

École polytechnique de Louvain

Restart Strategies with NoGoods in the MiniCP Constraint Programming Solver

Authors: **Nicolas CRUYSBERGHS, Louis LABBÉ**

Supervisor: **Yves DEVILLE**

Readers: **Vianney COPPÉ, Augustin DELECLUSE, Yves DEVILLE**

Academic year 2022–2023

Master [120] in Computer Science and Engineering

Abstract

Restart strategies are meta-heuristics used in Constraint Programming to avoid heavy-tailed behaviors encountered by systematic searches. In doing so, they may reexplore certain areas of the search tree, which could decrease efficiency. A solution to this problem is to incorporate learning with the help of nogoods, which are sequences of decisions leading to previously visited regions. Their management guides the later part of the search towards unexplored areas of the search space. The goal of this thesis is to integrate an efficient base, storing nogoods, in the MiniCP constraint programming solver and then to study their impact when used on a regular restart strategy and a large neighborhood search. Along the implementation, the nogood base was also extended to support an additional type of decision. Experimental results have shown that nogoods can improve the performances of a regular restart strategy and worsen the performances of a large neighborhood search.

Acknowledgements

We would like to express our gratitude to Pr. Yves Deville, who agreed to supervise our master's thesis subject and provide us with advice and support throughout the year.

We would also like to thank Augustin Delecluse for providing us with advanced constraints to model the tested problems, and for agreeing to be part of the jury with Vianney Coppé. The same goes for Pr. Christophe Lecoutre for his clarifications during the launch of this thesis.

Lastly, we would also like to sincerely thank our families and loved ones for their support and encouragement. Special thanks to Luc Cruysberghs for the proofreading¹.

¹DeepL and ChatGPT were also used to help translate and improve the writing style of certain sentences.

Contents

1	Introduction	1
2	Background	3
2.1	Constraint Programming	3
2.1.1	Constraint Satisfaction Problem	4
2.1.2	Constraint Optimization Problem	6
2.1.3	Propagation	10
2.1.4	Branching	11
2.1.5	Search	12
2.2	Restarts	14
2.3	Large Neighborhood Search	16
2.4	Nogoods	18
2.4.1	Nogood Recording	18
2.4.2	Restarts with Learning	21
2.4.3	Nogood Constraints	22
2.4.4	Watched Decisions	24
2.4.5	Minimizing Nogoods	28
2.5	MiniCP	30
2.5.1	State variables	30
2.5.2	Domains	31
2.5.3	Variables	33
2.5.4	Constraints and Propagation	33
2.5.5	Branching	34
2.5.6	Search	34
2.5.7	Practical Problem Modeling	36
3	Integration of Nogoods in MiniCP	39
3.1	Decision Path Saving	39
3.1.1	Decision	40
3.1.2	Path saving	40
3.2	Creating the Nogoods	41

3.3	Storing the Nogoods	44
3.3.1	Nogood Base	44
3.3.2	Watched Decisions	45
3.3.3	Variable Base	46
3.3.4	Example	47
3.4	Inferences with Nogoods	50
3.4.1	Notifier Procedure	50
3.4.2	Update Watched Decisions	51
3.4.3	Single Nogoods	52
3.4.4	Example	53
3.5	Supported Branchings	55
3.5.1	Binary Labeling	56
3.5.2	Binary Split	57
3.6	Large Neighborhood Search with Nogoods	59
3.7	Solving Problems with Nogoods	61
4	Experiments and Analyses	65
4.1	JobShop Problem	66
4.1.1	Nogoods Threshold Selection	66
4.1.2	Restarts with Binary Labeling	68
4.1.3	Restarts with Binary Split	73
4.2	Traveling Salesman Problem	76
4.2.1	Large Neighborhood Search	76
5	Discussion and Perspectives	81
5.1	Discussion	81
5.1.1	Restarts	81
5.1.2	Large Neighborhood Search	83
5.2	Improvements	84
5.3	Perspectives	86
6	Conclusion	87
A	Alternative Version of Nogood Generation	89
B	Minimizing Randomness for Nogood Comparison	91
C	Additional Benchmarks	93
C.1	Graph Coloring Problem	93
C.2	JobShop Problem	94
C.3	Traveling Salesman Problem	95
C.3.1	Node Analysis	95

C.3.2	Alternative Heuristic	96
C.3.3	Alternative Relaxation Phase	96

Chapter 1

Introduction

Restart strategies are well known meta-heuristics that guide a search process to avoid the heavy-tailed behaviors often encountered in systematic searches, such as Depth-First Search (DFS). They aim to explore different regions of the search space by periodically resetting the search process. However they are lacking in systematicity since regions of the search tree can be visited multiple times.

An approach to address this problem is to identify the branches taken during the search, named decisions, that led to such regions and prevent their repetition. This is where nogood recording comes into play as it adds learning to restarts. A nogood is the set of decisions leading to an already visited region and is used by including a clause that prevents its reexploration during the subsequent restarts' searches.

