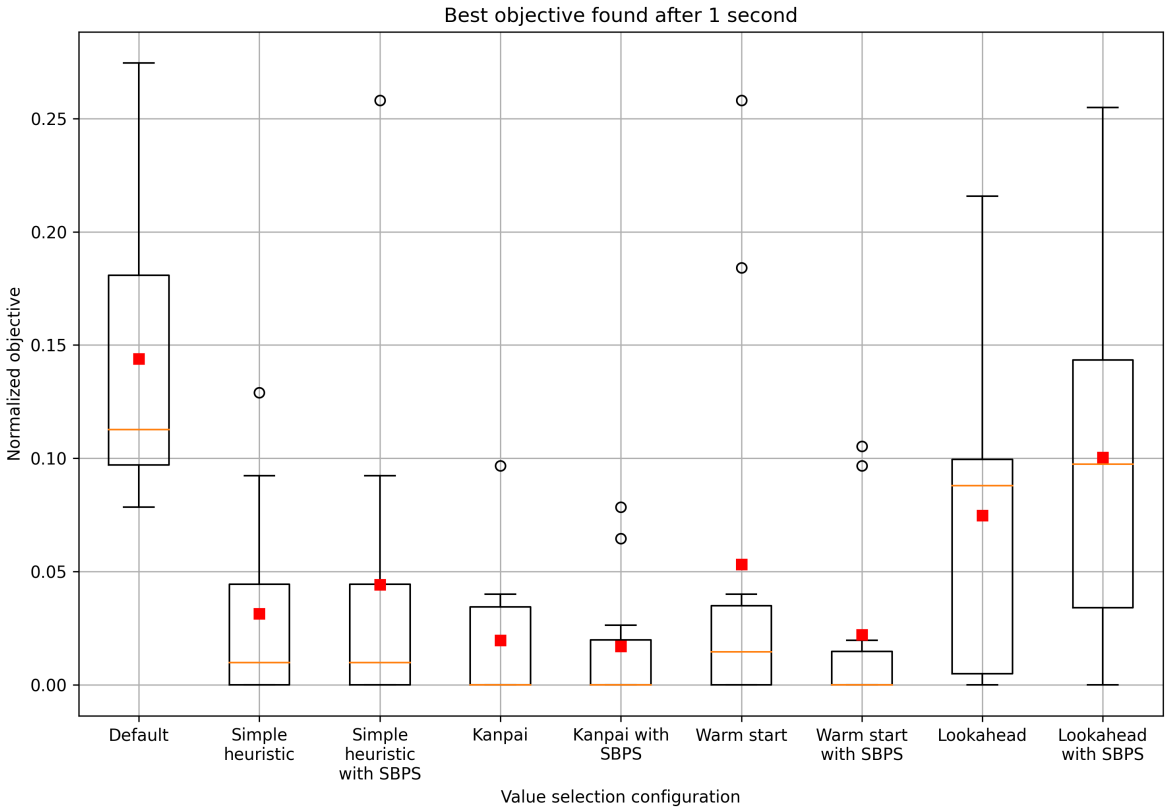


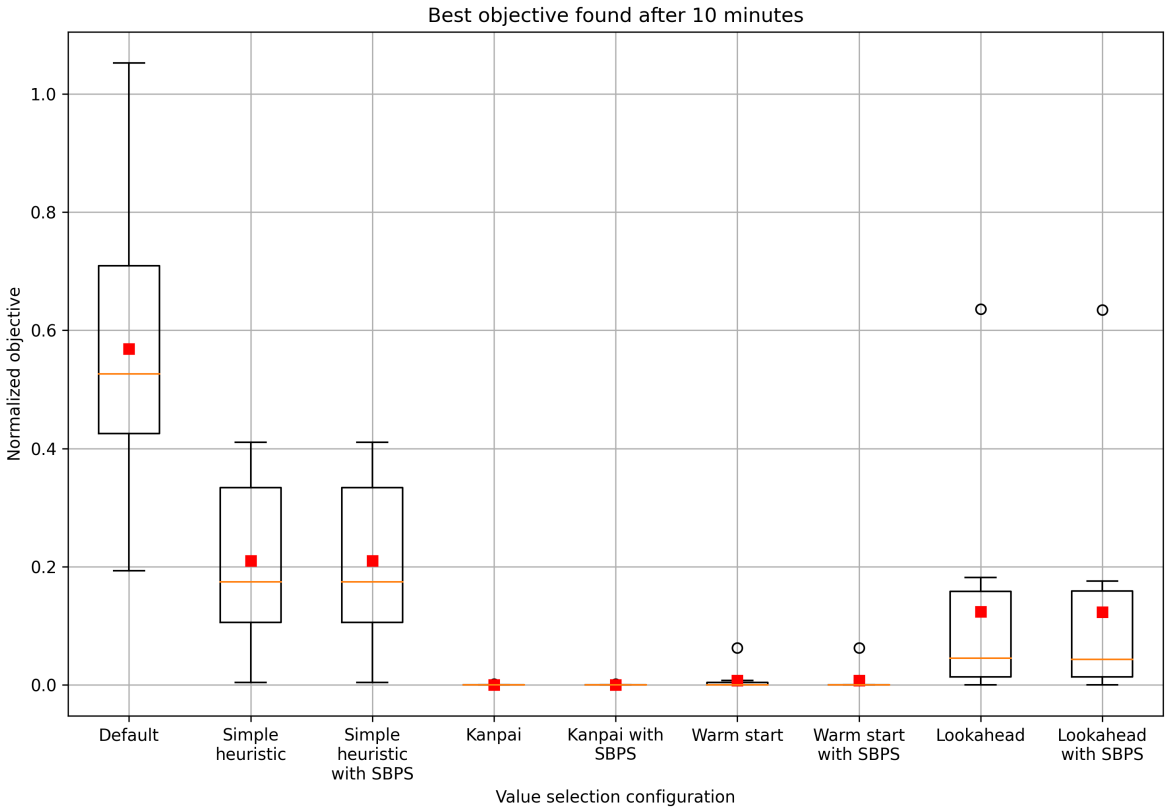
Figure 7.5: Results for time vs edits for pace_actionseq_22_2.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.1439	0.078431	0.09709	0.112632	0.180769	0.27451	0	1
Simple heuristic	0.031319	0	0	0.009804	0.044376	0.092308	-4.25272	0.000479
Simple heuristic with SBPS	0.044222	0	0	0.009804	0.044376	0.092308	-2.9426	0.008706
Kanpai	0.01956	0	0	0	0.034314	0.04	-5.11256	7.28E-05
Kanpai with SBPS	0.016926	0	0	0	0.019737	0.026316	-5.26657	5.23E-05
Warm start	0.053111	0	0	0.014612	0.034902	0.04	-2.49725	0.022436
Warm start with SBPS	0.022165	0	0	0	0.014706	0.019608	-4.71128	0.000174
Lookahead	0.074752	0	0.004902	0.087861	0.09951	0.215686	-2.19178	0.041787
Lookahead with SBPS	0.100414	0	0.033937	0.097407	0.143421	0.254902	-1.25228	0.226494

Figure 7.6: Best normalized objective found after 1 second for the small graphs.

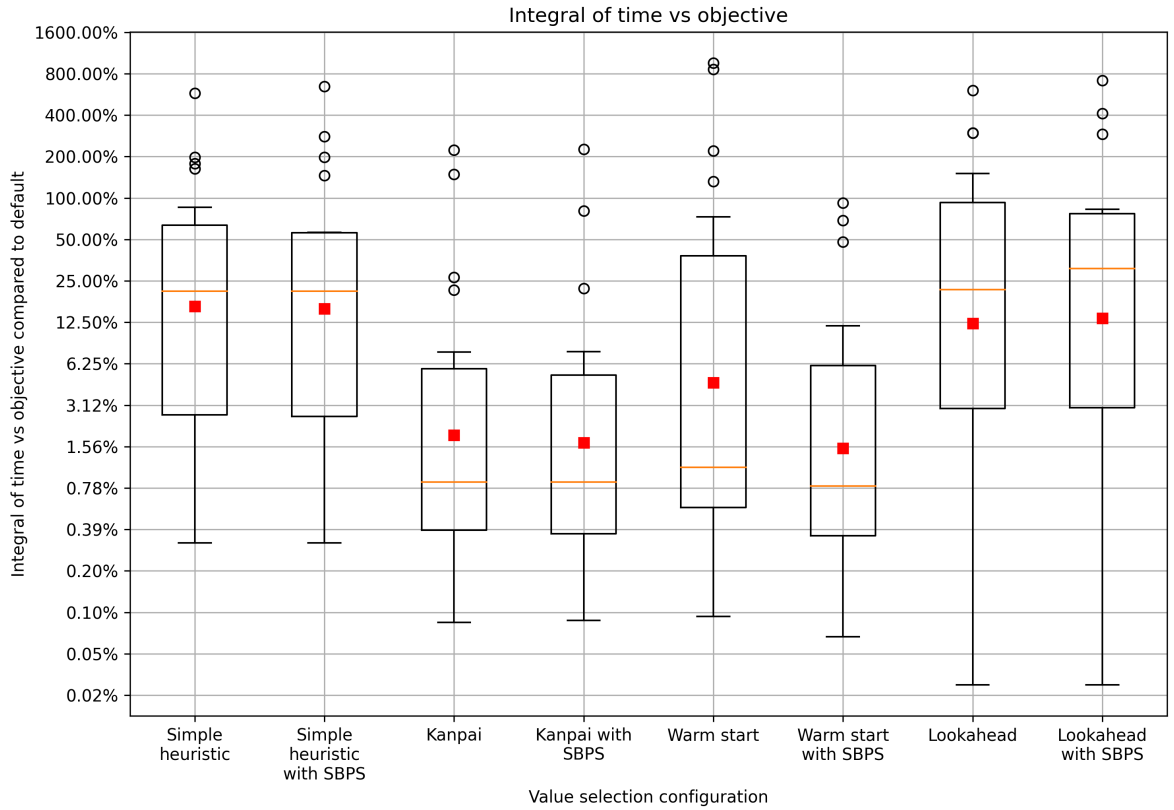
For the instance pace_actionseq_22_2, even if the overall time taken to find the optimal solution was slower than the default, early on, the lookahead approach had found a better solution than the default Chuffed configuration. In particular, figure 7.6 shows that in general, the lookahead, after 1 second more often had better solutions, when compared to the default Chuffed value selection, further indicating that the lookahead approach tends to lead the solver towards local minima. While the null hypothesis cannot be rejected for the lookahead configuration with SBPS, due to a high p-value of 0.23, the null hypothesis can be rejected for the lookahead configuration, due to having a p-value lower than 0.05.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Lookahead	0.12401	0	0.013033	0.044642	0.157702	0.18192	-4.51669	0.000267
Lookahead with SBPS	0.122981	0	0.013033	0.042659	0.158924	0.175223	-4.53002	0.000259

Figure 7.7: Average objective after 10 minutes for each configuration.

Continuing, figure 7.7 shows the normalized number of edits found for the 10 larger graphs. With a normalized mean of 0.12, the lookahead approach performs better than the simple heuristic, which has a normalized mean of 0.21. Furthermore, the median for the lookahead approach is 0.04, an even bigger improvement over the simple heuristic, with a median of 0.17. With the measured p-values for both lookahead configurations being under 0.0003, the null hypothesis can be rejected with high confidence.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Lookahead	0.122533	0.000291	0.029592	0.215486	0.925834	1.505768	-3.35508	0.00181
Lookahead with SBPS	0.13356	0.00029	0.029861	0.307724	0.768231	0.829741	-3.2323	0.002539

Figure 7.8: Average integral for each configuration.

Figure 7.8 shows the average integral in comparison to the default chuffed for all 20 graphs. Since these results essentially group the results of figure 7.3 and 7.7, the performance of the lookahead configuration also ranks between the two metrics. In particular, the performance displayed in 7.8 appears to not favor the lookahead approach as much as figure 7.7, however, when compared to 7.3, the performance of lookahead appears much more positive overall. With p-values of 0.002 and 0.003, enough evidence exist to reject the null hypothesis.

Based on the findings for the cluster editing problem, the lookahead configuration has a significant increase in solve time when compared to the default configuration. However, there is a significant improvement in the solutions found early on in the solve time, as shown in figures 7.6 and 7.8, and in particular for larger problems, as shown in figure 7.7.

7.3. Generalized results

This section will present the generalized results for the lookahead approach, using the MiniZinc challenge 2022 [24]. For more generalized problems, similar to the cluster editing problem, most of the benefits of this approach will likely be in the form of finding near-optimal solutions early on. Furthermore, it is expected for this approach to perform best on larger instances, and for problems where there is a clear correlation between the objective function value and the variables selected. On the other hand, smaller instances with many local minima solutions will likely cause the lookahead approach to perform worse overall.

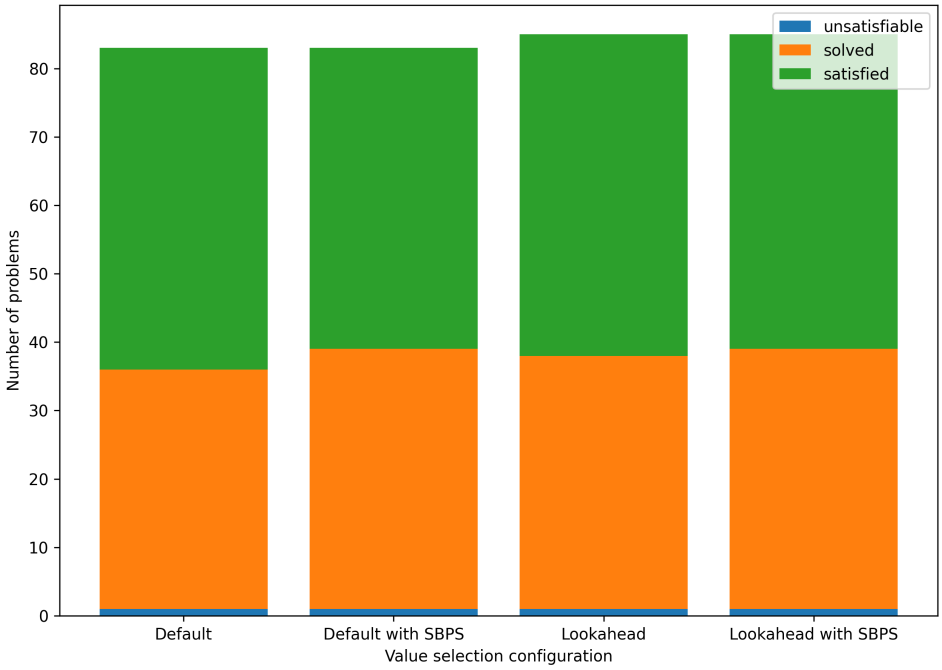
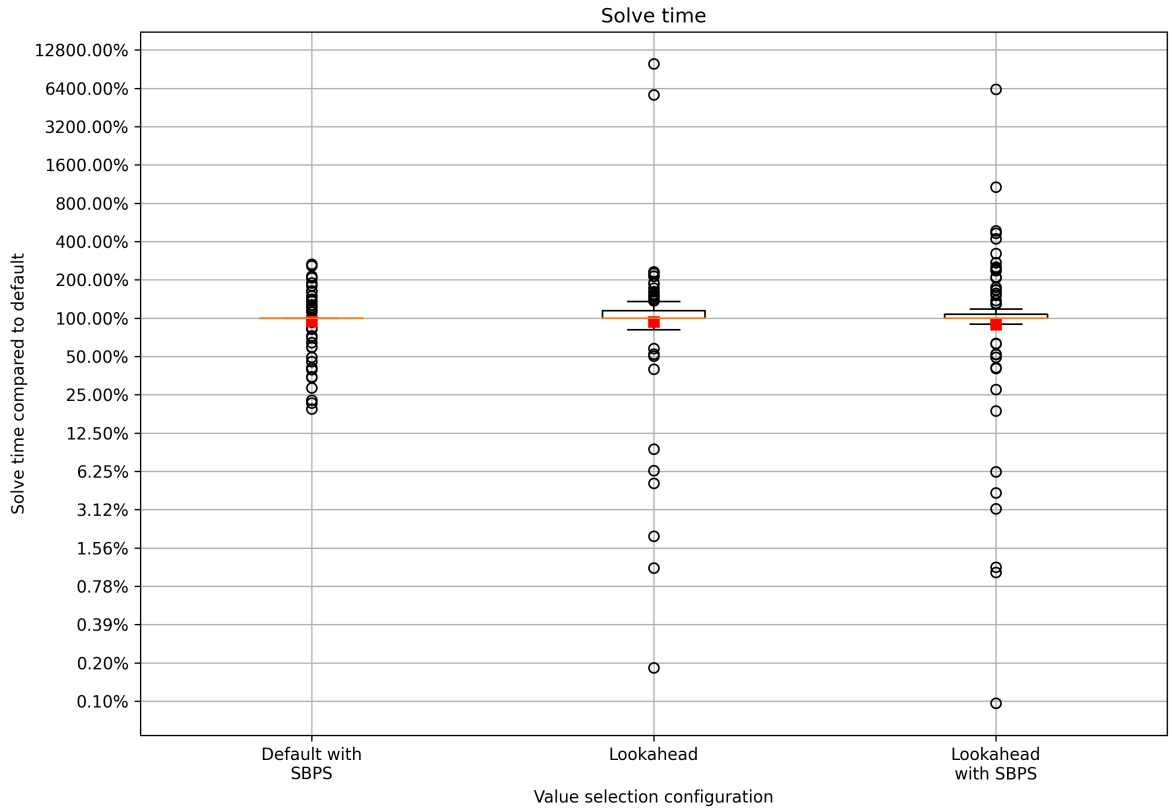


Figure 7.9: Total number of instances where a solution was found (orange), or optimality was proved (blue)

The MiniZinc challenge has a total of 95 optimization problem instances. Within 10 minutes, the default Chuffed configuration is able to find a solution satisfying all constraints for 82 out of the 95 problems, of which it proved optimality for 35. Continuing, SBPS found a satisfying solution for 82 instances and proved optimality for 38. Next, lookahead found a satisfying solution for 84 instances and proved optimality for 37. Last, lookahead with SBPS found a satisfying solution for 84 instances and proved optimality for 38.

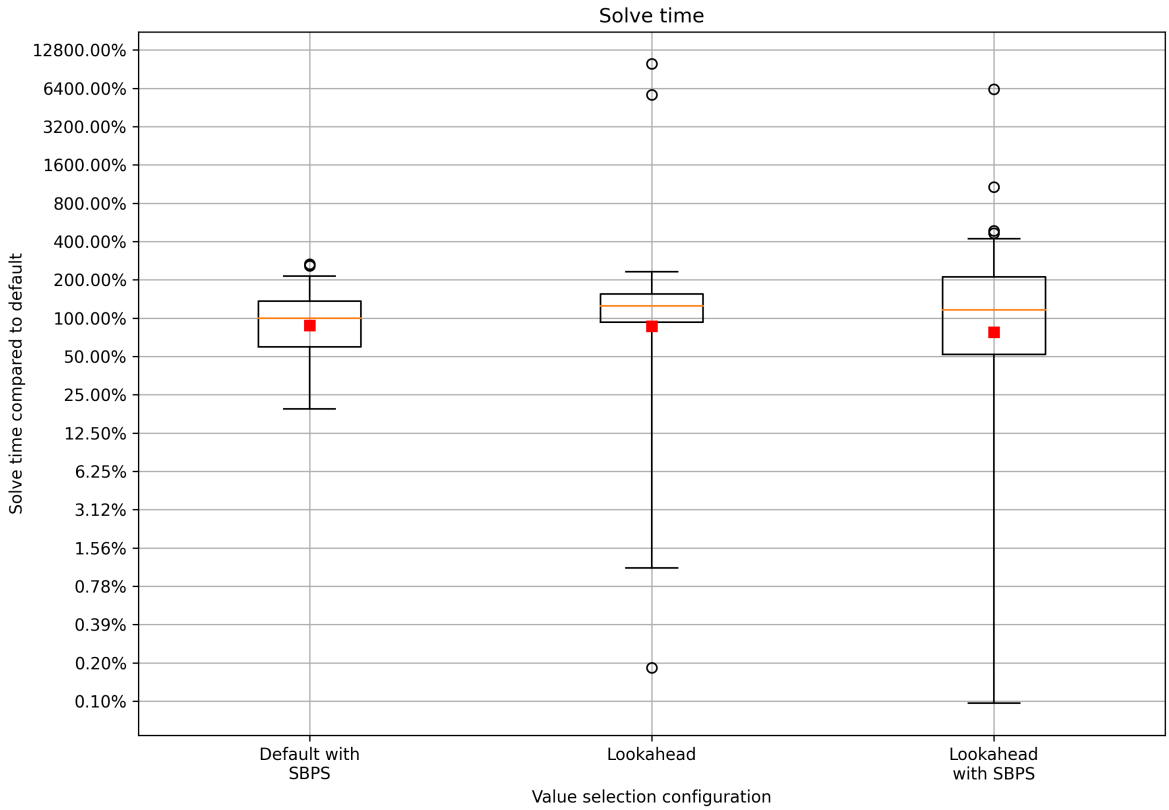
Based on these results, the look-ahead approach has only a small effect on if a solution will be found. Nonetheless, both lookahead approaches were able to find optimality for overall 2 more instances than the default Chuffed configuration. However, SBPS was still able to fully solve 1 more instance when compared to the lookahead approach without SBPS. Nontheless, the lookahead approach was able to find a satisfying solution for 2 more problems over the default and SBPS configurations.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.941057	1	1	1	1	1	-1.2959	0.196599
Lookahead	0.932888	0.810253	1	1	1.14456	1.348515	-0.54847	0.584023
Lookahead with SBPS	0.88766	0.895206	1	1	1.071042	1.17427	-0.87913	0.380456

Figure 7.10: Average solve time for each configuration

Figure 7.10 shows the average time taken to fully solve an instance when compared to default chuffed. In the case where the solver did not terminate within 10 minutes, the solve time is 600 seconds. The default lookahead has a solve time of 94.10%, and the lookahead approach with SBPS has a solve time of 88.76%. Unlike cluster editing, the mean solve time is lower than the default configuration solve time, however, looking at the IQR, the lower quartile and the medial are at 100% for the lookahead configurations, indicating that for 75% of the problems, the lookahead configurations had the same or slower solve time as the default configuration. With high p-values of 0.58 and 0.38, the null hypothesis cannot be rejected.