The first objective of this master's thesis is to integrate nogoods in MiniCP, a lightweight open-source constraint programming solver developed for educational purposes[1]. Such an integration has been undertaken by Skjelnes recently[2]. However, his implementation involves a substantial number of object instantiations for each recorded nogood decision. Since most of these are stateful objects, i.e. objects that can return to a previous state, they affect the possible gain in performance that the usage of nogoods could have over restart strategies without learning.

In this master's thesis, they are integrated differently by following the framework of Lecoutre described in [3] which avoids these instantiations by creating a separate base where everything is managed independently of the constraint programming solver. This framework also lightens the process of assessing whether the current decision sequence has ever been taken.

The second objective of this master’s thesis is to evaluate the impact of nogoods when incorporating learning to different restart strategies in MiniCP. For this purpose, different variations of nogoods will be evaluated on a regular restart strategy and a Large Neighborhood Search (LNS) strategy. The problems used for evaluation are two Constraint Optimization Problems (COPs), namely the JobShop Problem and the Traveling Salesman Problem (TSP). In addition, due to the many parameters affecting the search process, such as branching types, ordering heuristics, and the relaxation methods used in LNS, different variations will be tested to investigate their effect on nogood performances.

Chapter 2

Background

This chapter aims to describe the different theoretical concepts needed for this paper. Section 2.1 introduces the important concepts in constraint programming. Section 2.2 and 2.3 will discuss the fundamental aspects of the two search strategies, namely restarts and large neighborhood search, in constraint programming. Section 2.4 describes nogoods and how they are used to add learning to constraint programming solvers. Finally, Section 2.5 explains the main features of the MiniCP architecture used to solve constraint programming problems.

2.1 Constraint Programming

Constraint Programming (CP)[4] is a problem-solving paradigm that uses constraints to solve complex problems. The problems are typically combinatorial in nature and can be declaratively modeled by CP by the means of constraints restricting a search space defined by variables and their possible values. The primary goal is then for CP to identify feasible solutions within the search space, which occurs when it satisfies all defined constraints. This differs from procedural approaches as CP defines possible solutions without specifically knowing how to find them.

The problems solved by CP are of two types: problems where a single solution needs to be found, and those which aim to identify the optimal solution according to predefined objectives. These problems are referred to as Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP), respectively. They are described in Sections 2.1.1 and 2.1.2.

One notable Constraint Programming characteristic is that it is a resolution technique capable of exploring the entire search space in order to discover all

solutions. However, conducting an exhaustive search of the entire unfiltered search space is often impractical due to its computational cost. To address this challenge, CP systematically eliminates unfeasible regions from the search space through the use of propagation mechanisms linked with the problems constraints. These propagation mechanisms help narrow down the search space and will be detailed in Section 2.1.3. However, problems often become too large for the propagation to directly lead to solutions. Thus, branchings are made to reduce the problem size into multiple subproblems. These branchings, explained in Section 2.1.4, are handled by a search algorithm covered in Section 2.1.5.

2.1.1 Constraint Satisfaction Problem

A CSP is a mathematical problem representing a combinatorial problem. It is defined by variables having discrete and finite domains, along with constraints that limit the values to which variables can be assigned to.

Mathematically, a CSP is a triplet $\langle X, D, C \rangle$ [1] where $X = \{x_0, x_1, \dots, x_{n-1}\}$ denotes a set of n variables, $D = \{D(x_0), D(x_1), \dots, D(x_{n-1})\}$ represents a set of their respective domains of values, and $C = \{c_0, c_1, \dots, c_{m-1}\}$ represents a set of m constraints. Each domain $D(x)$ specifies the possible values of variable x , while C restricts combinations of values from a subset of variables X . Additionally, a variable x is referred to as fixed when its domain contains only one element such as $|D(x)| = 1$.

The process of solving a CSP involves assigning variables to values from their domain in such a way that all constraints are satisfied. A candidate solution, denoted as σ , is defined as a function that assigns each variable to a value from its domain, i.e., $\forall x \in X : \sigma(x) \in D(x)$. The validity of this solution is verified by constraint valuation. This valuation over a constraint c , involving all variables and a candidate σ , is represented by the Boolean value $c(\sigma) = c(\sigma(x_0), \sigma(x_1), \dots, \sigma(x_{n-1}))$. σ is considered a solution of the CSP if it satisfies all constraints, i.e. $\forall c \in C : c(\sigma) = \text{True}$. On the other hand, if at least one of the constraint's valuation is not *True*, it indicates an inconsistency. Also, a problem having at least one solution is described as satisfiable while those that do not are unsatisfiable. The set of solutions of a CSP $\langle X, D, C \rangle$ is denoted by $S(X, D, C)$.

Map Coloring Problem

The map coloring problem is a good example that illustrates this point. Indeed, it is a well known problem in combinatorial optimization and graph theory that can be modeled as a CSP. It consists of assigning colors to regions of a map in such a

way that no region of the same color share a boundary. It can easily be transposed to a graph coloring problem where vertices V represent the regions and edges E connect the neighboring regions such that no edge links two vertices of the same color in the graph $G = (V, E)$.

Formally, a map coloring problem with n regions, m colors $\{d_0, \dots, d_{m-1}\}$ and a graph representation $G = (V, E)$ can be modeled as the CSP $\langle X, D, C \rangle$ where each component is defined as:

- $X = \{x_0, x_1, \dots, x_{n-1}\} = \{x_i \mid i \in I\}$ where I is the index set $\{0, \dots, n-1\}$ and each x_i is a variable representing the color of region i .
- $D = \{D(x) \mid x \in X\}$ where $\forall x \in X : D(x) = \{d_0, \dots, d_{m-1}\}$.
- $C = \{x_i \neq x_j \mid i, j \in I^2, i < j, (x_i, x_j) \in E\}$ where $x_i \neq x_j$ imposes that x_i and x_j do not have the same color and where the notation (x_i, x_j) denotes an edge between the vertex x_i and x_j .

To illustrate this concept, consider a simple example such as the map of some of Australia's principal states and territories², as depicted in Figure 2.1 along with its corresponding graph representation[5].

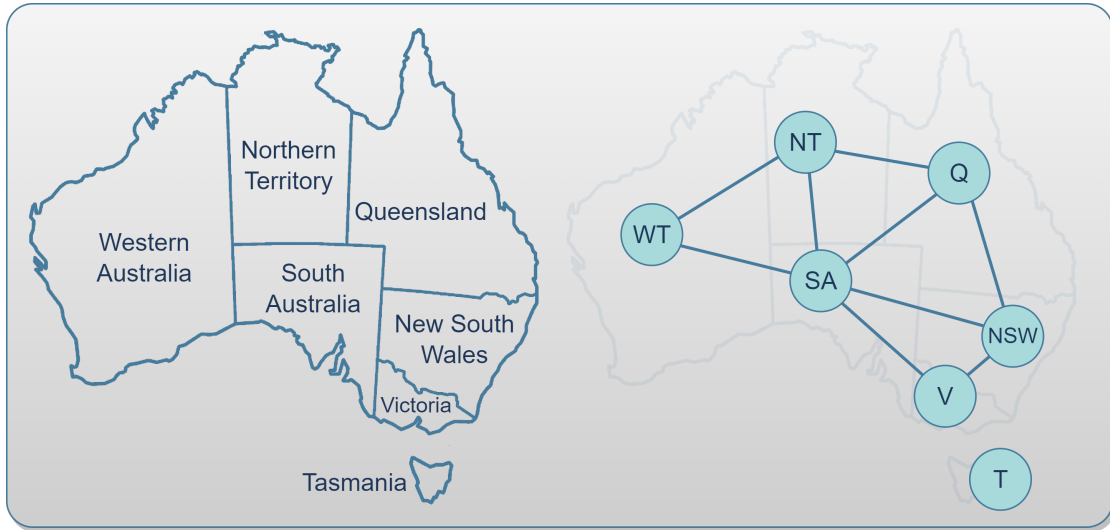


Figure 2.1: Map of Australia along with its graph representation.

In the context of the map coloring problem, each state or territory can be considered as a region that requires coloring. When only the three colors red,

²As stated, not all its states and territories are depicted in Figure 2.1.

purple, and green are allowed to solve the problem, this leads to a CSP with 3^7 candidate solutions. Figure 2.2 shows one of the solutions that satisfy all the constraints.

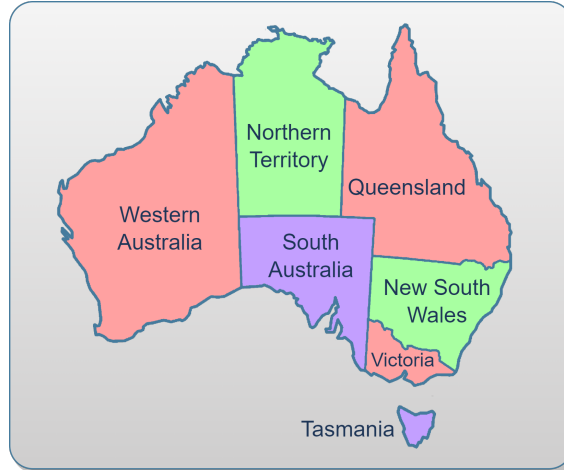


Figure 2.2: A solution to the map coloring problem of Figure 2.1.

Undoubtedly, as the number of regions increases and the available colors become limited, the map coloring problem grows to be increasingly challenging. In such cases, using a constraint programming solver proves to be highly convenient due to its simple problem modeling.