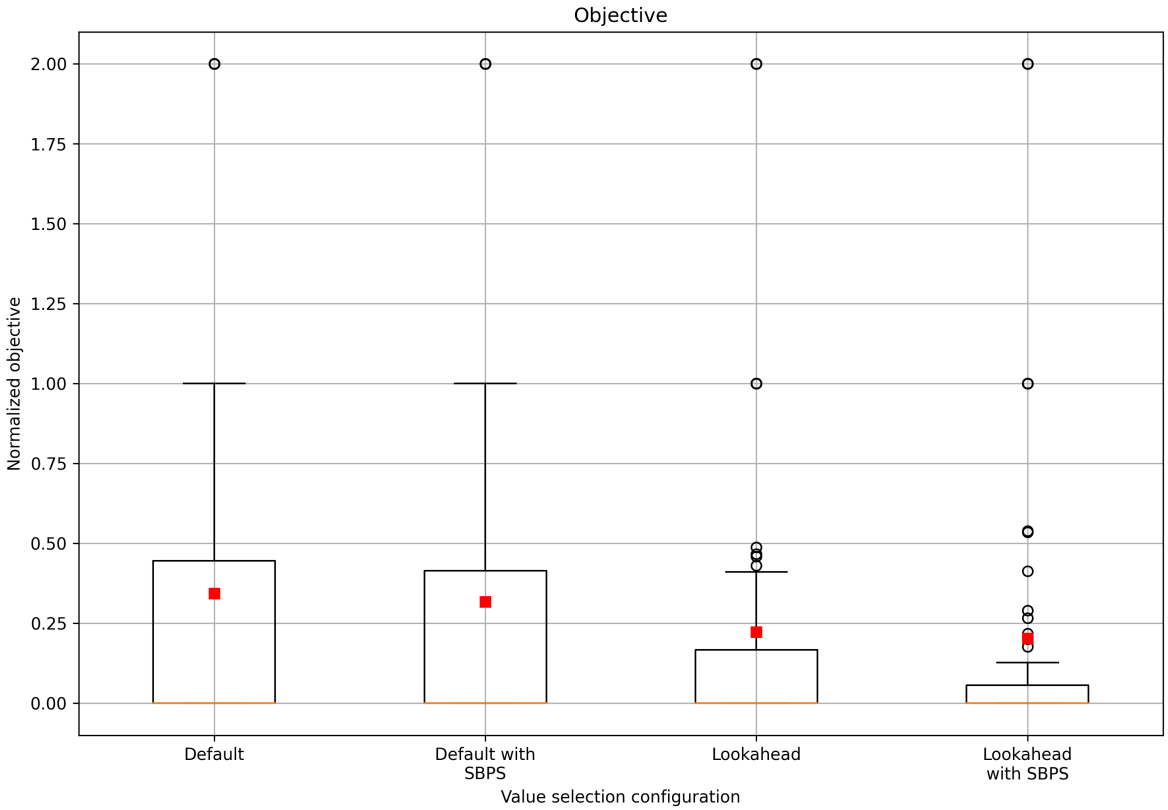
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.87963	0.19366	0.596365	1	1.362735	2.132173	-1.3012	0.196587
Lookahead	0.863587	0.010923	0.928092	1.245853	1.543004	2.309313	-0.54619	0.586320
Lookahead with SBPS	0.777576	0.000942	0.519112	1.163622	2.107895	4.222537	-0.87793	0.382370

Figure 7.11: Average solve time for each configuration, including only instances where at least one configuration terminated before the time limit (45 instances)

Figure 7.11 shows the solve time results, but filtered to include only instances where at least one configuration managed to terminate before the 10-minute time limit. The recorded means are slightly lower, with a value of 86.36% and 77.76% for the lookehaead and lookahead with SBPS configurations respectively. Again, the p-values are similar to the ones recorded for figure 7.10, and therefore, the null hypothesis cannot be rejected.

The median for both lookahead configurations is higher than the default configuration, with 124.59% for the lookahead configuration, and 116.36% for the lookahead configuration with SBPS. This indicates that similar to cluster editing, the lookahead approach potentially causes an overall time penalty to the total solve time. However, for some instances, the lookahead configuration significantly improves the solve time, leading to the mean being skewed towards a lower solve time.

The first reason why the lookahead configuration has a positive impact on the solve time for some problems is that finding an improved solution faster allows the solver to create stricter domains, creating many new nogoods, and decreasing the total search space that has to be explored. Next, since the lookahead approach performs a sat analysis when a lookahead leads to a conflict, and furthermore has a fail first strategy, where it biases decisions that decrease the domain, it likely also creates many new clauses that limit the search space even more. Therefore, it appears that overall, these performance benefits are able to better overcome the time overhead of the lookahead approach for certain instances.

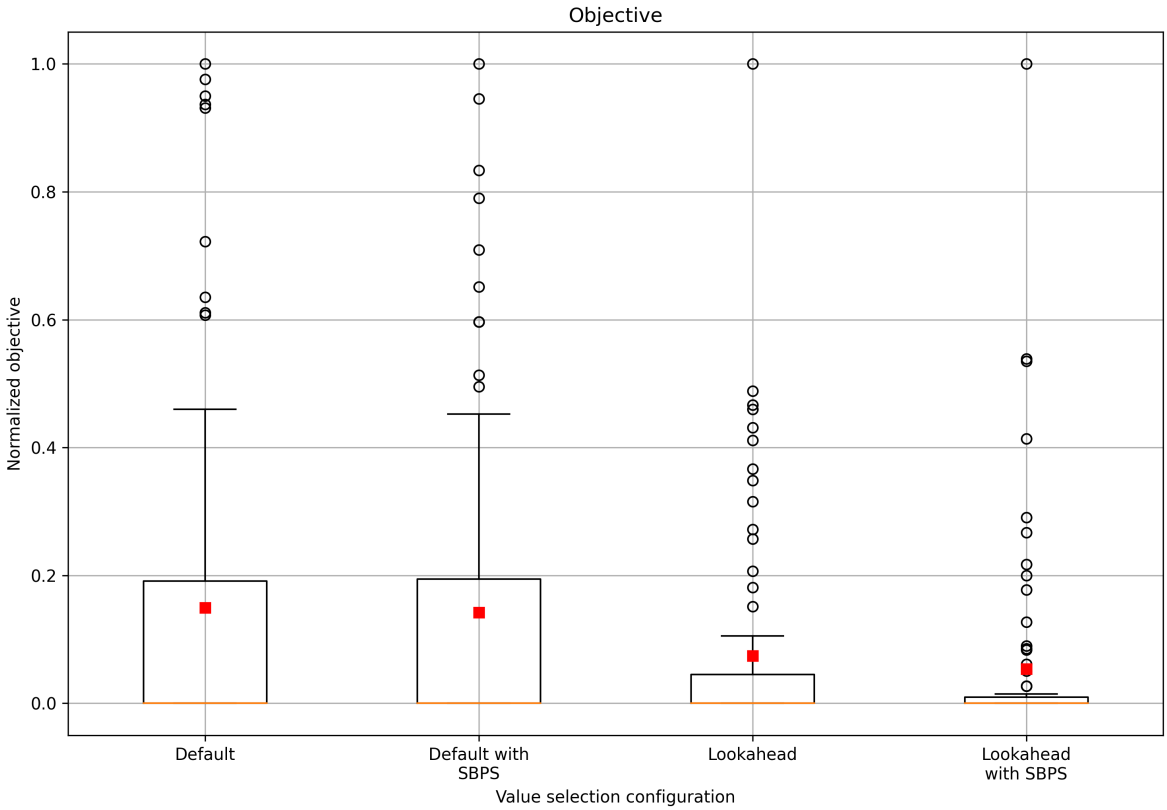


Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.343589	0	0	0.000396	0.446037	1	0	1
Default with SBPS	0.317792	0	0	0	0.41497	1	-0.31251	0.754997
Lookahead	0.223408	0	0	0	0.166477	0.411111	-1.54846	0.123194
Lookahead with SBPS	0.202514	0	0	0	0.055776	0.12727	-1.8143	0.071225

Figure 7.12: Best objective found after 10 minutes

Continuing, figure 7.12 shows the average best normalized objective found after 10 minutes. Here, 1 indicates the worse solution found throughout the search by all solvers in this instance that satisfies all constraints, while 0 indicates the best overall solution. A solution of 2 indicates that no solution was found. For each configuration, in the order presented, a mean of 0.3436, 0.3178, 0.2234, and 0.2025 was recorded.

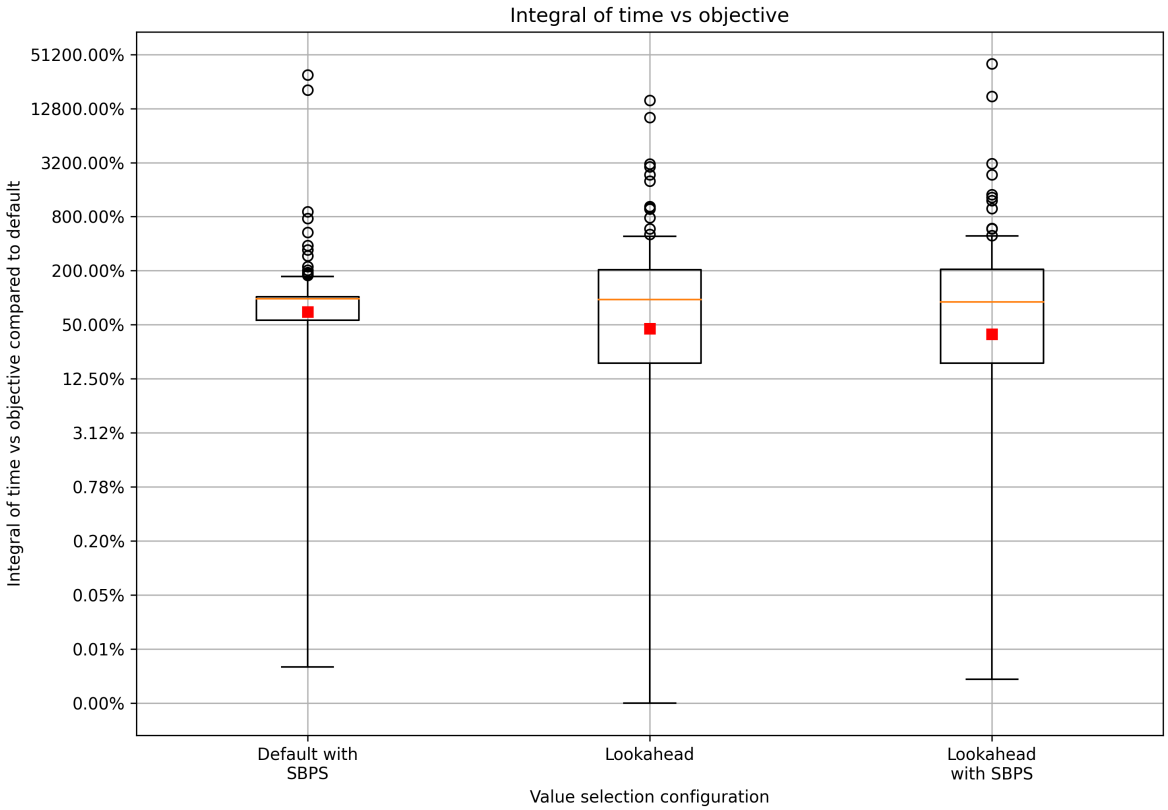
Although the means of the lookahead configurations are lower than the default configuration, with p-values of 0.12 and 0.07, the null hypothesis cannot be rejected. One reason for this is likely due to the high variance introduced by instances where no solution was found within the time limit. Therefore, it is likely that removing these values could result in a lower p-value.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.149093	0	0	0	0.191218	0.459642	0	1
Default with SBPS	0.141649	0	0	0	0.194444	0.452555	-0.17672	0.859957
Lookahead	0.074408	0	0	0	0.044602	0.105263	-2.05891	0.041166
Lookahead with SBPS	0.053656	0	0	0	0.009467	0.014514	-2.6927	0.007862

Figure 7.13: Best objective found after 10 minutes, including only instances where all configurations found a satisfying solution or terminated within the time limit (79 instances)

Figure 7.13 shows the average best normalized objective found after 10 minutes, excluding instances where at least one configuration did not find a solution within the time limit. With the filtered results, the p-values for the lookahead configuration are under 0.05, indicating that the null hypothesis can be rejected. Therefore, it can be stated with high confidence that for problems where a solution can be found within a time limit, the solution produced by the lookahead configuration will be on average closer to the optimal than the solution produced by the default configuration.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.698456	7.72E-05	0.559467	0.972961	1.027019	1.723453	-1.99523	0.047462
Lookahead	0.452958	3.05E-05	0.186817	0.950841	2.049801	4.838752	-2.75152	0.006513
Lookahead with SBPS	0.394823	5.64E-05	0.186463	0.894833	2.065496	4.86657	-3.12483	0.002061

Figure 7.14: Average integral for each configuration

Next, figure 7.14 shows the integral results compared to the default configuration. Overall, a mean of 45.26% is observed for the lookahead approach, and a mean of 39.48% is observed for the lookahead configuration with SBPS. Based on these results, low enough p-values were recorded, which provides evidence to reject the null hypothesis for all configurations. These results show the main advantage of the lookahead configuration, enabling it to find near-optimal solutions significantly faster than the default configuration, by better guiding the solver towards near-optimal solutions significantly faster than the default value selection.

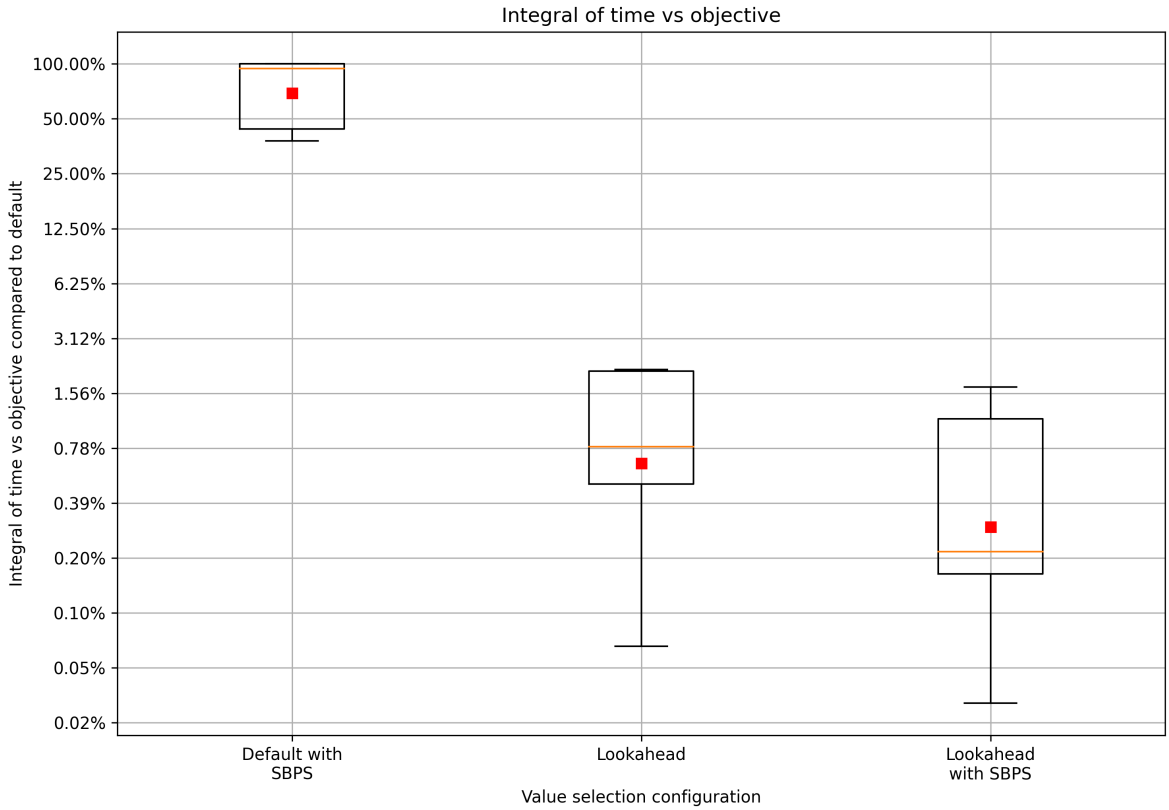
SBPS was able to have a mean of 69.85%. However, when looking at the IQR for SBPS, it ranges from 55.94%, up to 102.70%, while the range for the lookahead and lookahead with SBPS ranges from 18.68% to 204.98%, and from 18.64%, up to 206.54% respectively. This indicates a higher variance in the effectiveness of the lookahead approach when compared to SBPS, with many instances resulting in an overall worse integral performance.

7.4. MiniZinc challenge individual results

In this section, certain special cases that are outliers compared to the average results will be presented. These results better show how this configuration performs on certain edge case problems. For these results, it is important to note the limited amount of instances, with each problem having only 5 instances, which can skew the results heavily.