2.1.2 Constraint Optimization Problem

A COP on the other hand is a CSP with an optimization objective. Its goal is no longer to simply find a solution that satisfies the constraints, but to find one that optimizes its objective function.

It is defined by the quadruplet $\langle X, D, C, f \rangle$ [1] where f is an objective function that maps a candidate solution to a real number. The valuation of this function, i.e. objective value, over a candidate solution σ with $X = \{x_0, x_1, \dots, x_{n-1}\}$, is the real value $f(\sigma) = f(\sigma(x_0), \sigma(x_1), \dots, \sigma(x_{n-1}))$. Therefore all the solutions to such a problem may not optimize the objective function to the same extent. The optimal one, denoted as σ^* , is the solution where $f(\sigma^*) = \min_{\sigma \in S(X, D, C)} f(\sigma)$. It implies that no other solution has a better objective value. Although a minimization is used here, a maximization can also be modeled by simply taking the opposite of the valuations of f . Note that this valuation will also be referred to later as the objective value.

JobShop Problem

A combinatorial problem that can be modeled as a COP is the JobShop Problem, also known as JobShop Scheduling. It is a famous scheduling problem that consists in arranging multiple jobs on several machines. Each job consists of a sequence of tasks that must be performed in a specific order, with each task assigned to a particular machine.

The problem is therefore subject to two main groups of constraints, namely precedence constraints and no overlap constraints. Precedence constraints ensure that each task follows a specific order within its corresponding job, while no overlap constraints prohibit simultaneous processing of tasks on the same machine. Also, a task cannot be interrupted once it has started and must therefore run until completion.

The objective of the problem is to find an assignment of start times, one for each task, that satisfies the constraints and minimizes the makespan. The makespan represents the time from the earliest start time of the jobs to the latest end time and is also referred to as the length of the schedule.

To model the JobShop Problem as a COP, some variables first need to be defined, namely:

- J the set of jobs
- T the set of tasks with T_i representing the set of tasks of job i .
- $\mathcal{D}$ the set of duration of each task where d_{ij} is the duration of j^{th} task of job i .
- h the sum of the duration of all the tasks, i.e. $h = \sum_{i \in J} \sum_{j \in T_i} d_{ij}$.
- M the set of distinct machines with m_{ij} representing the machine used by j^{th} task of job i .

Now, the JobShop Problem can be modeled as the COP $\langle X, D, C, f \rangle$ where each component is defined as:

- $X = \{x_{ij} \mid i \in J, j \in T_i\}$ where x_{ij} represents the starting time of j^{th} task of job i .
- $D = \{D(x) \mid x \in X\}$ where $\forall x \in X : D(x) = \mathbb{N}$. In practice, since the problem minimizes the makespan, the upper bound of the domains can be restricted to h , s.t. $\forall x \in X : D(x) = \{x \in \mathbb{N} \mid x \leq h\}$ without losing any optimal solution.

- $C = C_p \cup C_o$ where C_p are the precedence constraints and C_o the overlap constraints. C_p is specified as $\{x_{ij} + d_{ij} \leq x_{i,j+1} \mid i \in J, j \in T_i \setminus \{T_{i,|T_i|-1}\}\}^3$ where $x_{ij} + d_{ij}$ represents the end of j^{th} task of job i . Since C_o is about overlap constraints on machines, another variable needs to be defined now that X has been introduced. Let $\mathcal{M}$ be another representation of tasks information organized by machine where each task is represented by its start and duration. $\mathcal{M}$ is defined by $\{m_k \mid k \in M\}$ where $\forall k \in M : m_k = \{(x_{ij}, d_{ij}) \mid i \in J, j \in T_i, m_{ij} = k\}$. This allows to define C_o as the set of constraints $\{x_a + d_a \leq x_b \vee x_b + d_b \leq x_a \mid m \in \mathcal{M}, (x_a, d_a) \in m, (x_b, d_b) \in m \setminus \{(x_a, d_a)\}\}$.
- $f = \max_{i \in J} (x_{i,|T_i|-1} + d_{i,|T_i|-1})$. The function maps a candidate solution to its makespan.

As a reminder of the COP objective, the aim is to find an optimal solution σ^* that minimizes the objective value, which corresponds to the makespan in this problem, so that $f(\sigma^*) = \min_{\sigma \in S(X,D,C)} f(\sigma)$.

Since the equations make it seem complex, Figure 2.3 illustrates an example with two of its solutions shown in Figure 2.4[6]. The solution **b)** is an optimal solution but not the only one since T_{12} could start at time 7 without modifying the makespan.

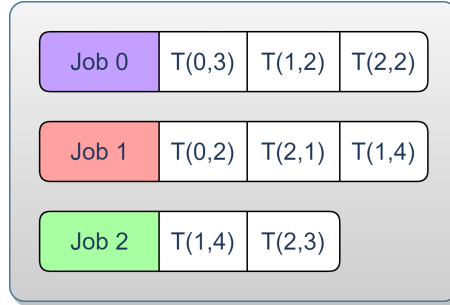


Figure 2.3: Example of JobShop Problem with 3 jobs.