7.4.1. Schelduling

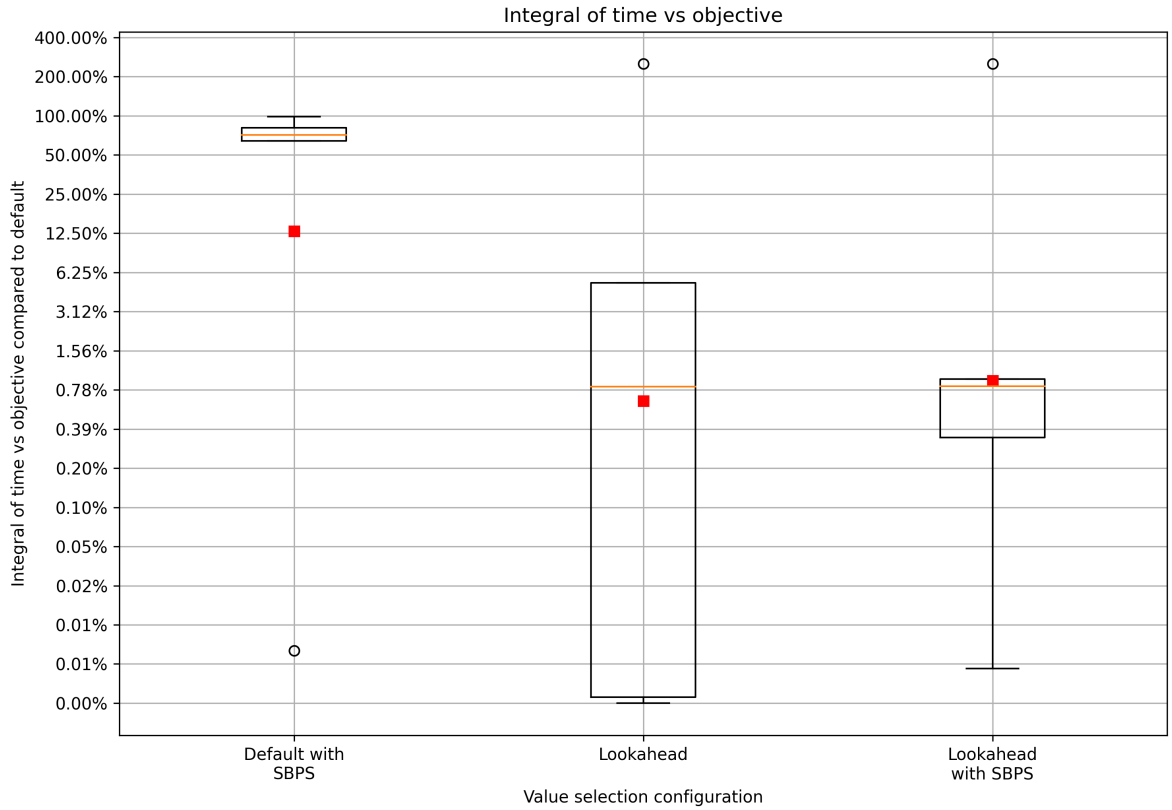
The *gfd-schedule* problem is a scheduling problem with the following constraints. First, each item is grouped by its *kind*. Each item is processed by a *group* using a *facility*. Each item has a *produced day* that it has to be processed after. Last, the maximum number of processed items per day is fixed. The objective is to minimize the use of the *facilities* and a *deadline penalty* that is added when items are processed after their *deadline*.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.690231	0.378159	0.43922	0.943224	1	1	-1.711156	0.125416
Lookahead	0.006442	0.000641	0.004969	0.007975	0.020674	0.021129	-7.87458	4.89E-05
Lookahead with SBPS	0.002887	0.000312	0.001594	0.002112	0.011292	0.016908	-8.114235	3.94E-05

Figure 7.15: Average integral for gfd-schedule

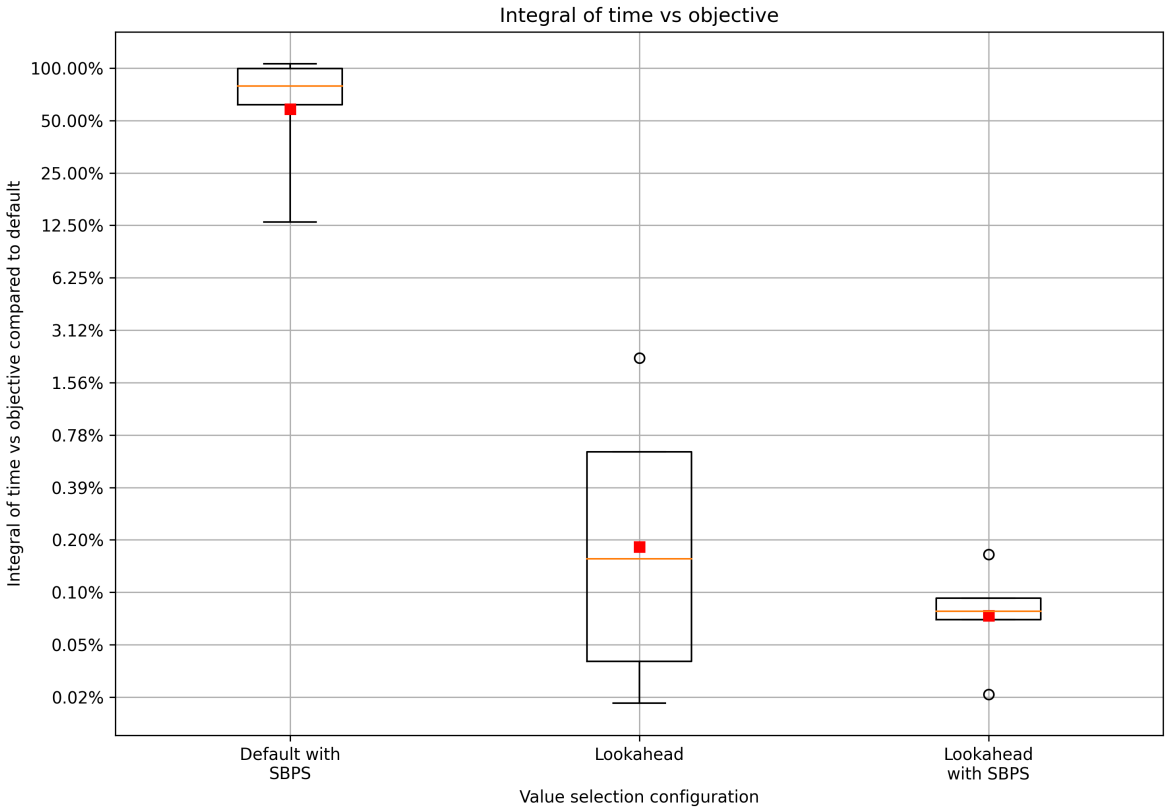
Figure 7.15 shows the integral results for the problem *gfd-schedule*. This problem presents the biggest overall improvement for the lookahead approach. The lookahead approach has a significantly faster solve time, with the lookahead configuration having a mean solve time of 2.53%, and the lookahead configuration with SBPS 1.51%. More notably, the default and SBPS configurations managed to find a solution and solve only 2 out of the 5 instances, while the two lookahead configurations were able to solve all 5 of the instances.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.129464	0.638874	0.638874	0.713348	0.809929	0.985332	-1.13631	0.288718
Lookahead	0.006399	3.05E-05	3.38E-05	0.008291	0.05173	0.05173	-2.52505	0.035528
Lookahead with SBPS	0.009197	5.64E-05	0.003349	0.008338	0.00942	0.00942	-2.81257	0.022753

Figure 7.16: Average integral for nfc

NFC, another scheduling problem that is modelled like a network graph, with the objective of minimizing the network flow also was able to perform significantly better using the lookahead approach, as shown in figure 7.16. with a mean integral of 0.64% and 0.92% for the lookahead configurations without and with SBPS respectively.



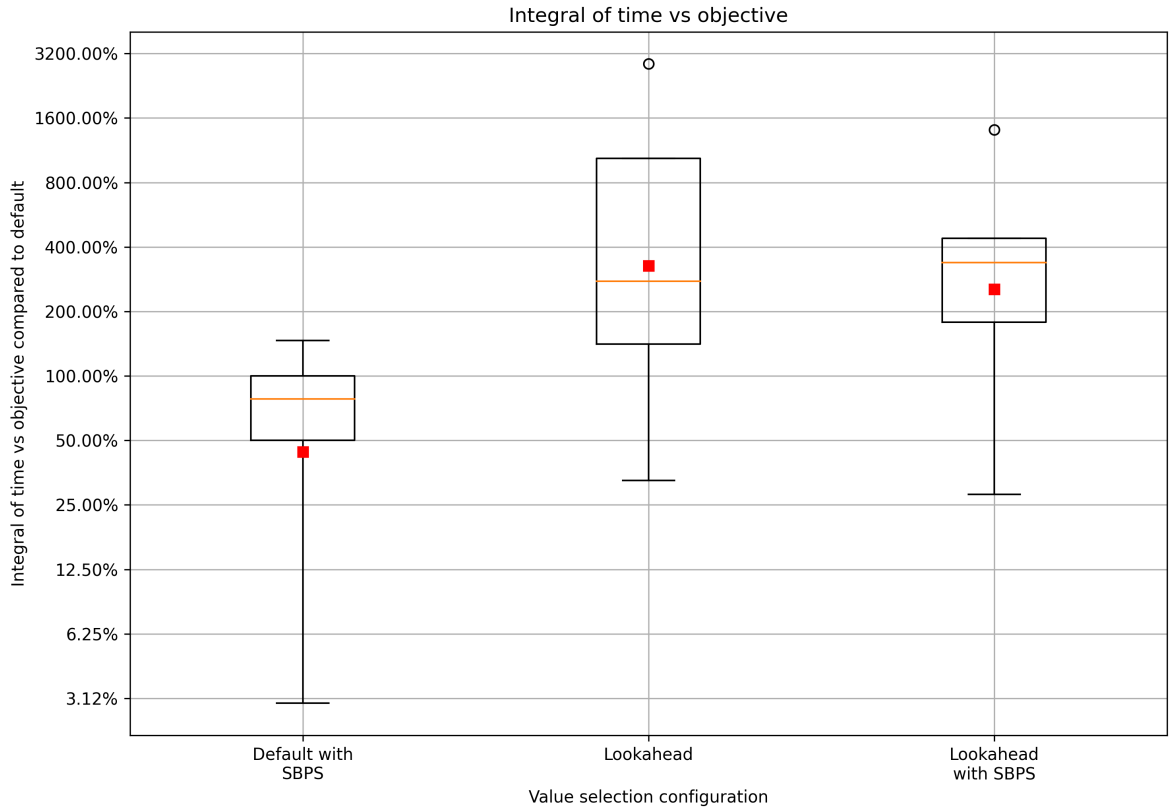
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.582046	0.130631	0.616329	0.78795	0.995581	1.057672	-1.40385	0.19797
Lookahead	0.001784	0.000225	0.000391	0.00152	0.006247	0.006247	-7.45778	7.21E-05
Lookahead with SBPS	0.000717	0.000682	0.000682	0.000759	0.000904	0.000904	-24.1287	9.28E-09

Figure 7.17: Average integral for wordpress

Continuing, *wordpress* is a scheduling problem concerning the assignment of certain components with *minimum required specs* to *virtual machines(VM)*, where each *VM* has certain *specs* and a *cost*. The objective is to minimize the total *cost* of the used *VMs*. Figure 7.17 shows the results for this problem. In particular, the mean for the lookahead configuration is 0.18%, and for the lookahead with SBPS 0.07%.

7.4.2. Diameter-constrained minimum spanning tree

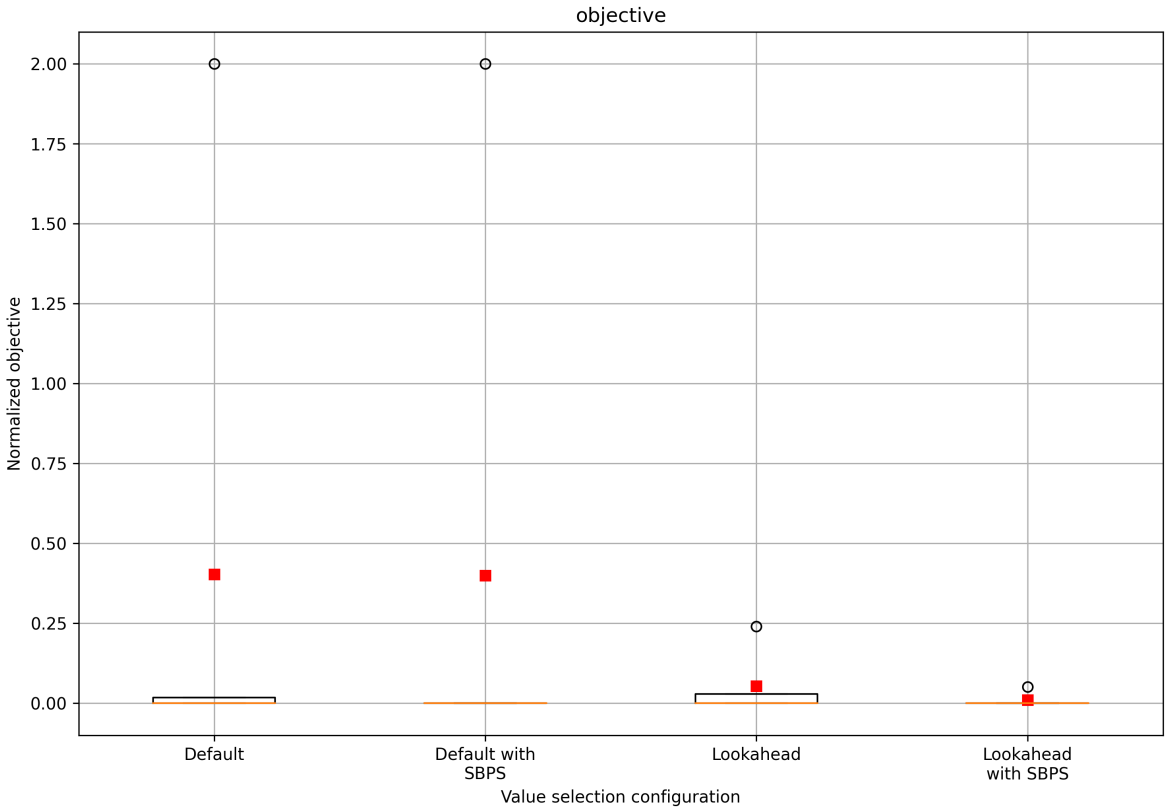
For the *diameter-constrained minimum spanning tree* problem, given an undirected weighted graph $G = (V, E)$, and an integer D, the goal is to find a spanning tree from G with minimum total weight costs, where the shortest path between any two nodes in the tree, in other words, the diameter of the tree is less than D.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.442305	0.029646	0.501194	0.779682	1	1.461253	-1.16924	0.275962
Lookahead	3.271864	0.325656	1.405826	2.763749	10.35485	10.35485	1.524203	0.165964
Lookahead with SBPS	2.535984	0.279567	1.784186	3.388908	4.396974	4.396974	1.442874	0.187042

Figure 7.18: Average integral for diameterc-mst

First, figure 7.18 shows the integral results for this problem. For the lookahead approach a value of 327.18% is observed. However, when looking at the average objective found after 10 minutes, shown in figure 7.19, the lookahead approach appears superior in comparison to the default configuration. Looking at the individual instances, all configurations managed to find optimality for 3 of the 5 instances. However, the non-lookahead configurations managed to find a solution only for 4 instances. This is represented in figure 7.19 by the outlier values with a normalized objective value of 2. Meanwhile, both lookahead configurations managed to find a solution for all 5 instances.

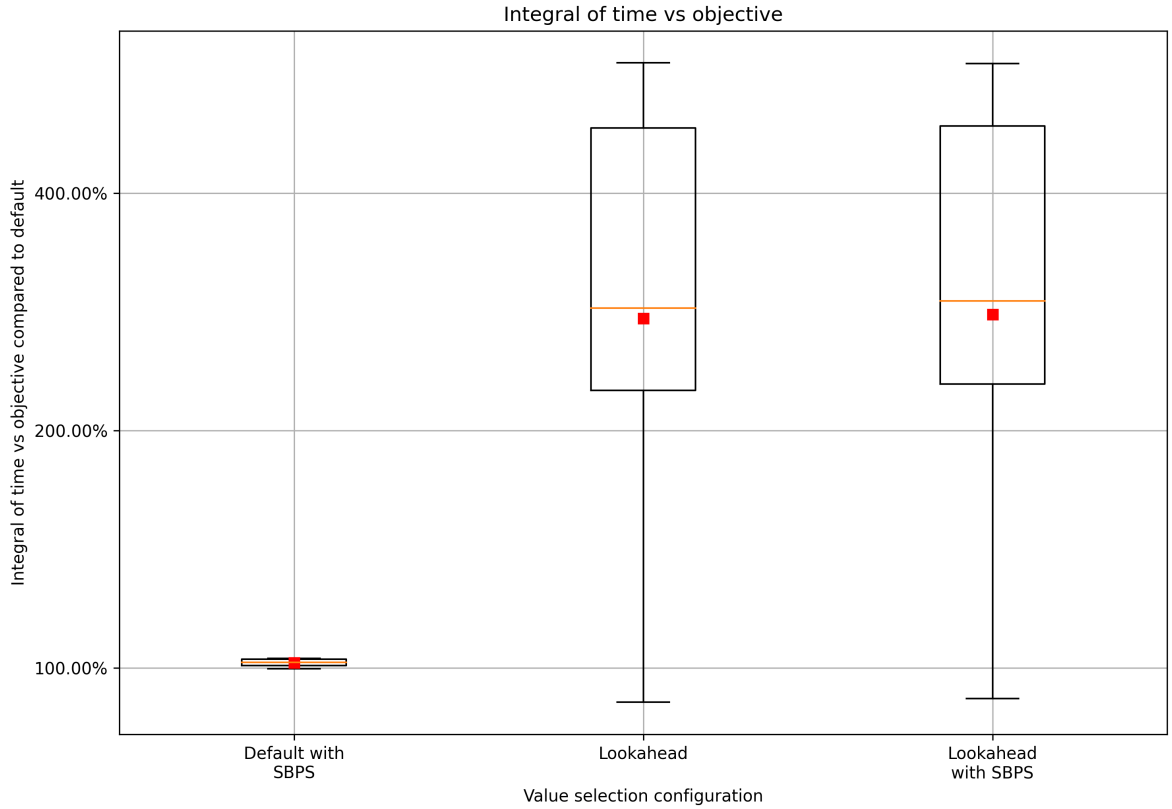


Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.403623	0	0	0	0.018116	0.018116	0	1
Default with SBPS	0.4	0	0	0	0	0	0.006412	0.995041
Lookahead	0.053797	0	0	0	0.028986	0.028986	0.870529	0.409372
Lookahead with SBPS	0.010145	0	0	0	0	0	0.985572	0.353209

Figure 7.19: Average objective for diameterc-mst

7.4.3. Sudoku

The following problem is a variation of the *sudoku* problem, with the addition of an optimization function with the goal of minimizing or maximizing the values of the cells depending on if they are odd or even.



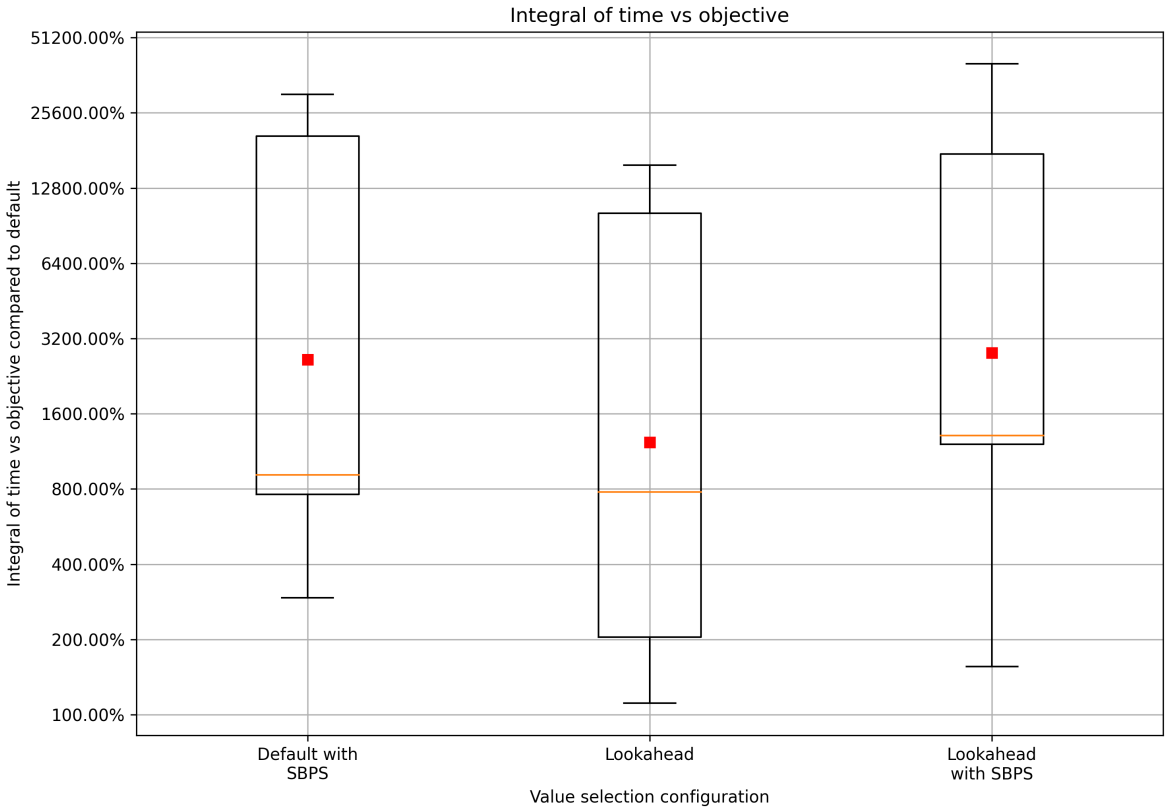
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	1.014769	0.997167	1.006864	1.01611	1.025466	1.028571	2.549092	0.034223
Lookahead	2.776173	0.904762	2.249339	2.86119	4.838752	5.852795	3.101901	0.014624
Lookahead with SBPS	2.806016	0.914286	2.291657	2.922096	4.86657	5.838509	3.154064	0.013515

Figure 7.20: Average integral for sudoku_opt

Figure 7.20 shows the result for the integral of the sudoku problem. The lookahead configurations without and with SBPS have a mean of 277.62% and 280.60%. For all instances and configurations, the first solution found was optimal. Therefore, this problem does not benefit from a heuristic designed to find optimal solutions, and hence, the lookahead approach has no positive effect on performance, while adding a significant time overhead.

7.4.4. Placing "Hearts" in Equilateral Triangular Grids

The following problem has the objective of maximizing the number of *hearts*as defined by the model, that can be placed inside an equilateral triangular grid with sides N. At the same time, the hearts cannot be placed lying on the corners of any possible equilateral triangle of any size or orientation.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	26.36469	2.935132	7.607625	9.121606	206.4276	302.9683	3.479405	0.008327
Lookahead	12.30881	1.11121	2.04019	7.771895	101.4007	158.1414	2.49844	0.037032
Lookahead with SBPS	28.0342	1.555162	12.07065	13.0853	175.4266	401.8426	3.324643	0.010467

Figure 7.21: Average integral for triangular

Figure 7.21 shows the results of the integral for this problem, where the lookahead approach had a mean integral of 1230.88%. Notably, SBPS also has a similar high integral value. Based on this, it appears that using a generalized heuristic value selection to attempt to improve the optimal value for this problem is an ineffective strategy.

7.5. Results discussion

Based on the results, it appears that the lookahead approach can provide significant advantages to Chuffed, not only for cluster editing but in general for many optimization problems. In particular, the lookahead approach assists the solver in finding better solutions early on.

The main advantage of the lookahead approach over the default VSIDS configuration and SBPS, is finding near-optmial solutions faster than the default configuration, as shown in figure 7.14. Furthermore, for larger instances, the lookahead configuration finds on average solutions closer to the optimal, as shown in figures 7.7 and 7.13. Continuing, combining the lookahead approach with SBPS, generally appears also to have a positive effect, but this advantage varies heavily depending on the problem.

The biggest drawback of the lookahead approach is the additional time overhead added by performing the lookahead propagations. As a result, the lookahead configuration often has an overall slower solve time, despite the improvements provided early on. Decreasing the performance overhead of the lookahead propagations, along with improving the stopping criterion of the lookahead approach would contribute greatly to decreasing this performance disadvantage.

Furthermore, despite the lookahead approach being efficient at finding near-optimal solutions it appears that the performance advantage shrinks when it comes to finding the optimal solution. This is likely caused by the additional overhead added by the lookahead propagations, and the lookahead approach being prone to reaching local minima.

Lastly, the lookahead approach seems to have varying performance depending on the problem. In particular, the two lookahead configurations performed best on scheduling problems, where the quality of the solutions at each point in time was on average significantly better than the default and SBPS configurations. In addition, on the scheduling problems, the solve time was also significantly faster for the lookahead configurations.

8

Conclusion & Future Improvements

This chapter discusses the main takeaways of the research, followed by future improvements that could be added to the lookahead approach to improve its performance.

8.1. Conclusion

When it comes to finding near-optimal or optimal solutions for optimization problems, there is a huge performance disparity between heuristic algorithms and constraint programming solvers. The research goal was to better understand how the behaviour of specialized heuristic solvers can assist constraint programming solvers in optimization problems.

Using the cluster editing problem as a case study, it was shown that combing the value selection of Chuffed with a heuristic algorithm specialized for solving the cluster editing instances, in particular, the Kanpai algorithm [10], could provide significant performance benefits to the performance of the CP solver. In particular, the combined algorithm showed a significant improvement in the time it takes to find near-optimal and optimal solutions.

Inspired by the steps taken in each interaction of the Kanpai algorithm [10], a generalized approach, which performs lookahead propagations during value selection and then selects the value that has the most optimal objective value within its domain was developed. This lookahead approach was able to find near-optimal solutions for many problems significantly faster than a configuration of Chuffed using just VSIDS, emulating the performance observed by heuristic solvers. In particular, on average, the lookahead configuration had a definite integral for the time vs objective graph 54.70% lower than the default Chuffed configuration on the generalized test suite.

Despite the overall improvements of the lookahead approach, performing lookahead propagations has a significant time cost. Therefore, the time taken to solve an instance increased for most problems, despite the benefits of the approach. Furthermore, the time penalty also affects the early solution quality for many problems that do not greatly benefit from lookahead.

Overall, the lookahead approach presents a way to better emulate the performance of heuristic solvers, in particular, finding near-optimal solutions faster for certain problems, but causes an overall slight increase in the total solve time.

8.2. Future Improvements

The main improvements for the lookahead approach concern decreasing the time penalty of the additional lookahead propagations.

The first improvement would be to reuse the results of one of the lookahead propagations. The propagation following the value selection should be equivalent to the lookahead propagation for the same value. Therefore, using the outcome of the lookahead propagation, instead of recomputing the propagation after selecting a value would help reduce the time penalty, as this would result in almost 1 additional propagation, instead of 2.

Continuing, the two lookahead propagation could run in parallel, as the two propagations do not rely on the outcome of the other in order to compute their result. This should further reduce the time penalty of the lookahead configuration.

Next, performing a lookahead for multiple variables, instead of the two values for a single variable could improve the branching decisions made by the solver. In particular, it would potentially reduce cases where the solver reaches local minima and is unable to improve the objective, as it would provide more possible decisions at each level.

Last, an improved stopping criterion that more accurately stops the lookahead approach when it can no longer provide significant benefits to the solver could improve the overall solve time.

References

[1] N. Bansal, A. Blum, and S. Chawla. "Clustering Gene Expression Patterns". In: *Journal of Computational Biology* (2004). DOI: https://doi.org/10.1023/B:MACH.0000033116.57574.95.

[2] N. Bansal, A. Blum, and S. Chawla. "Correlation Clustering". In: *Machine Learning* (2004). DOI: https://doi-org.tudelft.idm.oclc.org/10.1023/B:MACH.0000033116.57574.95.

[3] L. Bastos et al. "Efficient algorithms for cluster editing". In: *Journal of Combinatorial Optimization* (2016). DOI: https://doi.org/10.1007/s10878-014-9756-7.

[4] A. Ben-Dor, R. Shamir, and Z. Yakhini. "Clustering Gene Expression Patterns". In: *Journal of Computational Biology* (1999). DOI: http://doi.org/10.1089/106652799318274.

[5] T. Bläsius et al. "PACE Solver Description: KaPoCE: A Heuristic Cluster Editing Algorithm". In: *Dagstuhl Research Online Publication Server* (2021). DOI: 10.4230/LIPIcs.IPEC.2021.31.

[6] Daniel G. Bobrow and Bertram Raphael. "New Programming Languages for Artificial Intelligence Research". In: *ACM Comput. Surv.* 6.3 (1974), 153–174. ISSN: 0360-0300. DOI: 10.1145/356631.356632. URL: https://doi.org/10.1145/356631.356632.

[7] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. "Propositional Satisfiability and Constraint Programming: A Comparative Survey". In: *ACM Comput. Surv.* 38.4 (2006), 12–es. ISSN: 0360-0300. DOI: 10.1145/1177352.1177354. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1177352.1177354.

[8] G. Chu et al. *Chuffed, a lazy clause generation solver*. 2023. URL: https://github.com/chuffed/chuffed (visited on 03/23/2023).

[9] Alain Colmerauer and Philippe Roussel. "The Birth of Prolog". In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, 331–367. ISBN: 0201895021. URL: https://doi.org/10.1145/234286.1057820.

[10] E. Demirović. *Kanpai: A Bottom-Up Approach for Cluster Editing (PACE 2021)*. 2021. URL: https://bitbucket.org/EmirD/pace-2021/src/master/ (visited on 03/05/2023).

[11] E. Demirović, G. Chu, and P.J. Stuckey. "Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers". In: *International Conference on Principles and Practice of Constraint Programming* (2018). DOI: https://doi.org/10.1007/978-3-319-98334-9_7.

[12] R. van Driel, E. Demirović, and N. Yorke-Smith. "Learning Variable Activity Initialisation for Lazy Clause Generation Solvers". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2021). DOI: https://doi-org.tudelft.idm.oclc.org/10.1007/978-3-030-78230-6_4.

[13] R. A. van Driel. "Unsatisfiable core learning for Chuffed". In: *Delft University of Technology* (2020). DOI: http://resolver.tudelft.nl/uuid:18359282-e769-47d5-ac62-4ee6e1ec454e.

[14] P.J. Stuckey E. Demirović. "Constraint Programming for High School Timetabling: A Scheduling-Based Model with Hot Starts". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2018). DOI: https://doi.org/10.1007/978-3-319-93031-2_10.

[15] Google. *OR-Tools | Google for Developers*. 2023. URL: https://developers.google.com/optimization (visited on 06/19/2023).

[16] J. Gramm et al. "Graph-Modeled Data Clustering: Exact Algorithms for Clique Generation". In: *Theory of Computing Systems* (2005). DOI: https://doi-org.tudelft.idm.oclc.org/10.1007/s00224-004-1178-y.

[17] Robert M. Haralick and Gordon L. Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial Intelligence* 14.3 (1980), pp. 263–313. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(80)90051-X. URL: https://www.sciencedirect.com/science/article/pii/000437028090051X.

[18] Q. He et al. "Combining Clickstream Analyses and Graph-Modeled Data Clustering for Identifying Common Response Processes". In: *Psychometrika* (2021). DOI: https://doi.org/10.1007/s11336-020-09743-0.

[19] Leon Kellerhals et al. "The PACE 2021 Parameterized Algorithms and Computational Experiments Challenge: Cluster Editing". In: *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*. Ed. by Petr A. Golovach and Meirav Zehavi. Vol. 214. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 26:1–26:18. ISBN: 978-3-95977-216-7. DOI: 10.4230/LIPIcs.IPEC.2021.26. URL: https://drops.dagstuhl.de/opus/volltexte/2021/15409.

[20] Lehrstuhl Bioinformatik Jena. *Cluster Editing evaluation data*. 2023. URL: https://bio.informatik.uni-jena.de/data/#cluster_editing_data (visited on 06/06/2023).

[21] Jia Liang et al. "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers". In: June 2015. ISBN: 978-3-319-26286-4. DOI: 10.1007/978-3-319-26287-1_14.

[22] E. Lucas. *Récréations Mathématiques*. Gauthier-Villars, Paris, 1891.

[23] MiniZinc. *MiniZinc*. 2022. URL: https://www.minizinc.org (visited on 03/23/2023).

[24] MiniZinc. *MiniZinc Challenge 2022 Results*. 2022. URL: https://www.minizinc.org/challenge2022/results2022.html (visited on 03/23/2023).

[25] J.J. Wallace P. J. Fleming. "How not to lie with statistics: the correct way to summarize benchmark results." In: *Communications of the ACM* (1986). DOI: doi.org/10.1145/5666.5673.

[26] D. Rhebergen. "Cluster Editing with Diamond-free Vertices". In: *Delft University of Technology* (2021). DOI: http://resolver.tudelft.nl/uuid:33ad00d5-b520-489b-a376-15eec82ae10b.

[27] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. First edition. Elsevier, 2006. ISBN: 9780080463803.

[28] Peter Stuckey. "Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving". In: June 2010, pp. 5–9. ISBN: 978-3-642-13519-4. DOI: 10.1007/978-3-642-13520-0_3.

[29] M. Wallace and N. Yorke-Smith. "A new constraint programming model and solving for the cyclic hoist scheduling problem". In: *Constraints* (2020). DOI: https://doi-org.tudelft.idm.oclc.org/10.1007/s10601-020-09316-z.

[30] Toby Walsh. *SAT vs CSP: a commentary*. 2019. arXiv: 1910.00128 [cs.AI].

[31] G. Weil et al. "Constraint programming for nurse scheduling". In: *IEEE* (1995). DOI: 10.1109/51.395324.

A

Cluster Editing Results

A.1. Number of nodes

Table A.1: Total number of nodes

	Default	Simple heuristic	Kanpai	Warm start	Lookahead
exact003	315511 (1.000)	389791 (1.235)	287315 (0.911)	499527 (1.583)	332185 (1.053)
exact005	1426230 (1.000)	1423018 (0.998)	1403259 (0.984)	1545869 (1.084)	1268924 (0.890)
instance_nr_11-csv-thres-0-40	324504 (1.000)	193377 (0.596)	175689 (0.541)	208178 (0.642)	189266 (0.583)
instance_nr_313-csv-thres-0-45	439277 (1.000)	397769 (0.906)	258276 (0.588)	247629 (0.564)	315441 (0.718)
instance_nr_1243-csv-thres-0-50	708572 (1.000)	416023 (0.587)	544836 (0.769)	681669 (0.962)	767368 (1.083)
instance_nr_1679-csv-thres-0-45	1552858 (1.000)	1553092 (1.000)	1894593 (1.220)	2376777 (1.531)	2082946 (1.341)
pace_actionseq_21_10	270194 (1.000)	238148 (0.881)	279698 (1.035)	410885 (1.521)	271558 (1.005)
pace_actionseq_22_2	188794 (1.000)	44028 (0.233)	108591 (0.575)	114397 (0.606)	640672 (3.393)
pace_actionseq_23_10	188985 (1.000)	232100 (1.228)	365258 (1.933)	478510 (2.532)	235521 (1.246)
pace_actionseq_26_2	621440 (1.000)	341833 (0.550)	338780 (0.545)	413545 (0.665)	291556 (0.469)

Table A.2: Total number of nodes for SBPS configurations

	Default with SBPS	**Simple heuristic with SBPS**	**Kanpai with SBPS**	**Warm start with SBPS**	**Lookahead with SBPS**
exact003	279715 (0.887)	343024 (1.087)	262154 (0.831)	328724 (1.042)	233560 (0.740)
exact005	3641569 (2.553)	1590796 (1.115)	1845364 (1.294)	1149937 (0.806)	1367305 (0.959)
instance_nr_11-csv-thres-0-40	741256 (2.284)	176415 (0.544)	207317 (0.639)	208443 (0.642)	146059 (0.450)
instance_nr_313-csv-thres-0-45	506058 (1.152)	329017 (0.749)	325454 (0.741)	254638 (0.580)	361069 (0.822)
instance_nr_1243-csv-thres-0-50	989786 (1.397)	547358 (0.772)	869365 (1.227)	992154 (1.400)	759312 (1.072)
instance_nr_1679-csv-thres-0-45	5854636 (3.770)	2069092 (1.332)	1576226 (1.015)	2604950 (1.678)	3558017 (2.291)
pace_actionseq_21_10	334407 (1.238)	272154 (1.007)	314176 (1.163)	354241 (1.311)	438952 (1.625)
pace_actionseq_22_2	94262 (0.499)	83288 (0.441)	114600 (0.607)	106170 (0.562)	465971 (2.468)
pace_actionseq_23_10	464237 (2.456)	359054 (1.900)	248883 (1.317)	286850 (1.518)	404107 (2.138)
pace_actionseq_26_2	16705175 (26.881)	431388 (0.694)	647163 (1.041)	487535 (0.785)	266252 (0.428)

A.2. Solve time

Table A.3: Solve time (seconds)

	Default	Simple heuristic	Kanpai	Warm start	Lookahead
exact003	40.435 (1.000)	73.044 (1.806)	57.184 (1.414)	71.208 (1.761)	69.828 (1.727)
exact005	210.067 (1.000)	284.497 (1.354)	293.532 (1.397)	239.771 (1.141)	245.850 (1.170)
instance_nr_11-csv-thres-0-40	45.624 (1.000)	41.885 (0.918)	38.903 (0.853)	28.396 (0.622)	40.281 (0.883)
instance_nr_313-csv-thres-0-45	54.413 (1.000)	76.165 (1.400)	53.357 (0.981)	33.964 (0.624)	71.654 (1.317)
instance_nr_1243-csv-thres-0-50	121.354 (1.000)	84.779 (0.699)	123.539 (1.018)	116.324 (0.959)	173.566 (1.430)
instance_nr_1679-csv-thres-0-45	234.113 (1.000)	349.119 (1.491)	409.263 (1.748)	347.878 (1.486)	442.657 (1.891)
pace_actionseq_21_10	42.379 (1.000)	45.150 (1.065)	43.756 (1.032)	62.221 (1.468)	55.346 (1.306)
pace_actionseq_22_2	25.601 (1.000)	7.625 (0.298)	22.488 (0.878)	14.876 (0.581)	139.204 (5.437)
pace_actionseq_23_10	19.203 (1.000)	36.831 (1.918)	68.327 (3.558)	49.844 (2.596)	40.450 (2.106)
pace_actionseq_26_2	133.104 (1.000)	99.521 (0.748)	97.532 (0.733)	88.565 (0.665)	86.758 (0.652)

Table A.4: Solve time for SBPS configurations (seconds)

	Default with SBPS	**Simple heuristic with SBPS**	**Kanpai with SBPS**	**Warm start with SBPS**	**Lookahead with SBPS**
exact003	42.232 (1.044)	81.924 (2.026)	52.721 (1.304)	56.722 (1.403)	48.959 (1.211)
exact005	696.317 (3.315)	303.577 (1.445)	352.612 (1.679)	179.664 (0.855)	278.401 (1.325)
instance_nr_11-csv-thres-0-40	127.761 (2.800)	35.354 (0.775)	42.428 (0.930)	30.027 (0.658)	35.543 (0.779)
instance_nr_313-csv-thres-0-45	64.360 (1.183)	61.787 (1.136)	65.208 (1.198)	38.970 (0.716)	69.891 (1.284)
instance_nr_1243-csv-thres-0-50	207.621 (1.711)	118.651 (0.978)	197.653 (1.629)	196.360 (1.618)	182.570 (1.504)
instance_nr_1679-csv-thres-0-45	1099.442 (4.696)	493.302 (2.107)	350.467 (1.497)	391.360 (1.672)	838.225 (3.580)
pace_actionseq_21_10	60.554 (1.429)	55.869 (1.318)	52.207 (1.232)	65.583 (1.548)	82.944 (1.957)
pace_actionseq_22_2	12.303 (0.481)	15.235 (0.595)	25.356 (0.990)	13.080 (0.511)	93.700 (3.660)
pace_actionseq_23_10	67.064 (3.492)	55.534 (2.892)	45.907 (2.391)	39.780 (2.072)	71.816 (3.740)
pace_actionseq_26_2	3595.310 (27.011)	146.465 (1.100)	189.479 (1.424)	101.861 (0.765)	77.195 (0.580)

A.3. Time taken to find the optimal solution

Table A.5: Time taken to find the optimal solution (seconds)

	Default	**Simple heuristic**	**Kanpai**	**Warm start**	**Lookahead**
exact003	13.543 (1.000)	62.317 (4.601)	45.296 (3.345)	60.700 (4.482)	42.289 (3.123)
exact005	53.520 (1.000)	92.331 (1.725)	160.825 (3.005)	143.383 (2.679)	28.541 (0.533)
instance_nr_11-csv-thres-0-40	34.134 (1.000)	0.052 (0.002)	0.021 (0.001)	0.019 (0.001)	11.975 (0.351)
instance_nr_313-csv-thres-0-45	43.132 (1.000)	71.203 (1.651)	0.012 (0.000)	0.013 (0.000)	25.966 (0.602)
instance_nr_1243-csv-thres-0-50	95.912 (1.000)	44.820 (0.467)	44.200 (0.461)	109.177 (1.138)	123.196 (1.284)
instance_nr_1679-csv-thres-0-45	88.106 (1.000)	0.047 (0.001)	0.048 (0.001)	219.157 (2.487)	0.009 (0.000)
pace_actionseq_21_10	11.858 (1.000)	0.084 (0.007)	1.975 (0.167)	30.429 (2.566)	0.954 (0.080)
pace_actionseq_22_2	18.871 (1.000)	0.147 (0.008)	0.014 (0.001)	0.017 (0.001)	136.770 (7.248)
pace_actionseq_23_10	9.664 (1.000)	16.699 (1.728)	0.032 (0.003)	44.452 (4.600)	24.707 (2.557)
pace_actionseq_26_2	94.505 (1.000)	0.022 (0.000)	0.023 (0.000)	0.024 (0.000)	0.005 (0.000)