³ $x_{i,j+1}$ represents the start of the task following the j^{th} task of job i . The set generator then stops at the penultimate task.

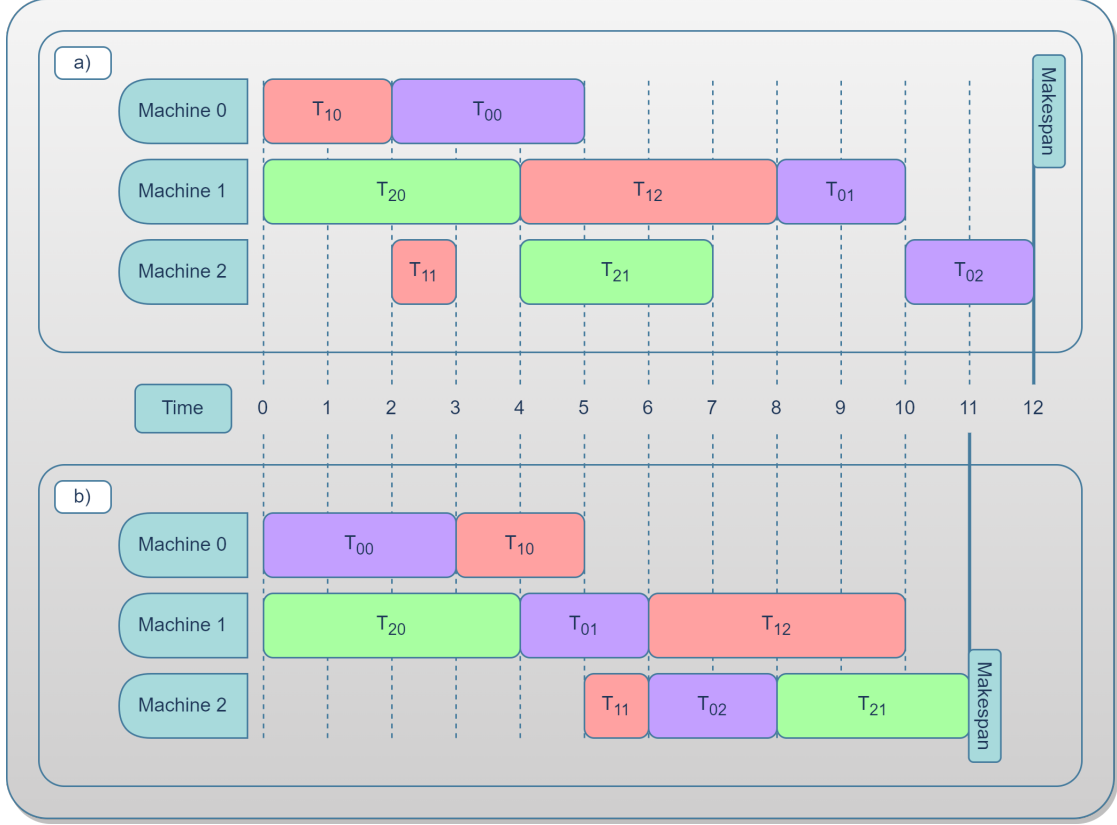


Figure 2.4: Example of two solutions to the JobShop Problem of Figure 2.3.

Traveling Salesman Problem

Another well known combinatorial problem that can be modeled as a COP is the Traveling Salesman Problem (TSP)[7, 8]. It consists in visiting some locations, separated by a given distance from each other, so that each location is not visited twice. Additionally, the path formed needs to be a cycle so that it returns to the place of departure. The objective is to find the shortest cycle to realize this travel, i.e. the travel with the minimal distance.

Thus, TSP can be translated in graph theory as an undirected weighted connected graph $G = (V, E)$. Locations are the vertices V and the links between the locations are the edges E , where their associated weight is their distance. The goal is then to find the minimal weighted Hamiltonian cycle in G .

Given $dist(l_i, l_j)$, a function providing the distance between two locations l_i and l_j , a TSP with locations $L = \{l_0, \dots, l_{n-1}\}$ can formally be modeled as the COP $\langle X, D, C, f \rangle$ where each component is defined as:

- $X = \{x_0, \dots, x_{n-1}\}$ where x_i is a variable representing the location visited right after location l_i , i.e. the successor location of l_i . X is then denoted the sequence of successors.
- $D = \{D(x_0), \dots, D(x_{n-1})\}$ where $\forall x_i \in X : D(x_i) = L \setminus \{l_i\}$
- $C = C_p \cup \{C_c\}$. C_p is made of difference constraints such that $C_p = \{x_i \neq x_j \mid x_i, x_j \in X\}$. C_c is a constraint restricting X so that it forms a circuit with no sub-circuits. Globally, C imposes that the sequence of successors forms a Hamiltonian cycle.
- $f = \sum_{i=0}^{n-1} \text{dist}(l_i, x_i)$ where f maps a candidate solution to the sum of covered distances.