Table A.6: Time taken to find the optimal solution for SBPS configurations (seconds)

	Default with SBPS	**Simple heuristic with SBPS**	**Kanpai with SBPS**	**Warm start with SBPS**	**Lookahead with SBPS**
exact003	8.201 (0.606)	47.553 (3.511)	30.996 (2.289)	7.428 (0.548)	5.992 (0.442)
exact005	591.768 (11.057)	156.913 (2.932)	0.838 (0.016)	74.166 (1.386)	76.859 (1.436)
instance_nr_11-csv-thres-0-40	113.477 (3.324)	0.053 (0.002)	0.021 (0.001)	0.020 (0.001)	7.969 (0.233)
instance_nr_313-csv-thres-0-45	44.519 (1.032)	31.420 (0.728)	0.011 (0.000)	0.015 (0.000)	31.935 (0.740)
instance_nr_1243-csv-thres-0-50	172.503 (1.799)	33.311 (0.347)	84.104 (0.877)	0.020 (0.000)	97.995 (1.022)
instance_nr_1679-csv-thres-0-45	848.331 (9.629)	0.046 (0.001)	0.034 (0.000)	0.084 (0.001)	0.009 (0.000)
pace_actionseq_21_10	15.993 (1.349)	0.083 (0.007)	0.312 (0.026)	0.031 (0.003)	15.820 (1.334)
pace_actionseq_22_2	5.121 (0.271)	0.129 (0.007)	0.014 (0.001)	0.016 (0.001)	92.419 (4.897)
pace_actionseq_23_10	44.918 (4.648)	26.776 (2.771)	2.100 (0.217)	1.983 (0.205)	33.856 (3.503)
pace_actionseq_26_2	3551.174 (37.577)	0.022 (0.000)	0.023 (0.000)	0.025 (0.000)	0.006 (0.000)

A.4. Integral

Table A.7: Integral of time (seconds) vs normalized edits plot

	Default	Simple heuristic	Kanpai	Warm start	Lookahead
exact003	0.733 (1.000)	4.232 (5.777)	1.638 (2.236)	6.305 (8.606)	2.172 (2.965)
exact005	2.132 (1.000)	1.831 (0.859)	3.169 (1.486)	2.823 (1.324)	0.564 (0.265)
instance_nr_11-csv-thres-0-40	3.124 (1.000)	0.023 (0.007)	0.021 (0.007)	0.019 (0.006)	0.480 (0.154)
instance_nr_313-csv-thres-0-45	2.975 (1.000)	4.858 (1.633)	0.012 (0.004)	0.013 (0.004)	3.598 (1.209)
instance_nr_1243-csv-thres-0-50	8.212 (1.000)	0.906 (0.110)	1.765 (0.215)	2.157 (0.263)	12.366 (1.506)
instance_nr_1679-csv-thres-0-45	2.911 (1.000)	0.022 (0.008)	0.021 (0.007)	2.132 (0.732)	0.003 (0.001)
pace_actionseq_21_10	0.561 (1.000)	0.017 (0.031)	0.150 (0.266)	1.234 (2.199)	0.097 (0.173)
pace_actionseq_22_2	2.247 (1.000)	0.030 (0.013)	0.014 (0.006)	0.017 (0.008)	13.622 (6.061)
pace_actionseq_23_10	0.520 (1.000)	0.926 (1.782)	0.020 (0.039)	4.976 (9.571)	1.544 (2.970)
pace_actionseq_26_2	7.052 (1.000)	0.022 (0.003)	0.023 (0.003)	0.024 (0.003)	0.005 (0.001)
exact015	149.965 (1.000)	32.268 (0.215)	0.124 (0.001)	1.630 (0.011)	38.293 (0.255)
exact020	179.890 (1.000)	101.100 (0.562)	0.200 (0.001)	0.164 (0.001)	0.052 (0.000)
exact025	126.583 (1.000)	64.282 (0.508)	0.428 (0.003)	0.342 (0.003)	50.461 (0.399)
exact030	130.447 (1.000)	1.626 (0.012)	0.436 (0.003)	0.450 (0.003)	3.217 (0.025)
exact035	149.698 (1.000)	27.191 (0.182)	1.231 (0.008)	1.216 (0.008)	3.841 (0.026)
exact040	203.146 (1.000)	33.006 (0.162)	1.845 (0.009)	2.137 (0.011)	7.562 (0.037)
exact045	315.867 (1.000)	100.685 (0.319)	5.117 (0.016)	3.532 (0.011)	55.475 (0.176)
exact050	58.436 (1.000)	116.069 (1.986)	4.431 (0.076)	5.024 (0.086)	21.190 (0.363)
exact055	216.111 (1.000)	44.549 (0.206)	5.022 (0.023)	25.516 (0.118)	6.679 (0.031)
exact060	231.181 (1.000)	129.309 (0.559)	11.878 (0.051)	12.126 (0.052)	192.187 (0.831)

Table A.8: Integral of time (seconds) vs normalized edits plot for SBPS configurations

	Default with SBPS	**Simple heuristic with SBPS**	**Kanpai with SBPS**	**Warm start with SBPS**	**Lookahead with SBPS**
exact003	0.340 (0.464)	4.734 (6.462)	1.656 (2.260)	0.675 (0.921)	0.309 (0.421)
exact005	82.026 (38.465)	3.099 (1.453)	0.065 (0.031)	1.466 (0.687)	1.511 (0.709)
instance_nr_11-csv-thres-0-40	31.282 (10.013)	0.022 (0.007)	0.021 (0.007)	0.020 (0.006)	0.605 (0.194)
instance_nr_313-csv-thres-0-45	4.042 (1.359)	1.228 (0.413)	0.011 (0.004)	0.015 (0.005)	2.259 (0.760)
instance_nr_1243-csv-thres-0-50	14.733 (1.794)	0.681 (0.083)	6.617 (0.806)	0.018 (0.002)	6.524 (0.794)
instance_nr_1679-csv-thres-0-45	220.617 (75.779)	0.021 (0.007)	0.021 (0.007)	0.024 (0.008)	0.004 (0.001)
pace_actionseq_21_10	0.712 (1.269)	0.017 (0.030)	0.030 (0.054)	0.019 (0.033)	1.639 (2.921)
pace_actionseq_22_2	1.059 (0.471)	0.026 (0.011)	0.014 (0.006)	0.016 (0.007)	9.245 (4.114)
pace_actionseq_23_10	7.824 (15.050)	1.457 (2.803)	0.115 (0.221)	0.249 (0.480)	3.716 (7.149)
pace_actionseq_26_2	763.118 (108.208)	0.022 (0.003)	0.023 (0.003)	0.025 (0.004)	0.006 (0.001)
exact015	210.920 (1.406)	32.269 (0.215)	0.128 (0.001)	0.097 (0.001)	38.291 (0.255)
exact020	188.976 (1.051)	101.102 (0.562)	0.198 (0.001)	0.176 (0.001)	0.052 (0.000)
exact025	117.793 (0.931)	64.291 (0.508)	0.431 (0.003)	0.341 (0.003)	50.926 (0.402)
exact030	119.676 (0.917)	1.629 (0.012)	0.423 (0.003)	0.448 (0.003)	3.072 (0.024)
exact035	148.243 (0.990)	27.268 (0.182)	1.226 (0.008)	1.208 (0.008)	3.845 (0.026)
exact040	197.011 (0.970)	33.075 (0.163)	1.849 (0.009)	2.133 (0.010)	6.545 (0.032)
exact045	410.662 (1.300)	100.838 (0.319)	5.029 (0.016)	3.575 (0.011)	53.543 (0.170)
exact050	53.112 (0.909)	116.077 (1.986)	4.461 (0.076)	5.050 (0.086)	21.044 (0.360)
exact055	211.103 (0.977)	44.556 (0.206)	5.044 (0.023)	25.521 (0.118)	6.754 (0.031)
exact060	318.718 (1.379)	129.359 (0.560)	11.751 (0.051)	12.045 (0.052)	191.820 (0.830)

A.5. Best objective value found after 10 minutes

Table A.9: Best objective value (edits) found after 10 minutes

	Default	Simple heuristic	Kanpai	Warm start	Lookahead
exact015	235 (0.362)	185 (0.107)	164 (0.000)	165 (0.005)	189 (0.128)
exact020	441 (0.554)	311 (0.337)	110 (0.000)	110 (0.000)	110 (0.000)
exact025	698 (0.422)	570 (0.213)	439 (0.000)	439 (0.000)	542 (0.168)
exact030	818 (0.435)	282 (0.004)	277 (0.000)	277 (0.000)	288 (0.009)
exact035	1106 (0.499)	510 (0.086)	385 (0.000)	385 (0.000)	402 (0.012)
exact040	1636 (0.677)	670 (0.105)	492 (0.000)	492 (0.000)	532 (0.024)
exact045	2028 (1.052)	1376 (0.325)	1085 (0.000)	1085 (0.000)	1248 (0.182)
exact050	1905 (0.193)	2350 (0.378)	1440 (0.000)	1440 (0.000)	1598 (0.066)
exact055	3075 (0.720)	1722 (0.135)	1410 (0.000)	1554 (0.062)	1449 (0.017)
exact060	2104 (0.769)	1818 (0.410)	1492 (0.001)	1497 (0.008)	1998 (0.636)

Table A.10: Best objective value (edits) found after 10 minutes for SBPS configurations (seconds)

	Default with SBPS	Simple heuristic with SBPS	Kanpai with SBPS	Warm start with SBPS	Lookahead with SBPS
exact015	295 (0.668)	185 (0.107)	164 (0.000)	164 (0.000)	189 (0.128)
exact020	483 (0.625)	311 (0.337)	110 (0.000)	110 (0.000)	110 (0.000)
exact025	680 (0.393)	570 (0.213)	439 (0.000)	439 (0.000)	543 (0.169)
exact030	773 (0.398)	282 (0.004)	277 (0.000)	277 (0.000)	288 (0.009)
exact035	1099 (0.494)	510 (0.086)	385 (0.000)	385 (0.000)	402 (0.012)
exact040	1600 (0.656)	670 (0.105)	492 (0.000)	492 (0.000)	526 (0.020)
exact045	2309 (1.366)	1376 (0.325)	1085 (0.000)	1085 (0.000)	1242 (0.175)
exact050	1862 (0.175)	2350 (0.378)	1440 (0.000)	1440 (0.000)	1597 (0.065)
exact055	3031 (0.701)	1722 (0.135)	1410 (0.000)	1554 (0.062)	1449 (0.017)
exact060	2293 (1.006)	1818 (0.410)	1492 (0.001)	1497 (0.008)	1997 (0.635)

B

MiniZinc Challenge Results

B.1. Solve time

Table B.1: Solve time (seconds)

	Default	**Default with SBPS**	**Lookahead**	**Lookahead with SBPS**
accap: accap_a12_f100_t60	600.000 (1.000)	295.851 (0.493)	600.000 (1.000)	600.000 (1.000)
accap: accap_a18_f140_t85	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
accap: accap_a40_f800_t180	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
accap: accap_a4_f30_t15	394.994 (1.000)	478.413 (1.211)	600.000 (1.519)	353.601 (0.895)
accap: accap_a5_f40_t20	128.791 (1.000)	43.950 (0.341)	51.359 (0.399)	600.000 (4.659)
arithmetic-target: 4108_with_1_2_2_3_3_5_6_6_7_8_9	71.368 (1.000)	87.098 (1.220)	75.708 (1.061)	301.354 (4.223)
arithmetic-target: 6872_with_1_2_3_3_4_4_5_6_7_9_10	99.084 (1.000)	81.340 (0.821)	228.816 (2.309)	249.995 (2.523)
arithmetic-target: 814_with_1_2_4_6_6_7_8_9	8.596 (1.000)	3.509 (0.408)	4.478 (0.521)	2.361 (0.275)
arithmetic-target: 8657_with_1_1_2_3_4_4_5_5_8_9_50	281.403 (1.000)	600.000 (2.132)	600.000 (2.132)	330.443 (1.174)
arithmetic-target: 910_with_1_2_3_6_12_25_50_87	4.632 (1.000)	7.486 (1.616)	2.670 (0.576)	2.931 (0.633)
blocks-world: 16-4-13	377.276 (1.000)	149.257 (0.396)	533.765 (1.415)	493.779 (1.309)
blocks-world: 16-4-40	600.000 (1.000)	207.266 (0.345)	567.108 (0.945)	600.000 (1.000)
blocks-world: 16-4-45	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)

Continued on next page

Table B.1: Solve time (seconds) (Continued)

blocks-world: 16-4-5	600.000 (1.000)	600.000 (1.000)	302.527 (0.504)	111.653 (0.186)
blocks-world: 16-4-83	600.000 (1.000)	273.580 (0.456)	600.000 (1.000)	244.675 (0.408)
diameterc-mst: c_v15_a105_d6	48.594 (1.000)	54.319 (1.118)	60.541 (1.246)	85.071 (1.751)
diameterc-mst: c_v15_a105_d9	148.092 (1.000)	308.145 (2.081)	280.956 (1.897)	232.927 (1.573)
diameterc-mst: c_v20_a190_d4	45.683 (1.000)	74.988 (1.641)	52.400 (1.147)	68.986 (1.510)
diameterc-mst: c_v20_a190_d9	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
diameterc-mst: s_v40_a100_d7	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
generalized-peacable-queens: n11_q5	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
generalized-peacable-queens: n13_q5	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
generalized-peacable-queens: n25_q4	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
generalized-peacable-queens: n8_q3	278.501 (1.000)	284.028 (1.020)	274.609 (0.986)	360.074 (1.293)
generalized-peacable-queens: n9_q5	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
gfd-schedule: n180f7d50m30k18_10124	600.000 (1.000)	600.000 (1.000)	11.655 (0.019)	6.037 (0.010)
gfd-schedule: n200f5d50m40k6_10124	600.000 (1.000)	600.000 (1.000)	30.274 (0.050)	19.132 (0.032)
gfd-schedule: n55f2d50m30k3_10124	3.464 (1.000)	0.985 (0.284)	0.219 (0.063)	0.147 (0.042)
gfd-schedule: n60f7d50m30k10_10124	2.752 (1.000)	1.996 (0.725)	0.258 (0.094)	0.171 (0.062)
gfd-schedule: n85f3d50m8k20_10124	600.000 (1.000)	600.000 (1.000)	1.074 (0.002)	0.565 (0.001)
ma-path-finding: ins_g16_p10_a20	113.941 (1.000)	73.111 (0.642)	92.321 (0.810)	56.128 (0.493)
ma-path-finding: ins_g16_p10_a30	406.227 (1.000)	239.732 (0.590)	600.000 (1.477)	255.317 (0.629)
ma-path-finding: ins_g16_p20_a30	600.000 (1.000)	503.061 (0.838)	600.000 (1.000)	600.000 (1.000)
ma-path-finding: ins_g24_p20_a10	112.699 (1.000)	79.286 (0.704)	182.731 (1.621)	131.139 (1.164)
ma-path-finding: ins_g32_p20_a10	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)

Continued on next page

Table B.1: Solve time (seconds) (Continued)

nfc: 12_2_11	600.000 (1.000)	600.000 (1.000)	6.554 (0.011)	6.609 (0.011)
nfc: 18_3_12	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
nfc: 24_4_2	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
nfc: 30_5_12	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
nfc: 30_5_6	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
roster-sickness: large-2-2	1.062 (1.000)	0.987 (0.929)	2.390 (2.250)	1.774 (1.670)
roster-sickness: large-2	203.561 (1.000)	286.518 (1.408)	280.283 (1.377)	490.507 (2.410)
roster-sickness: large-4-2	0.006 (1.000)	0.007 (1.167)	0.008 (1.333)	0.006 (1.000)
roster-sickness: large-4	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
roster-sickness: small-4	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
spot5: 1405	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
spot5: 1506	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
spot5: 404	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
spot5: 507	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
spot5: 509	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
stripboard: common-emitter-complex	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
stripboard: common-emitter-simple	134.368 (1.000)	183.108 (1.363)	134.691 (1.002)	54.142 (0.403)
stripboard: envelope-detector	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
stripboard: nand-gate	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
stripboard: opamp-integrator	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
sudoku_opt: sudoku_p20	1.515 (1.000)	2.460 (1.624)	2.043 (1.349)	2.092 (1.381)
sudoku_opt: sudoku_p22	84.988 (1.000)	224.850 (2.646)	130.078 (1.531)	176.187 (2.073)

Continued on next page

Table B.1: Solve time (seconds) (Continued)

sudoku_opt: sudoku_p23	6.512 (1.000)	9.741 (1.496)	13.890 (2.133)	20.933 (3.215)
sudoku_opt: sudoku_p29	8.884 (1.000)	16.810 (1.892)	20.407 (2.297)	21.031 (2.367)
sudoku_opt: sudoku_p90	0.129 (1.000)	0.163 (1.264)	0.240 (1.860)	0.320 (2.481)
team-assignment: data1_4_6	0.043 (1.000)	0.055 (1.279)	0.074 (1.721)	0.040 (0.930)
team-assignment: data1_6_6	0.180 (1.000)	0.465 (2.583)	0.268 (1.489)	0.874 (4.856)
team-assignment: data2_6_15	153.245 (1.000)	91.390 (0.596)	209.457 (1.367)	254.380 (1.660)
team-assignment: data3_4_31	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
team-assignment: data3_5_31	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
tower: 100_100_20_100-04	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
tower: tower_070_070_15_070-09	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
tower: tower_070_070_15_085-09	0.380 (1.000)	0.367 (0.966)	37.800 (99.474)	0.801 (2.108)
tower: tower_300_300_40_200-00	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
tower: tower_500_500_50_300-01	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
traveling-tppv: circ14cnonbal	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
traveling-tppv: circ14enonbal	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
traveling-tppv: circ20bnonbal	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
traveling-tppv: circ20fnonbal	0.607 (1.000)	0.597 (0.984)	34.472 (56.791)	38.202 (62.936)
traveling-tppv: circ20jnonbal	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
triangular: n10	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
triangular: n18	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
triangular: n24	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
triangular: n30	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)

Continued on next page

Table B.1: Solve time (seconds) (Continued)

triangular: n39	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
vaccine: v11	463.979 (1.000)	104.975 (0.226)	430.615 (0.928)	240.857 (0.519)
vaccine: v7	525.355 (1.000)	101.740 (0.194)	600.000 (1.142)	600.000 (1.142)
vaccine: v8	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
vaccine: v857	13.008 (1.000)	2.808 (0.216)	13.458 (1.035)	6.806 (0.523)
vaccine: v946	55.881 (1.000)	53.083 (0.950)	77.236 (1.382)	600.000 (10.737)
wordpress: Wordpress10_Offers500	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
wordpress: Wordpress11_Offers500	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
wordpress: Wordpress12_Offers500	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
wordpress: Wordpress7_Offers500	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
wordpress: Wordpress8_Offers500	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
yumi-static: example_instance_4_GS_SG_yumi_...	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
yumi-static: p_10_GGGGG_GGGGG_yumi_grid_...	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
yumi-static: p_4_GG_GG_yumi_grid_setup_3_3	100.618 (1.000)	181.705 (1.806)	155.254 (1.543)	274.369 (2.727)
yumi-static: p_4_GS_SG_yumi_grid_setup_3_3	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)
yumi-static: p_8_SSSSS_SSS_yumi_grid_setup_...	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)	600.000 (1.000)

B.2. Objective

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found)

	Default	**Default with SBPS**	**Lookahead**	**Lookahead with SBPS**
accap: accap_a12_f100_t60	136.000 (0.002)	127.000 (0.000)	190.000 (0.016)	169.000 (0.011)
accap: accap_a18_f140_t85	157.000 (0.000)	154.000 (0.000)	274.000 (0.016)	264.000 (0.015)
accap: accap_a40_f800_t180	5446.000 (0.937)	4388.000 (0.597)	3005.000 (0.152)	2534.000 (0.000)

Continued on next page

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found) (Continued)

accap: **accap_a4_f30_t15**	103.000 (0.000)	103.000 (0.000)	103.000 (0.000)	103.000 (0.000)
accap: **accap_a5_f40_t20**	97.000 (0.000)	97.000 (0.000)	97.000 (0.000)	97.000 (0.000)
arithmetic-target: **4108_with_1_2_2_3_3_5_6_6_7_8_9**	7.000 (0.000)	7.000 (0.000)	7.000 (0.000)	7.000 (0.000)
arithmetic-target: **6872_with_1_2_3_3_4_4_5_6_7_9_10**	7.000 (0.000)	7.000 (0.000)	7.000 (0.000)	7.000 (0.000)
arithmetic-target: **814_with_1_2_4_6_6_7_8_9**	5.000 (0.000)	5.000 (0.000)	5.000 (0.000)	5.000 (0.000)
arithmetic-target: **8657_with_1_1_2_3_4_4_5_5_8_9_50**	7.000 (0.000)	10.000 (0.000)	17.000 (0.000)	7.000 (0.000)
arithmetic-target: **910_with_1_2_3_6_12_25_50_87**	5.000 (0.000)	5.000 (0.000)	5.000 (0.000)	5.000 (0.000)
blocks-world: **16-4-13**	30.000 (0.000)	30.000 (0.000)	30.000 (0.000)	30.000 (0.000)
blocks-world: **16-4-40**	33.000 (0.348)	25.000 (0.000)	25.000 (0.000)	30.000 (0.217)
blocks-world: **16-4-45**	43.000 (0.432)	27.000 (0.000)	44.000 (0.459)	28.000 (0.027)
blocks-world: **16-4-5**	26.000 (0.036)	27.000 (0.071)	25.000 (0.000)	25.000 (0.000)
blocks-world: **16-4-83**	30.000 (0.172)	25.000 (0.000)	31.000 (0.207)	25.000 (0.000)
diameterc-mst: **c_v15_a105_d6**	314.000 (0.000)	314.000 (0.000)	314.000 (0.000)	314.000 (0.000)
diameterc-mst: **c_v15_a105_d9**	290.000 (0.000)	290.000 (0.000)	290.000 (0.000)	290.000 (0.000)
diameterc-mst: **c_v20_a190_d4**	349.000 (0.000)	349.000 (0.000)	349.000 (0.000)	349.000 (0.000)
diameterc-mst: **c_v20_a190_d9**	332.000 (0.018)	327.000 (0.000)	335.000 (0.029)	341.000 (0.051)
diameterc-mst: **s_v40_a100_d7**	N/A	N/A	821.000 (0.240)	773.000 (0.000)
generalized-peacable-queens: **n11_q5**	4.000 (0.000)	4.000 (0.000)	N/A	N/A
generalized-peacable-queens: **n13_q5**	N/A	N/A	N/A	N/A
generalized-peacable-queens: **n25_q4**	N/A	N/A	N/A	N/A
generalized-peacable-queens: **n8_q3**	4.000 (0.000)	4.000 (0.000)	4.000 (0.000)	4.000 (0.000)
generalized-peacable-queens: **n9_q5**	N/A	N/A	2.000 (0.000)	2.000 (0.000)

Continued on next page

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found) (Continued)

gfd-schedule: n180f7d50m30k18_10124	N/A	N/A	45.000 (0.000)	45.000 (0.000)
gfd-schedule: n200f5d50m40k6_10124	N/A	N/A	18.000 (0.000)	18.000 (0.000)
gfd-schedule: n55f2d50m30k3_10124	9.000 (0.000)	9.000 (0.000)	9.000 (0.000)	9.000 (0.000)
gfd-schedule: n60f7d50m30k10_10124	20.000 (0.000)	20.000 (0.000)	20.000 (0.000)	20.000 (0.000)
gfd-schedule: n85f3d50m8k20_10124	N/A	N/A	138.000 (0.000)	138.000 (0.000)
ma-path-finding: ins_g16_p10_a20	240.000 (0.000)	240.000 (0.000)	240.000 (0.000)	240.000 (0.000)
ma-path-finding: ins_g16_p10_a30	334.000 (0.000)	334.000 (0.000)	356.000 (0.062)	334.000 (0.000)
ma-path-finding: ins_g16_p20_a30	723.000 (0.722)	401.000 (0.000)	609.000 (0.466)	413.000 (0.027)
ma-path-finding: ins_g24_p20_a10	193.000 (0.000)	193.000 (0.000)	193.000 (0.000)	193.000 (0.000)
ma-path-finding: ins_g32_p20_a10	409.000 (0.607)	253.000 (0.000)	319.000 (0.257)	275.000 (0.086)
nfc: 12_2_11	1132.000 (0.635)	1032.000 (0.453)	784.000 (0.000)	784.000 (0.000)
nfc: 18_3_12	1911.000 (0.931)	1662.000 (0.597)	1254.000 (0.048)	1218.000 (0.000)
nfc: 24_4_2	2828.000 (0.950)	2824.000 (0.946)	1856.000 (0.008)	1848.000 (0.000)
nfc: 30_5_12	3540.000 (0.976)	3310.000 (0.790)	2330.000 (0.000)	2340.000 (0.008)
nfc: 30_5_6	3030.000 (0.793)	2915.000 (0.000)	N/A	N/A
roster-sickness: large-2-2	191062.000 (0.000)	191062.000 (0.000)	191062.000 (0.000)	191062.000 (0.000)
roster-sickness: large-2	247896.000 (0.000)	247896.000 (0.000)	247896.000 (0.000)	247896.000 (0.000)
roster-sickness: large-4-2	233969.000 (0.000)	233969.000 (0.000)	233969.000 (0.000)	233969.000 (0.000)
roster-sickness: large-4	217452.000 (0.006)	217464.000 (0.000)	217450.000 (0.007)	217438.000 (0.014)
roster-sickness: small-4	14457.000 (0.000)	14457.000 (0.000)	14457.000 (0.000)	14457.000 (0.000)
spot5: 1405	551536.000 (0.270)	572547.000 (0.452)	520458.000 (0.000)	527464.000 (0.061)
spot5: 1506	451707.000 (0.393)	463707.000 (0.495)	405669.000 (0.000)	420577.000 (0.127)

Continued on next page

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found) (Continued)

spot5: 404	116.000 (0.041)	120.000 (0.122)	116.000 (0.041)	114.000 (0.000)
spot5: 507	31425.000 (0.000)	32429.000 (0.090)	33439.000 (0.181)	32421.000 (0.090)
spot5: 509	38544.000 (0.000)	43473.000 (0.289)	44478.000 (0.349)	43492.000 (0.291)
stripboard: common-emitter-complex	190.000 (0.833)	140.000 (0.000)	N/A	N/A
stripboard: common-emitter-simple	40.000 (0.000)	40.000 (0.000)	40.000 (0.000)	40.000 (0.000)
stripboard: envelope-detector	110.000 (0.268)	80.000 (0.024)	130.000 (0.431)	77.000 (0.000)
stripboard: nand-gate	90.000 (0.236)	56.000 (0.000)	N/A	N/A
stripboard: opamp-integrator	N/A	N/A	N/A	N/A
sudoku_opt: sudoku_p20	-3.000 (0.000)	-3.000 (0.000)	-3.000 (0.000)	-3.000 (0.000)
sudoku_opt: sudoku_p22	229.000 (0.000)	229.000 (0.000)	229.000 (0.000)	229.000 (0.000)
sudoku_opt: sudoku_p23	-227.000 (0.000)	-227.000 (0.000)	-227.000 (0.000)	-227.000 (0.000)
sudoku_opt: sudoku_p29	-359.000 (0.000)	-359.000 (0.000)	-359.000 (0.000)	-359.000 (0.000)
sudoku_opt: sudoku_p90	-55.000 (0.000)	-55.000 (0.000)	-55.000 (0.000)	-55.000 (0.000)
team-assignment: data1_4_6	2948.000 (0.000)	2948.000 (0.000)	2948.000 (0.000)	2948.000 (0.000)
team-assignment: data1_6_6	6970.000 (0.000)	6970.000 (0.000)	6970.000 (0.000)	6970.000 (0.000)
team-assignment: data2_6_15	26773.000 (0.000)	26773.000 (0.000)	26773.000 (0.000)	26773.000 (0.000)
team-assignment: data3_4_31	24622.000 (0.460)	27664.000 (0.378)	31584.000 (0.272)	41683.000 (0.000)
team-assignment: data3_5_31	N/A	N/A	37564.000 (0.105)	37674.000 (0.000)
tower: 100_100_20_100-04	88.000 (0.042)	50.000 (0.833)	90.000 (0.000)	86.000 (0.083)
tower: tower_070_070_15_070-09	57.000 (0.000)	48.000 (0.167)	57.000 (0.000)	57.000 (0.000)
tower: tower_070_070_15_085-09	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)
tower: tower_300_300_40_200-00	149.000 (0.122)	104.000 (0.513)	157.000 (0.052)	163.000 (0.000)

Continued on next page

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found) (Continued)

tower: tower_500_500_50_300-01	167.000 (0.000)	111.000 (0.709)	167.000 (0.000)	153.000 (0.177)
traveling-tppv: circ14cnonbal	504.000 (0.288)	474.000 (0.000)	480.000 (0.058)	474.000 (0.000)
traveling-tppv: circ14enonbal	476.000 (0.367)	432.000 (0.000)	476.000 (0.367)	464.000 (0.267)
traveling-tppv: circ20bnonbal	1498.000 (0.377)	1468.000 (0.262)	1482.000 (0.315)	1400.000 (0.000)
traveling-tppv: circ20fnonbal	N/A	N/A	N/A	N/A
traveling-tppv: circ20jnonbal	1498.000 (0.611)	1428.000 (0.222)	1462.000 (0.411)	1388.000 (0.000)
triangular: n10	20.000 (0.000)	20.000 (0.000)	20.000 (0.000)	20.000 (0.000)
triangular: n18	38.000 (0.000)	35.000 (0.150)	38.000 (0.000)	34.000 (0.200)
triangular: n24	52.000 (0.034)	43.000 (0.345)	53.000 (0.000)	41.000 (0.414)
triangular: n30	67.000 (0.000)	58.000 (0.346)	67.000 (0.000)	53.000 (0.538)
triangular: n39	89.000 (0.000)	61.000 (0.651)	68.000 (0.488)	66.000 (0.535)
vaccine: v11	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)
vaccine: v7	45.000 (0.000)	45.000 (0.000)	43.000 (0.105)	45.000 (0.000)
vaccine: v8	N/A	N/A	N/A	N/A
vaccine: v857	127.000 (0.000)	127.000 (0.000)	127.000 (0.000)	127.000 (0.000)
vaccine: v946	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)	85.000 (0.000)
wordpress: Wordpress10_Offers500	42941.000 (0.210)	3232.000 (0.000)	3466.000 (0.001)	3350.000 (0.001)
wordpress: Wordpress11_Offers500	33136.000 (0.124)	57195.000 (0.225)	3531.000 (0.000)	3464.000 (0.000)
wordpress: Wordpress12_Offers500	67739.000 (0.312)	76868.000 (0.357)	3920.000 (0.000)	4004.000 (0.000)
wordpress: Wordpress7_Offers500	2022.000 (0.000)	2116.000 (0.001)	2126.000 (0.001)	2022.000 (0.000)
wordpress: Wordpress8_Offers500	6230.000 (0.027)	4124.000 (0.011)	2924.000 (0.002)	2674.000 (0.000)
yumi-static: example_instance_4_GS_SG_yumi_...	482.000 (0.000)	482.000 (0.000)	509.000 (0.026)	482.000 (0.000)

Continued on next page

Table B.2: Best objective found after 10 minutes (N/A indicates no solution found) (Continued)

yumi-static: p_10_GGGGG_GGGGG_yumi_grid_...	N/A	N/A	N/A	N/A
yumi-static: p_4_GG_GG_yumi_grid_setup_3_3	626.000 (0.000)	626.000 (0.000)	626.000 (0.000)	626.000 (0.000)
yumi-static: p_4_GS_SG_yumi_grid_setup_3_3	595.000 (0.060)	617.000 (0.094)	589.000 (0.050)	557.000 (0.000)
yumi-static: p_8_SSSSS_SSS_yumi_grid_setup_...	N/A	N/A	N/A	N/A

B.3. Integral

Table B.3: Integral of time (seconds) vs normalized edits plot

	Default	**Default with SBPS**	**Lookahead**	**Lookahead with SBPS**
accap: accap_a12_f100_t60	8.191 (1.000)	8.177 (0.998)	253.807 (30.988)	255.281 (31.168)
accap: accap_a18_f140_t85	15.923 (1.000)	25.497 (1.601)	9.650 (0.606)	8.934 (0.561)
accap: accap_a40_f800_t180	562.472 (1.000)	466.883 (0.830)	111.080 (0.197)	22.020 (0.039)
accap: accap_a4_f30_t15	2.553 (1.000)	0.632 (0.248)	2.502 (0.980)	2.332 (0.914)
accap: accap_a5_f40_t20	2.551 (1.000)	0.351 (0.137)	0.249 (0.098)	8.410 (3.297)
arithmetic-target: 4108_with_1_2_2_3_3_5_6_6_7_8_9	26.194 (1.000)	1.625 (0.062)	17.227 (0.658)	30.494 (1.164)
arithmetic-target: 6872_with_1_2_3_3_4_4_5_6_7_9_10	6.185 (1.000)	23.580 (3.812)	61.412 (9.929)	24.350 (3.937)
arithmetic-target: 814_with_1_2_4_6_6_7_8_9	4.132 (1.000)	3.715 (0.899)	0.577 (0.140)	0.563 (0.136)
arithmetic-target: 8657_with_1_1_2_3_4_4_5_5_8_9_50	40.005 (1.000)	3.187 (0.080)	138.337 (3.458)	35.798 (0.895)
arithmetic-target: 910_with_1_2_3_6_12_25_50_87	6.928 (1.000)	6.923 (0.999)	0.322 (0.046)	0.322 (0.046)
blocks-world: 16-4-13	73.426 (1.000)	42.808 (0.583)	716.147 (9.753)	725.440 (9.880)
blocks-world: 16-4-40	442.735 (1.000)	47.552 (0.107)	129.177 (0.292)	145.595 (0.329)
blocks-world: 16-4-45	510.588 (1.000)	315.119 (0.617)	386.154 (0.756)	219.309 (0.430)
blocks-world: 16-4-5	544.305 (1.000)	509.435 (0.936)	161.421 (0.297)	47.473 (0.087)
blocks-world: 16-4-83	190.179 (1.000)	363.861 (1.913)	175.088 (0.921)	188.036 (0.989)
diameterc-mst: c_v15_a105_d6	1.955 (1.000)	0.058 (0.030)	2.749 (1.406)	3.489 (1.784)
diameterc-mst: c_v15_a105_d9	0.499 (1.000)	0.389 (0.780)	14.279 (28.618)	7.041 (14.112)
diameterc-mst: c_v20_a190_d4	0.170 (1.000)	0.249 (1.461)	1.762 (10.355)	0.748 (4.397)
diameterc-mst: c_v20_a190_d9	16.865 (1.000)	8.453 (0.501)	46.610 (2.764)	57.153 (3.389)
diameterc-mst: s_v40_a100_d7	1200.000 (1.000)	1200.000 (1.000)	390.787 (0.326)	335.481 (0.280)

Continued on next page

Table B.3: Integral of time (seconds) vs normalized edits plot (Continued)

generalized-peacable-queens: n11_q5	12.928 (1.000)	12.694 (0.982)	302.655 (23.411)	302.359 (23.388)
generalized-peacable-queens: n13_q5	202.639 (1.000)	0.471 (0.002)	203.625 (1.005)	1200.000 (5.922)
generalized-peacable-queens: n25_q4	N/A	N/A	N/A	N/A
generalized-peacable-queens: n8_q3	0.011 (1.000)	0.011 (1.000)	0.005 (0.455)	0.005 (0.455)
generalized-peacable-queens: n9_q5	600.000 (1.000)	600.000 (1.000)	51.507 (0.086)	51.462 (0.086)
gfd-schedule: n180f7d50m30k18_10124	1200.000 (1.000)	1200.000 (1.000)	5.963 (0.005)	2.534 (0.002)
gfd-schedule: n200f5d50m40k6_10124	1200.000 (1.000)	1200.000 (1.000)	9.570 (0.008)	1.912 (0.002)
gfd-schedule: n55f2d50m30k3_10124	1.460 (1.000)	0.641 (0.439)	0.031 (0.021)	0.016 (0.011)
gfd-schedule: n60f7d50m30k10_10124	1.488 (1.000)	1.403 (0.943)	0.031 (0.021)	0.025 (0.017)
gfd-schedule: n85f3d50m8k20_10124	572.406 (1.000)	216.460 (0.378)	0.367 (0.001)	0.179 (0.000)
ma-path-finding: ins_g16_p10_a20	87.450 (1.000)	60.658 (0.694)	45.611 (0.522)	37.083 (0.424)
ma-path-finding: ins_g16_p10_a30	298.886 (1.000)	158.425 (0.530)	404.728 (1.354)	300.305 (1.005)
ma-path-finding: ins_g16_p20_a30	616.863 (1.000)	330.592 (0.536)	748.052 (1.213)	639.589 (1.037)
ma-path-finding: ins_g24_p20_a10	81.999 (1.000)	64.134 (0.782)	80.204 (0.978)	64.695 (0.789)
ma-path-finding: ins_g32_p20_a10	626.702 (1.000)	460.196 (0.734)	512.796 (0.818)	487.726 (0.778)
nfc: 12_2_11	381.041 (1.000)	271.814 (0.713)	0.013 (0.000)	0.022 (0.000)
nfc: 18_3_12	561.703 (1.000)	358.858 (0.639)	29.057 (0.052)	1.881 (0.003)
nfc: 24_4_2	575.903 (1.000)	567.456 (0.985)	4.775 (0.008)	5.425 (0.009)
nfc: 30_5_12	585.626 (1.000)	474.315 (0.810)	0.018 (0.000)	4.883 (0.008)
nfc: 30_5_6	479.826 (1.000)	0.037 (0.000)	1200.000 (2.501)	1200.000 (2.501)
roster-sickness: large-2-2	0.018 (1.000)	0.020 (1.106)	0.079 (4.266)	0.070 (3.798)
roster-sickness: large-2	0.066 (1.000)	0.088 (1.345)	0.180 (2.743)	0.191 (2.913)

Continued on next page

Table B.3: Integral of time (seconds) vs normalized edits plot (Continued)

roster-sickness: large-4-2	0.006 (1.000)	0.006 (1.000)	0.008 (1.333)	0.005 (0.833)
roster-sickness: large-4	4.143 (1.000)	1.210 (0.292)	6.705 (1.618)	8.407 (2.029)
roster-sickness: small-4	1.880 (1.000)	0.019 (0.010)	0.089 (0.047)	0.373 (0.198)
spot5: 1405	290.830 (1.000)	292.696 (1.006)	0.848 (0.003)	37.273 (0.128)
spot5: 1506	332.950 (1.000)	323.948 (0.973)	5.960 (0.018)	77.441 (0.233)
spot5: 404	43.946 (1.000)	78.585 (1.788)	33.159 (0.755)	10.170 (0.231)
spot5: 507	10.896 (1.000)	58.465 (5.366)	108.918 (9.996)	53.913 (4.948)
spot5: 509	126.504 (1.000)	197.310 (1.560)	209.241 (1.654)	175.631 (1.388)
stripboard: common-emitter-complex	974.842 (1.000)	786.472 (0.807)	1200.000 (1.231)	1200.000 (1.231)
stripboard: common-emitter-simple	13.263 (1.000)	23.561 (1.777)	25.614 (1.931)	11.870 (0.895)
stripboard: envelope-detector	328.798 (1.000)	91.218 (0.277)	416.025 (1.265)	21.778 (0.066)
stripboard: nand-gate	291.326 (1.000)	72.595 (0.249)	1200.000 (4.119)	275.438 (0.945)
stripboard: opamp-integrator	N/A	N/A	N/A	N/A
sudoku_opt: sudoku_p20	0.706 (1.000)	0.704 (0.997)	2.020 (2.861)	2.063 (2.922)
sudoku_opt: sudoku_p22	16.608 (1.000)	16.722 (1.007)	80.362 (4.839)	80.824 (4.867)
sudoku_opt: sudoku_p23	4.159 (1.000)	4.226 (1.016)	9.355 (2.249)	9.531 (2.292)
sudoku_opt: sudoku_p29	3.220 (1.000)	3.302 (1.025)	18.846 (5.853)	18.800 (5.839)
sudoku_opt: sudoku_p90	0.105 (1.000)	0.108 (1.029)	0.095 (0.905)	0.096 (0.914)
team-assignment: data1_4_6	0.020 (1.000)	0.022 (1.090)	0.007 (0.360)	0.005 (0.234)
team-assignment: data1_6_6	0.037 (1.000)	0.075 (2.022)	0.016 (0.441)	0.155 (4.141)
team-assignment: data2_6_15	183.286 (1.000)	139.957 (0.764)	137.957 (0.753)	41.305 (0.225)
team-assignment: data3_4_31	474.804 (1.000)	393.635 (0.829)	164.114 (0.346)	1.236 (0.003)