2.1.3 Propagation

Constraint propagation[4, 9] serves as a powerful tool to eliminate unfeasible regions from the search space. These regions are candidate solutions that do not align with constraints, so ignoring them as early as possible prevents useless calculations. Each constraint defined in the problem model is associated with a propagation procedure responsible for identifying and removing unfeasible regions which corresponds to removing values from domains. These propagation procedures are initiated when needed by **propagators** that use the variables, their current domain, and their previous domain to make deductions. The act of removing regions from the search space, done by the propagators in this case, is called pruning.

Constraints use various methods to prune variable domains, each offering a trade-off between computational intensity and pruning strength. The first method, known as Bound Consistency, focuses solely on applying constraints to the bounds of variable domains, disregarding the specific values within the minimum and maximum bounds. In contrast, the second method, Domain Consistency, examines the entire domain when applying constraints, resulting in significantly longer computation times compared to the former approach. Using domain propagation allows for an even stronger form of consistency named Generalized Arc Consistency (GAC), meaning that all constraints involving two or more variables in a constraint satisfaction problem are satisfied simultaneously[3]. The selection between the different consistencies is left to the implementer, as each has advantages and disadvantages. Figure 2.5 shows an example of both strategies when a new constraint links two variables mathematically.

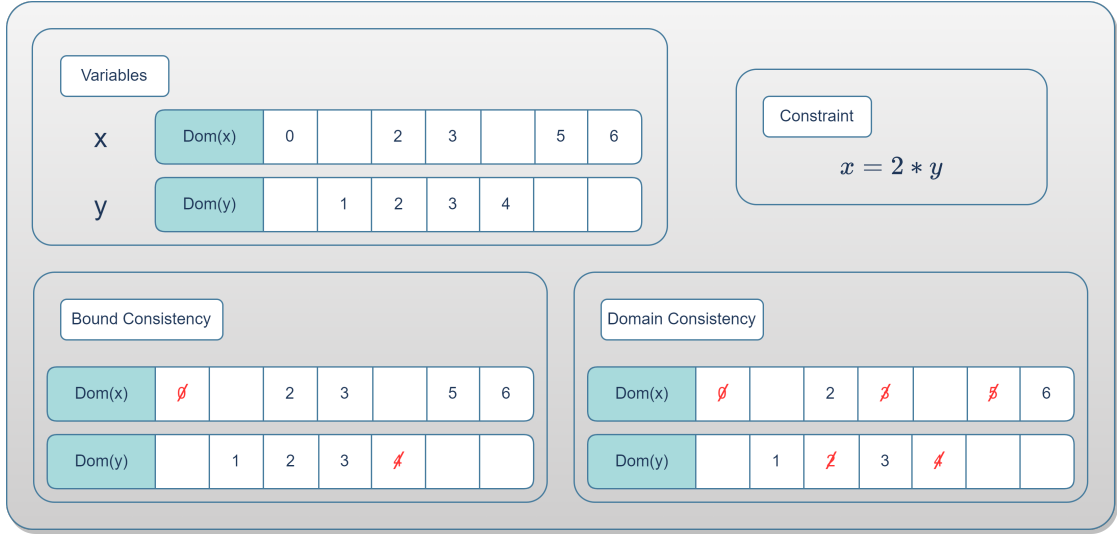


Figure 2.5: Example of Bound and Domain Consistency in action.

Once no further pruning can be done via propagation, a **fixpoint** is reached, which can have 3 different states:

- **Feasible Solution** - The variables in the problem have each been assigned a value that meet the specified constraints, and no further pruning is needed.
- **Inconsistency** - A propagator's pruning lead to an empty domain, meaning no possible assignment for that variable. Thus, there is no valid solution that satisfies all constraints simultaneously.
- **Inconclusive** - The propagators are not able to further prune the search space. These variables are left in an ambiguous or partially constrained state and solutions cannot be found until further constraints are added. Although a complete solution has not been found, the algorithm may still have made some progress by narrowing down the feasible domain for these variables.

However, larger problems often lead to inconclusive fixpoints when establishing the problems constraints. So an additional technique that uses branching recursively is applied to further constrain the domains of the remaining unfixed variables.