Continued on next page

Table B.3: Integral of time (seconds) vs normalized edits plot (Continued)

team-assignment: data3_5_31	1200.000 (1.000)	1200.000 (1.000)	272.079 (0.227)	19.903 (0.017)
tower: 100_100_20_100-04	244.493 (1.000)	544.424 (2.227)	17.836 (0.073)	132.934 (0.544)
tower: tower_070_070_15_070-09	79.882 (1.000)	147.488 (1.846)	144.905 (1.814)	46.465 (0.582)
tower: tower_070_070_15_085-09	0.752 (1.000)	0.726 (0.965)	14.954 (19.886)	0.978 (1.300)
tower: tower_300_300_40_200-00	279.066 (1.000)	480.956 (1.723)	58.470 (0.210)	63.744 (0.228)
tower: tower_500_500_50_300-01	126.538 (1.000)	431.918 (3.413)	21.761 (0.172)	133.136 (1.052)
traveling-tppv: circ14cnonbal	211.092 (1.000)	10.188 (0.048)	37.183 (0.176)	51.128 (0.242)
traveling-tppv: circ14enonbal	243.462 (1.000)	96.448 (0.396)	231.493 (0.951)	189.891 (0.780)
traveling-tppv: circ20bnonbal	267.195 (1.000)	286.708 (1.073)	204.821 (0.767)	122.152 (0.457)
traveling-tppv: circ20fnonbal	N/A	N/A	N/A	N/A
traveling-tppv: circ20jnonbal	374.015 (1.000)	294.252 (0.787)	259.466 (0.694)	135.376 (0.362)
triangular: n10	5.867 (1.000)	17.221 (2.935)	6.520 (1.111)	9.125 (1.555)
triangular: n18	0.313 (1.000)	94.693 (302.968)	31.693 (101.401)	125.596 (401.843)
triangular: n24	22.691 (1.000)	206.980 (9.122)	46.294 (2.040)	273.897 (12.071)
triangular: n30	27.308 (1.000)	207.750 (7.608)	212.236 (7.772)	357.335 (13.085)
triangular: n39	1.895 (1.000)	391.132 (206.428)	299.641 (158.141)	332.393 (175.427)
vaccine: v11	254.590 (1.000)	24.998 (0.098)	581.574 (2.284)	383.854 (1.508)
vaccine: v7	14.513 (1.000)	1.189 (0.082)	73.793 (5.085)	41.228 (2.841)
vaccine: v8	N/A	N/A	N/A	N/A
vaccine: v857	6.864 (1.000)	0.936 (0.136)	7.192 (1.048)	5.033 (0.733)
vaccine: v946	2.179 (1.000)	1.644 (0.754)	4.488 (2.059)	4.580 (2.102)
wordpress: Wordpress10_Offers500	591.408 (1.000)	466.001 (0.788)	0.899 (0.002)	0.535 (0.001)

Continued on next page

Table B.3: Integral of time (seconds) vs normalized edits plot (Continued)

wordpress: Wordpress11_Offers500	1030.613 (1.000)	1090.051 (1.058)	0.403 (0.000)	0.260 (0.000)
wordpress: Wordpress12_Offers500	1159.268 (1.000)	1154.145 (0.996)	0.261 (0.000)	0.880 (0.001)
wordpress: Wordpress7_Offers500	21.966 (1.000)	2.869 (0.131)	0.475 (0.022)	0.015 (0.001)
wordpress: Wordpress8_Offers500	184.791 (1.000)	113.892 (0.616)	1.154 (0.006)	0.297 (0.002)
yumi-static: example_instance_4_GS_SG_yumi_...	97.672 (1.000)	51.436 (0.527)	21.359 (0.219)	17.066 (0.175)
yumi-static: p_10_GGGGG_GGGGG_yumi_grid_...	N/A	N/A	N/A	N/A
yumi-static: p_4_GG_GG_yumi_grid_setup_3_3	11.931 (1.000)	10.971 (0.920)	7.826 (0.656)	23.230 (1.947)
yumi-static: p_4_GS_SG_yumi_grid_setup_3_3	74.613 (1.000)	70.393 (0.943)	79.533 (1.066)	57.356 (0.769)
yumi-static: p_8_SSSSS_SSS_yumi_grid_setup_...	N/A	N/A	N/A	N/A

C

Scientific paper

This appendix includes a draft scientific paper about the lookahead approach for constraint optimization problems.

Lookahead Chuffed value selection for constraint optimization problems

Angelos Zoumis

TU Delft, The Netherlands

Abstract. Constraint programming solvers provide a generalizable approach to finding solutions for optimization problems. As constraint programming is used for many real-world applications, such as bioinformatics, finance, telecommunications, engineering, and transportation [12, 11], improving the performance of constraint programming solvers on optimization problems is of great importance. A new "lookahead" approach for value selection is developed that attempts to assist the solver in finding near-optimal solutions faster. This approach works by temporarily performing additional propagations before selecting a value and then selecting the value that includes a more optimal solution within its domain after propagation. When combining this approach with Chuffed [2], this approach added a significant time overhead that increases the overall solving time for many problems, with the lookahead configuration having a median solve time of 8.67% for the optimization problems of the MiniZinc Challenge 2022 [8]. However, the lookahead configuration was able to find near-optimal solutions significantly faster than the default value selection. In particular, for the same optimization problems, on average, the lookahead configuration had a definite integral for the time vs objective graph 54.70% lower than the default Chuffed configuration.

Keywords: Constraint Programming · Optimization problem · Chuffed · Lookahead · Value Selection.

1 Introduction

Constraint programming (CP) is a powerful approach for solving a wide range of problems involving constraints. It deals with the modelling of problems with variables that are subject to constraints, and solving them by providing a set of assignments to the variables, such that all constraints are satisfied. Furthermore, for optimization problems, the variables also have to maximize an objective function [10]. One of the main advantages of constraint programming is that it only needs a model of the problem to work, instead of a tailor-made algorithm, making them extremely versatile and generalizable.

Constraint programming has been applied to a wide range of real-world problems, including scheduling, planning, resource allocation and configuration, and design problems, for many different fields, such as bioinformatics, finance, telecommunications, engineering, and transportation [12, 11]. For example, in

scheduling, constraint programming can be used to find optimal, or near-optimal schedules that take into account a wide range of constraints, such as resource and personnel availability, and temporal constraints, while having an objective function, like minimizing costs. Therefore, solving, but also finding near-optimal answers for CP problems as quickly as possible is of great interest to many companies that rely on CP solvers.

Despite many improvements, CP solvers still struggle to find optimal, or near-optimal solutions at a reasonable time for certain problems in comparison to other approaches to finding near-optimal solutions, such as heuristic solvers. Therefore, the goal of this paper is to present a new value selection approach for constraint optimization solvers that allow the discovery of near-optimal solutions faster.

This research paper describes a lookahead approach for value selection that attempt to prioritize reaching solutions with a more optimal optimization value first.

The paper will be organized as follows. First, section 2 describes the background work of constraint programming. Next, the lookahead approach that is developed will be presented in section 3. Continuing, section 4 details the experimental setup and section 5 shows and analyzes the results. Section 6 discusses other related work, detailing different state-of-the-art techniques, focused mainly on approaches affecting the branching of Chuffed. Section 7 discusses various different possible future improvements to the lookahead approach, and finally, section 8 presents the conclusion of the paper.

2 Background

This section details the background work on which the lookahead approach is built. In particular, constraint programming will be detailed.

The main concept of Constraint programming is modelling a problem and solving said problem through the use of CP solvers. Therefore the field of constraint programming is split into two sub-fields. One focused on the modelling of the problems and the languages designed to implement said models. The second field focuses on the algorithms used to solve or optimize said models [10].

2.1 Modeling

A constraint programming model describes the rules and restrictions of a problem. A constraint optimization model consists of 4 parts:

- **Variables:** The main part of a CP model is the set of variables that should be assigned a value.
- **Domain:** The domain of possible values that the variable can be assigned. Any assignment of values within the domain of each variable is a candidate solution.
- **Constraints:** Each constraint is a condition that needs to be satisfied.

– **Objective function:** Optionally, a model can have an objective function that needs to be maximized or minimized based on the problem. The goal of the research is to improve the performance of CP solvers in optimization problems. Therefore, each used model will always have an objective function.

2.2 CP solvers

Constraint programming solvers take as an input a model and attempt to find a solution, where each variable has an assigned value that is within the domain of each variable while satisfying all the constraints and maximizing/minimizing the optimization function. In general, algorithms that attempt to solve such problems are split into two cooperating strategies, propagation or inference, and branching or searching [10].

Propagation is a vital step in CP solvers, reducing the search space of candidate solutions, and enabling solvers to work more efficiently, especially as the problem size increases. By default, using backtracking to solve constraint satisfaction problems almost always leads to thrashing behaviours [10, 1]. Thrashing is the repeated exploration of sets of subtree modules that are failing due to the same assignments and only differ in assignments that are not related to the cause of failure. As the size of the problem increases, and hence, the size of the search space increases exponentially, thrashing becomes the main contributor to running time in backtracking. Propagation can significantly reduce thrashing, by making implicit constraints explicit, and removing values from the domains of variables that are not consistent, meaning that these values produce a non-satisfiable solution.

Searching is the task of navigating through the search tree of a problem, in order to find a solution satisfying all constraints. Backtracking is the fundamental search method for CP problems, as it guarantees to find a satisfactory solution if one exists [10]. Backtracking builds up a partial solution, by selecting values for variables until reaching a solution, or a conflict, at which point, it backtracks to a previous decision where a different choice can be made, and makes a different decision. This approach potentially visits all feasible partial solutions and hence guarantees to find any existing satisfying solutions. This approach is better than brute force, as it checks after each decision if the constraints are met, instead of doing that only until a full solution is found. Hence, discovering that a partial solution cannot satisfy all constraints prunes the subtree of that partial solution, resulting in multiple full solutions that do not satisfy all constraints being removed from consideration.

3 Approach

This section describes the basic concept behind the lookahead value selection algorithm, along with certain implementation details.

3.1 Lookahead value selection

The lookahead approach presents a new method to perform value selection, where certain propagations are prematurely performed, in order to observe how selecting a candidate value affects the domain of the optimization function.

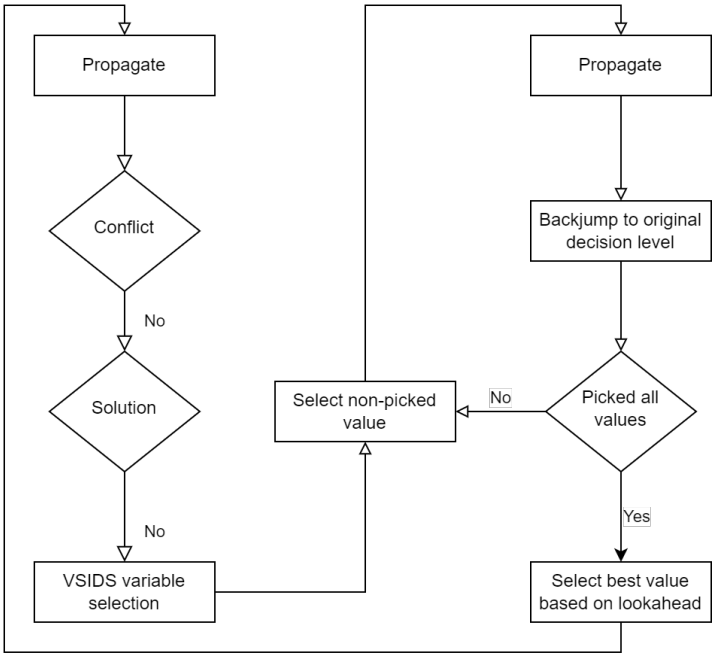


Fig. 1: An example of the main components of the lookahead constraint programming solver.

The core process of the lookahead approach is shown in figure 1. After VSIDS selects an SAT variable, first, a lookahead propagation is performed for one possible value for the variable, and the domain of the optimization function after the lookahead propagation is recorded, if there are no conflicts. Next, similarly, the lookahead propagation for the other possible value is performed, and again, assuming no conflicts, the domain of the optimization function is recorded. After both lookahead propagations are performed, the recorded optimization function domains for the two possible values are compared. Finally, the value that has the domain that has a more optimal value is ultimately selected.

In the case of a tie, the value that results in the smaller domain for the optimal value is selected. This is based on the fail first strategy [6]. If one of the lookaheads ends in a conflict, the lookahead performs an SAT analysis of the conflict and selects the other value. This allows the solver to quickly learn

additional clauses and nogoods. If a tie has not been resolved, the value is selected based on the default behaviour of Chuffed.

The lookahead propagation performs the following steps. It takes as input the selected variable and the candidate value, and first, create a temporary new decision level. Then, it updates the value of the selected variable and enqueues the updated clauses for propagation. After fully propagating, the updated optimization function domain is recorded, and if there has been a conflict, a sat analysis is performed. After, a backjump is performed to the previous decision level, essentially undoing all changes. Last, the recorded domain, and if there has been a conflict is returned. Algorithm 1 show the pseudocode of this function.

Algorithm 1 Lookahead Propagation

```
function LOOKAHEADPROPAGATE(variable, value)
    NEWDECISIONLEVEL
    assign[variable] ← value
    SAT.ENQUEUE(variable, value)
    PROPAGATE
    domain ← optVal.domain
    conflict ← sat.conflict
    if conflict then
        SAT.ANALYZE
    end if
    BACKJUMP(previousDecisionLevel)
    return domain, conflict
end function
```

3.2 Lookahead approach additions

Now that the main core of the lookahead approach has been designed, further additions are made to the algorithm, in order to improve performance in specific scenarios. These improvements include a slightly different variable selection when many variables have the same activity score, preferring full solutions, even if the domain could be further improved, a way to decrease unnecessary lookahead propagations, and fully stopping lookaheads after a certain amount of conflicts.

First, from early testing, it was noted that the algorithm would often repeatedly pick SAT variables generated from the same integer variable, and iterate toward one direction of its domain. Therefore, in order to avoid such a scenario, when multiple variables have the same activity score, SAT variables generated from boolean variables are prioritized. As the solver starts assigning different activity scores to each variable the effect of this issue diminishes.

Continuing, in order to produce solutions faster, when one of the two candidate values produces a fixed value for the optimization function, that value is preferred. In case both values lead to a solution, the more optimal solution is preferred. This allows the solver to produce solutions more often and earlier.

The next improvement reduces the number of lookahead propagations in case of conflicts. Based on the current implementation, if one value results in a conflict then there is no need to perform lookahead for both values. Therefore, by always starting the lookahead propagation with the non-default value, if there is a conflict after the propagation, then the default value can be selected without the need for second a lookahead propagation. Based on these additions, algorithm 2 shows the full branch function for the lookahead approach.

Algorithm 2 Lookahead Branch

```
function LOOKAHEADBRANCH
    candidates ← variables tied for the highest activity score
    variable ← an unassigned boolean variable from candidates if possible
    value ← default value for variable
    domain_0, conflict_0 ← LOOKAHEADPROPAGATE(variable, !value)
    if conflict_0 then
        return variable ← value
    end if
    domain_1, conflict_1 ← LOOKAHEADPROPAGATE(variable, value)
    if domain_0 has a more optimal value than domain_1 or (domain_0 most optimal
solution is the same as domain_1 and |domain_0| < |domain_1|) then
        return variable ←!value
    end if
    return variable ← value
end function
```

Last, after finding the optimal, or near-optimal solution, the utility of the lookahead approach diminishes, while the additional overhead of the lookahead propagations makes the algorithm significantly slower. Based on early testing, approximately 40% of the total solve time is spent on lookahead propagation, where a significant number of that time is after the solver has found the best solution. Stopping only after finding the first solution would not be ideal, as there is a high chance the first solution found is still far from optimal. Therefore, it was decided to use the number of conflicts caused after finding at least a solution as a stopping criterion for the lookahead approach. This is because after finding a near-optimal or optimal solution, the chance for a conflict to occur after a decision increases, due to the stricter domain bounds.

The code for Chuffed with the lookahead approach can be found in the following repository: `https://github.com/AZoumis/chuffed`

4 Experimental Setup

This section describes the experimental setup along with the data that will be used to test the performance of the lookahead algorithm.

To test the performance of the lookahead configuration, the implementation of the lookahead configuration will be tested against the VSIDS [7] value selection

of Chuffed without using the lookahead algorithm in certain metrics. Solution-based phase saving (SBPS) [3] will also be compared and combined with the lookahead implementation, as it is another value-selection heuristic that also similarly attempts to improve the performance of the solver on optimization problems.

The instances used will be the ones used for the MiniZinc challenge 2022 [8], with a time limit of 10 minutes for each instance.

The following metrics are used:

1. **Solve time**: The time it takes for Chuffed to solve a problem, hence the time it takes to find the optimal solution and prove optimality. This metric will show which configurations are overall the fastest at fully solving a problem. The graph is normalized in relation to the default configuration solve time, with the default configuration solve time being 1.0.
2. **Objective**: The best objective value found within the 10-minute time limit. The objective is normalized by setting the best value found by all configurations to 0.0, and the worst possible value found at any point by any configuration to 1.0. If a configuration has not found any solution, the objective is set to 2.0.
3. **Definite integral**: The definite integral, or area under the graph of the time vs objective graph of each instance. The y-axis, representing the best objective found at a point in time is normalized the same way as the plain objective metric. The overall integral is normalized in the same way as the solve time, with the default configuration having a normalized integral value of 1.0.

For calculating the mean for the solve time and integral, the geometric mean is used, as it produces a mean that better represents the actual performance differences between configurations [9].

5 Results

This section will present the averaged results for the lookahead approach, using the MiniZinc challenge 2022 [8]. It is expected that this approach might have an increased overall solve time due to the additional time overhead of this approach. However, when it comes to finding near-optimal and optimal solutions, the approach is expected to perform best for problems where there is a clear correlation between the objective function value and the variables selected, and for larger instances, where with the default value selection, Chuffed struggles to find near-optimal solutions. On the other hand, smaller instances with many local minima solutions will likely cause the lookahead approach to perform worse overall.

The MiniZinc challenge has a total of 95 optimization problem instances. Within 10 minutes, the default Chuffed configuration is able to find a solution satisfying all constraints for 82 out of the 95 problems, of which it proved optimality for 35. Continuing, SBPS found a satisfying solution for 82 instances

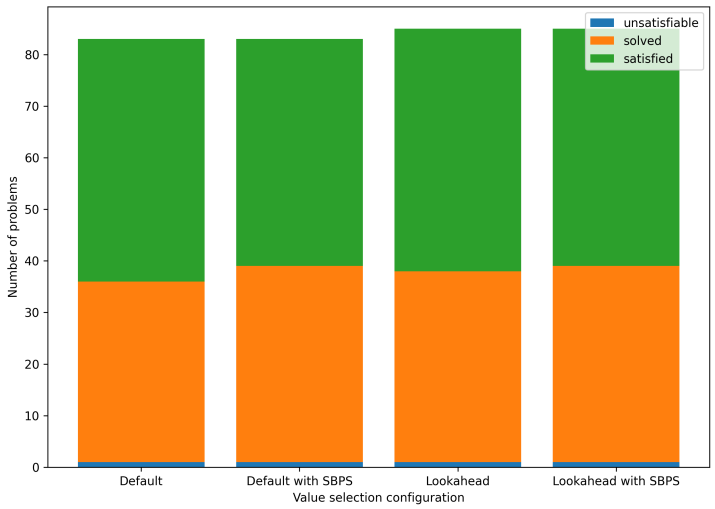
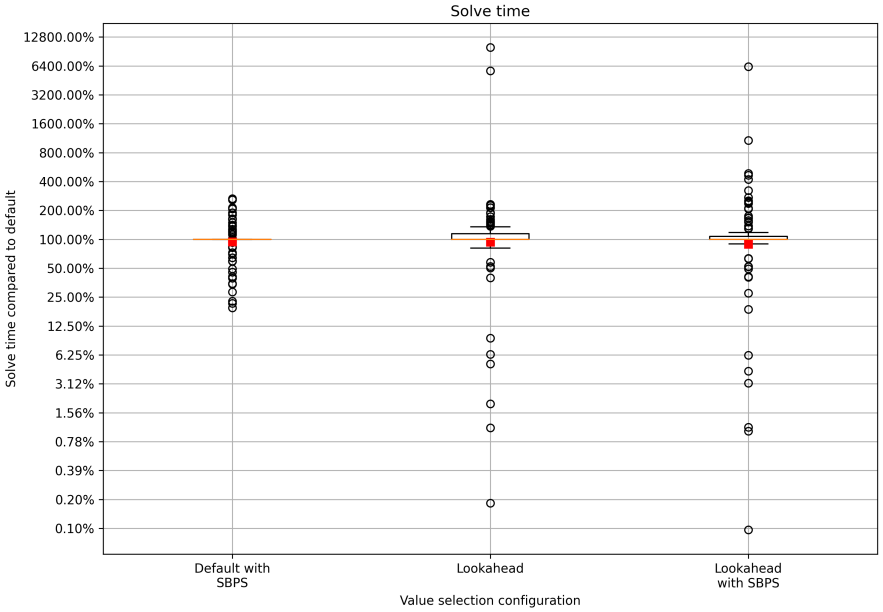


Fig. 2: Total number of instances where a solution was found (orange), or optimality was proved (blue)

and proved optimality for 38. Next, lookahead found a satisfying solution for 84 instances and proved optimality for 37. Last, lookahead with SBPS found a satisfying solution for 84 instances and proved optimality for 38.