2.1.4 Branching

Branching divides problems into smaller subproblems by adding constraints. This allows the propagation algorithm to go further in depth and find valid solutions when initial modeling constraints are not sufficient. Formally, as stated in [1],

an n -way branching for a CSP defined by $\langle X, D, C \rangle$ is a set of $\{\langle X, D, C \cup \{c_0\} \rangle, \dots, \langle X, D, C \cup \{c_{n-1}\} \rangle\}$ such that these two conditions are met:

$$S(X, D, C) = \bigcup_{i \in 0 \dots n-1} S(X, D, C \cup \{c_i\})$$

$$\bigcup_{\substack{i, j \in 0 \dots n-1 \\ i < j}} S(X, D, C \cup \{c_i\}) \cap S(X, D, C \cup \{c_j\}) = \emptyset^4$$

The first condition simply states that solutions may not be lost when branching. The second maximizes efficiency by stating that solutions cannot be repeated.

For example, given a CSP with $X = \{x\}$, $D(x) = \{v_0, \dots, v_{n-1}\}$ and one of its value v_i , here are two common 2-way branchings:

- Binary Labeling : $x = v_i \vee x \neq v_i$
- Binary Split : $x \leq v_i \vee x > v_i$

2.1.5 Search

A recursive usage of a branching strategy builds a **search tree** in which the root node represents the original CSP, and each child node represents a subproblem obtained through branching. Due to the branching properties, no potential solutions are lost since the union of all subproblems covers the entirety of the original problem.

The search tree is traversed and expanded using a **search algorithm**, typically following a Depth-First Search (DFS)[10] order. The recursivity from DFS is important in this approach because it ensures that no useless branches are generated during the search process. For every node, the search algorithm has a CSP that reaches a **fixpoint** through propagation which, as explained above, can be in three different states leading to different actions being taken. In the case of an inconclusive **fixpoint**, a new branching is made from this node to further constrain the problem. If an inconsistency is reached, the search backtracks, meaning heads back up the tree, to a node that still has children to expand to explore other paths. Lastly, in the case of a feasible solution, the search can stop if only one solution is wanted, usually for CSPs, or continues if all solutions are needed, which is important for COPs to prove optimality.

The branchings used during the search are often decided by a variable or value ordering heuristic that determines the shape of the tree. Different problems may

⁴ $\emptyset$ represents the empty set.

require different heuristics to achieve a faster and more effective search for the optimal solution. By carefully selecting and implementing appropriate variable and value ordering heuristics, the search algorithm can effectively guide the branching decisions and shape the search tree in a way that accelerates the search process towards a solution, in the case of a CSP, or the optimal solution in the case of a COP.

Search Example: N-Queens Problem

One well known problem that is often used to represent the search algorithm is the N-Queens problem. The goal is to place N queens on an $N * N$ chessboard in such a way that no two queens can attack each other. Figures 2.6 and 2.7 show a search example that solves this problem on a $4 * 4$ grid. The notation used is as follows: $Q_i = j$, denotes that the queen on column i is placed on row j , with the top left square corresponding to row 0 and column 0. For each step, the decision taken is shown on the left while the resulting state after propagation is shown on the right. The leaf nodes can either be inconsistencies or solutions, respectively white or green squares on the figures.

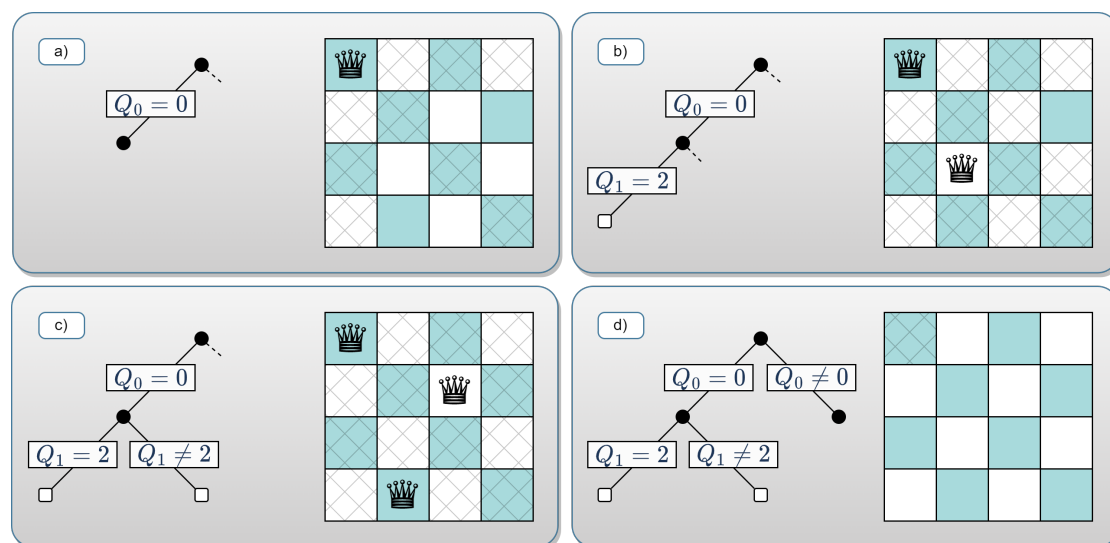


Figure 2.6: Search example for N-Queens problem.