Based on these results, the look-ahead approach has only a small effect on if a solution will be found. Nonetheless, both lookahead approaches were able to find optimality for overall 2 more instances than the default Chuffed configuration. However, SBPS was still able to fully solve 1 more instance when compared to the lookahead approach without SBPS. Nonetheless, the lookahead approach was able to find a satisfying solution for 2 more problems over the default and SBPS.

Figure 3 shows the average time taken to fully solve an instance when compared to default chuffed. In the case where the solver did not terminate within 10 minutes, the solve time is 600 seconds. The default lookahead has a solve time of 94.10%, and the lookahead approach with SBPS has a solve time of 88.76%. Although the mean solve time is lower than the default configuration solve time, looking at the IQR, the lower quartile and the medial are at 100% for the lookahead configurations, indicating that for 75% of the problems, the lookahead configurations had the same or slower solve time as the default configuration. With high p-values of 0.58 and 0.38, the null hypothesis cannot be rejected.



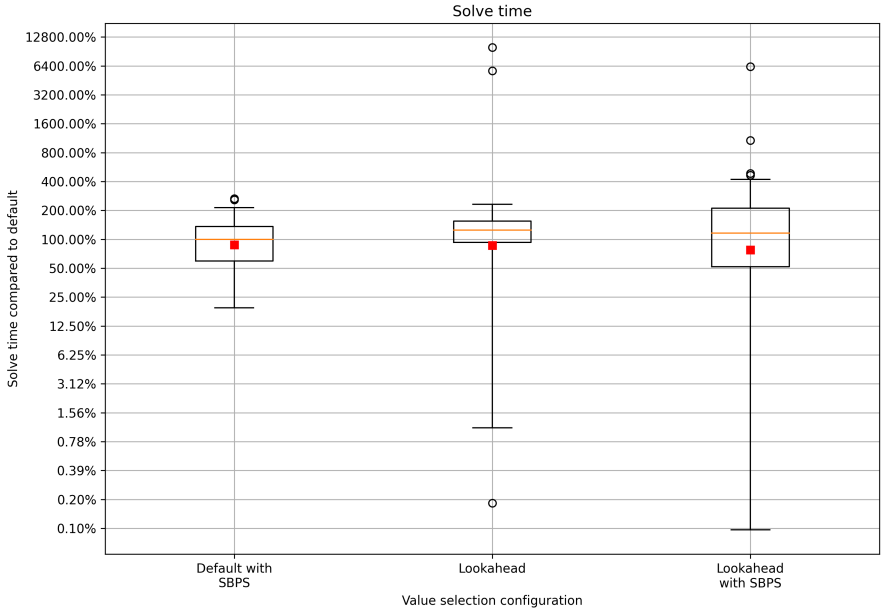
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.941057	1	1	1	1	1	-1.2959	0.196599
Lookahead	0.932888	0.810253	1	1	1.14456	1.348515	-0.54847	0.584023
Lookahead with SBPS	0.88766	0.895206	1	1	1.071042	1.17427	-0.87913	0.380456

Fig. 3: Average solve time for each configuration

Figure 4 shows the solve time results, but filtered to include only instances where at least one configuration managed to terminate before the 10-minute time limit. The recorded means are slightly lower, with a value of 86.36% and 77.76% for the lookehaead and lookahead with SBPS configurations respectively. Again, the p-values are similar to the ones recorded for figure 3, and therefore, the null hypothesis cannot be rejected.

The median for both lookahead configurations is higher than the default configuration, with 124.59% for the lookahead configuration, and 116.36% for the lookahead configuration with SBPS. This indicates that the lookahead approach potentially causes an overall time penalty to the total solve time. However, for some instances, the lookahead configuration significantly improves the solve time, leading to the mean being skewed towards a lower solve time.

The first reason why the lookahead configuration has a positive impact on the solve time for some problems is that finding an improved solution faster allows the solver to create stricter domains, creating many new nogoods, and decreasing the total search space that has to be explored. Next, since the lookahead approach performs a sat analysis when a lookahead leads to a conflict,



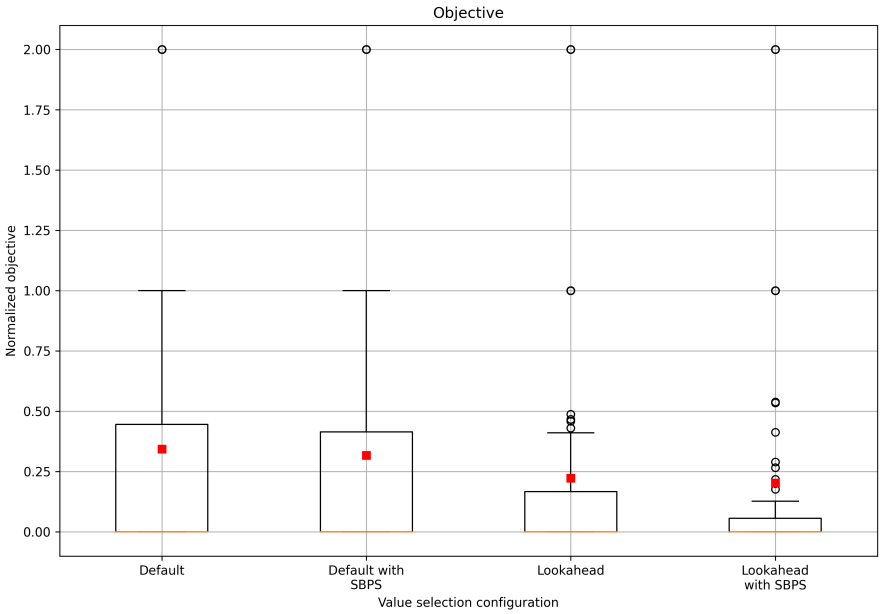
Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.87963	0.19366	0.596365	1	1.362735	2.132173	-1.3012	0.196587
Lookahead	0.863587	0.010923	0.928092	1.245853	1.543004	2.309313	-0.54619	0.586320
Lookahead with SBPS	0.777576	0.000942	0.519112	1.163622	2.107895	4.222537	-0.87793	0.382370

Fig. 4: Average solve time for each configuration, including only instances where at least one configuration terminated before the time limit (45 instances)

and furthermore has a fail first strategy, where it biases decisions that decrease the domain, it likely also creates many new clauses that limit the search space even more. Therefore, it appears that overall, these performance benefits are able to better overcome the time overhead of the lookahead approach for certain instances.

Continuing, figure 5 shows the average best normalized objective found after 10 minutes. Here, 1 indicates the worse solution found throughout the search by all solvers in this instance that satisfies all constraints, while 0 indicates the best overall solution. A solution of 2 indicates that no solution was found. For each configuration, in the order presented, a mean of 0.3436, 0.3178, 0.2234, and 0.2025 was recorded.

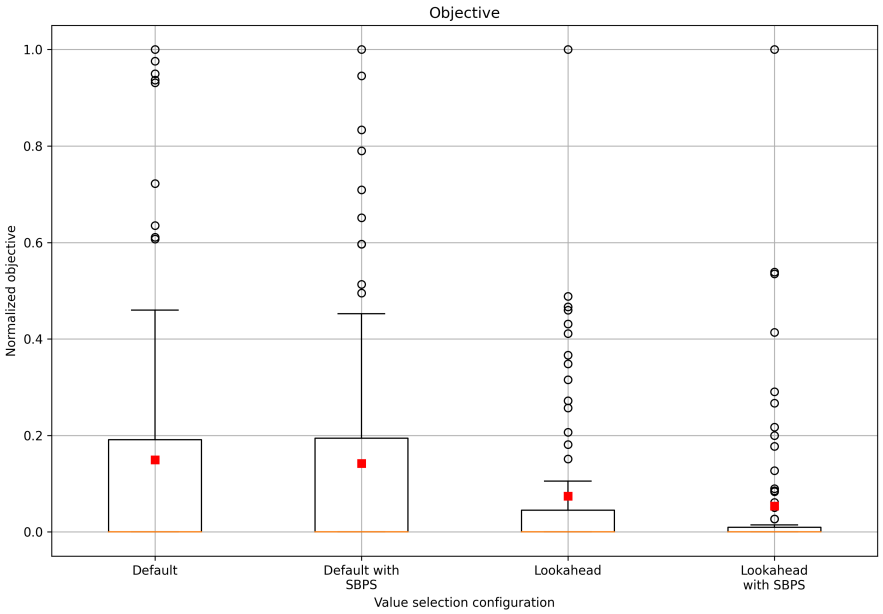
Although the means of the lookahead configurations are lower than the default configuration, with p-values of 0.12 and 0.07, the null hypothesis cannot be rejected. One reason for this is likely due to the high variance introduced by instances where no solution was found within the time limit. Therefore, it is likely that removing these values could result in a lower p-value.



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.343589	0	0	0.000396	0.446037	1	0	1
Default with SBPS	0.317792	0	0	0	0.41497	1	-0.31251	0.754997
Lookahead	0.223408	0	0	0	0.166477	0.411111	-1.54846	0.123194
Lookahead with SBPS	0.202514	0	0	0	0.055776	0.12727	-1.8143	0.071225

Fig. 5: Best objective found after 10 minutes

Figure 6 shows the average best normalized objective found after 10 minutes, excluding instances where at least one configuration did not find a solution within the time limit. With the filtered results, the p-values for the lookahead configuration are under 0.05, indicating that the null hypothesis can be rejected. Therefore, it can be stated with high confidence that for problems where a solution can be found within a time limit, the solution produced by the lookahead configuration will be on average closer to the optimal than the solution produced by the default configuration.

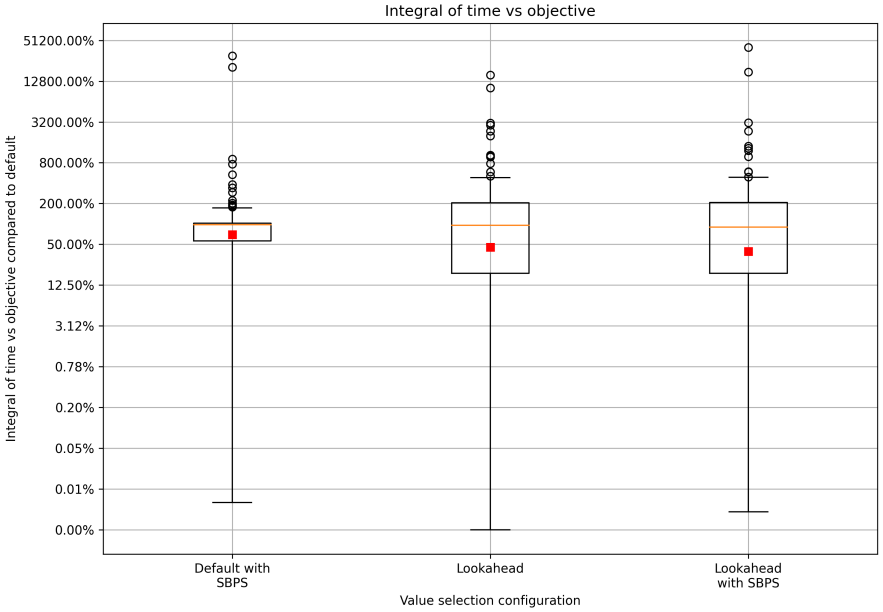


Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default	0.149093	0	0	0	0.191218	0.459642	0	1
Default with SBPS	0.141649	0	0	0	0.194444	0.452555	-0.17672	0.859957
Lookahead	0.074408	0	0	0	0.044602	0.105263	-2.05891	0.041166
Lookahead with SBPS	0.053656	0	0	0	0.009467	0.014514	-2.6927	0.007862

Fig. 6: Best objective found after 10 minutes, including only instances where all configurations found a satisfying solution or terminated within the time limit (79 instances)

Next, figure 7 shows the integral results compared to the default configuration. Overall, a mean of 45.26% is observed for the lookahead approach, and a mean of 39.48% is observed for the lookahead configuration with SBPS. Based on these results, low enough p-values were recorded, which provides evidence to reject the null hypothesis for all configurations. These results show the main advantage of the lookahead configuration, enabling it to find near-optimal solutions significantly faster than the default configuration, by better guiding the solver towards near-optimal solutions significantly faster than the default value selection.

SBPS was able to have a mean of 69.85%. However, when looking at the IQR for SBPS, it ranges from 55.94%, up to 102.70%, while the range for the lookahead and lookahead with SBPS ranges from 18.68% to 204.98%, and from 18.64%, up to 206.54% respectively. This indicates a higher variance in the ef-



Configuration	Mean	Lower whisker	Lower quartile	Median	Upper quartile	Upper whisker	t-value	p-value
Default with SBPS	0.698456	7.72E-05	0.559467	0.972961	1.027019	1.723453	-1.99523	0.047462
Lookahead	0.452958	3.05E-05	0.186817	0.950841	2.049801	4.838752	-2.75152	0.006513
Lookahead with SBPS	0.394823	5.64E-05	0.186463	0.894833	2.065496	4.86657	-3.12483	0.002061

Fig. 7: Average integral for each configuration

fectiveness of the lookahead approach when compared to SBPS, with many instances resulting in an overall worse integral performance.

Based on the results, it appears that the lookahead approach can provide significant advantages to Chuffed, for many optimization problems. In particular, the lookahead approach on average managed to find near-optimal and optimal solutions significantly faster than the default configuration. However, due to the additional time overhead of this approach, the solve time is often increased, and many problems and instances that do not greatly benefit from the lookahead approach will overall perform worse, with an increase in time taken to find similar solutions to the default configuration.

6 Related Work

The following section discusses related work that was used to implement the lookahead approach, such as MiniZinc and Chuffed, as well as other state-of-the-art approaches used by CP solvers to improve performance.

One of the main ways of improving the branching performance was through the use of heuristics, which better guide the solver through the search tree. One of the earlier concepts of heuristics was using a lookahead procedure called forward checking, which employs the **most likely to fail** principle, which branches on decisions that are more likely to fail[6]. This heuristic was shown to perform better than standard backtracking.

For hybrid CP-SAT solvers such as Chuffed, using SAT branching heuristics, such as the Variable State Independent Decaying Sum (VSIDS) heuristic, can further improve the variable selection[7]. VSIDS works by assigning an activity score to each variable and increasing said score based on how many clauses that variable is involved in. During branching, the variable with the highest activity is selected. Through the use of additive bumping and multiplicative decay, a bias towards variables that have a greater presence in recently learnt clauses is created. Due to VSIDS being independent of the current state of assignments, backtracking does not require any changes to the activity score, making this heuristic incredibly efficient.

Solution-based phase saving (SBPS) tries to emulate local search, by attempting to search through the neighbourhood around the current best solution.[3]. This approach is useful for optimization problems, finding more optimal solutions faster than the default value selection for certain problems. Similar to the looked approach, SBPS also affects the value selection.

Additional approaches that improve the performance of Chuffed using Machine learning exist, such as initializing an activity score for VSIDS[4] or predicting unsatisfiable cores[5].

Overall, many techniques have been used to make CP solvers more efficient for both satisfaction and optimization problems, and new techniques are constantly being implemented that further improve the performance of solvers in certain areas. Many of the mentioned heuristics rely on first gathering information about the problem, making them less impactful at the start. The lookahead approach attempts to better guide the SAT value selection, especially in the early stages of computation.

7 Future Improvements

The main improvements for the lookahead approach concern decreasing the time penalty of the additional lookahead propagations.

The first improvement would be to reuse the results of one of the lookahead propagations. The propagation following the value selection should be equivalent to the lookahead propagation for the same value. Therefore, using the outcome

of the lookahead propagation, instead of recomputing the propagation after selecting a value would help reduce the time penalty, as now, only a maximum of 1 additional propagation would occur, instead of 2.

Continuing, the two lookahead propagation could run in parallel, as the two propagations do not rely on the outcome of the other in order to compute their result. This should further reduce the time penalty of the lookahead configuration.

Next, performing a lookahead for multiple variables, instead of the two values for a single variable could improve the branching decisions made by the solver. In particular, it would potentially reduce cases where the solver reaches local minima and is unable to improve the objective, as it would provide more possible decisions at each level.

Last, an improved stopping criterion that more accurately stops the lookahead approach when it can no longer provide significant benefits to the solver could improve the overall solve time.

8 Conclusion

The lookahead approach to value selection for optimization problems has an average positive effect in regards to assisting the solver in finding near-optimal solutions faster. In particular, the lookahead configuration had a definite integral for the time vs objective graph 54.70% lower than the default Chuffed configuration on the generalized test suite.

Despite the overall improvements of the lookahead approach, performing lookahead propagations has a significant time cost. Therefore, the total solve time increased for most problems, despite the benefits of the approach. Furthermore, the time penalty also affects the early solution quality for many problems that do not benefit from lookaheads.

Overall, the lookahead approach presents a way to better emulate the performance of heuristic solvers, in particular, finding near-optimal solutions faster, but causes an overall slight increase in the total solve time.

References

[1] Daniel G. Bobrow and Bertram Raphael. “New Programming Languages for Artificial Intelligence Research”. In: *ACM Comput. Surv.* 6.3 (Sept. 1974), pp. 153–174. ISSN: 0360-0300. DOI: 10.1145/356631.356632. URL: https://doi.org/10.1145/356631.356632.

[2] G. Chu et al. *Chuffed, a lazy clause generation solver*. 2023. URL: https://github.com/chuffed/chuffed (visited on 03/23/2023).

[3] E. Demirović, G. Chu, and P.J. Stuckey. “Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers”. In: *International Conference on Principles and Practice of Constraint Programming* (2018). DOI: https://doi.org/10.1007/978-3-319-98334-9_7.

[4] R. van Driel, E. Demirović, and N. Yorke-Smith. "Learning Variable Activity Initialisation for Lazy Clause Generation Solvers". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2021). DOI: https://doi-org.tudelft.idm.oclc.org/10.1007/978-3-030-78230-6_4.

[5] R. A. van Driel. "Unsatisfiable core learning for Chuffed". In: *Delft University of Technology* (2020). DOI: http://resolver.tudelft.nl/uuid:18359282-e769-47d5-ac62-4ee6e1ec454e.

[6] Robert M. Haralick and Gordon L. Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial Intelligence* 14.3 (1980), pp. 263–313. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(80)90051-X. URL: https://www.sciencedirect.com/science/article/pii/000437028090051X.

[7] Jia Liang et al. "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers". In: June 2015. ISBN: 978-3-319-26286-4. DOI: 10.1007/978-3-319-26287-1_14.

[8] MiniZinc. *MiniZinc Challenge 2022 Results*. 2022. URL: https://www.minizinc.org/challenge2022/results2022.html (visited on 03/23/2023).

[9] J.J. Wallace P. J. Fleming. "How not to lie with statistics: the correct way to summarize benchmark results." In: *Communications of the ACM* (1986). DOI: doi.org/10.1145/5666.5673.

[10] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. First edition. Elsevier, 2006. ISBN: 9780080463803.

[11] M. Wallace and N. Yorke-Smith. "A new constraint programming model and solving for the cyclic hoist scheduling problem". In: *Constraints* (2020). DOI: https://doi-org.tudelft.idm.oclc.org/10.1007/s10601-020-09316-z.

[12] G. Weil et al. "Constraint programming for nurse scheduling". In: *IEEE* (1995). DOI: 10.1109/51.395324.