- a) The first branching is made from the root node, which represents the initial CSP. At this step, the queen in column 0 is added, leading to additional constraints on the available positions for the remaining queens. However, since the unplaced queens still have more than one available spot, no further propagation can be carried out at this point.

- b) The second queen, column 1, is added via a new branching made on top of the previous one. This branching directly creates an inconsistency though because the third queen is left with no available spots on the board. Thus, no solutions can derive from this decision and a backtrack must be made.
- c) The backtracking goes up until it finds a node that still has a decision to expand. In this example, it is shown as three dots on the right of a node. This step now prevents the second queen from being placed in row 2. Even though the branching did not add a queen, it created a column with only one position left. Thus the propagator takes over and creates a chain reaction of decisions where queens are placed in positions where they become the only viable choice. In this node, two queens were placed but it created an inconsistency in the last column, hence initiating a backtrack.
- d) This backtrack goes all the way to the root node and the new decision taken now prevents the first queen from being placed in row 0. Nothing else can be inferred from this decision so the search continues as no solution has been found and there are still some nodes to expand.

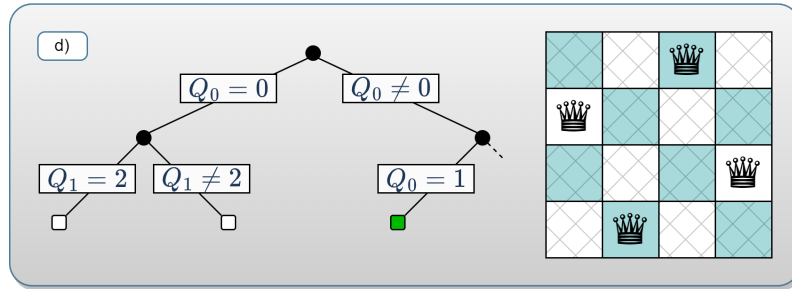


Figure 2.7: Solution found during the search example for N-Queens problem.

- e) With the final branching placing the first queen on row 1, the propagation can take over and place all the remaining queens, ultimately finding a solution to the problem. If it was not deduced as of now, the heuristic used is simply choosing the leftmost unplaced queen and branching on the lowest available row value. At this state, the search can go on to find all possible solutions in the search tree or stop if desired.

2.2 Restarts

The search process in CP often exhibits a heavy-tailed characteristic, where certain portions of the search tree demand extensive exploration before a solution, if any,

is found. Conversely, some branches lead to quick solutions. As search strategies may encounter this phenomenon while traversing fruitless subtrees, the concept of restarts has been introduced, to mitigate these issues. This concept consists in undoing all preceding search decisions and using an alternative search strategy to explore different segments of the search tree[11]. These restarts are activated after reaching a chosen stopping criterion for the search such as time, number of failures or nodes traveled.

Implementing multiple different search strategies can take a long time so a common practice is to have randomization involved in the process. This randomization can work on the variables or values chosen for branching. Thus, for every restart, the search tree will have a different shape with the nodes completely rearranged.

While restarts prove to be effective for specific CP problems, their average performance is slower than a regular search when the entire tree needs to be searched. This is because some search heuristics may overlap leading to states being traversed multiple times.

Restart Example: N-Queens Continued

The N-Queens problem shown in the previous section also happens to be heavy-tailed since the initial queen placement significantly influences the final arrangements. The example in Figure 2.7 could have been solved with just one branching $Q_0 = 1$ instead of the whole tree. Nonetheless, due to the problem's relatively small size, using restarts is not necessary, as the search tree remains manageable. However, as the grid size increases, the problem's complexity grows, which is where restarts become interesting.

Figure 2.8 shows the only distinct solution for the 6-Queens problem. If the same heuristic used in Figure 2.6 was used for this 6*6 grid, the search would have to iterate through every possible node where the queen is in the corner first. When using restarts, it would iterate until reaching a certain stopping criterion, such as encountering 5 inconsistencies, and then initiate a restart with a new heuristic. This alternate heuristic could place the queen somewhere else than the corner, steering the search in a more favorable direction.

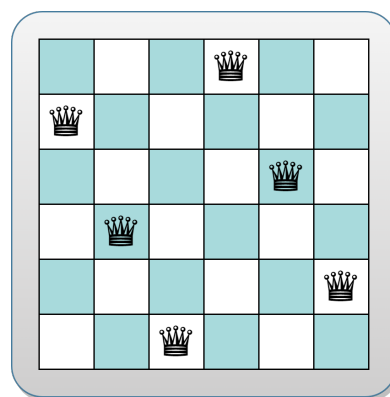


Figure 2.8: 6-Queens solution.

2.3 Large Neighborhood Search

The larger the problem, the more difficult it can be to find the optimal solution since traversing the entire search space to prove optimality is unmanageable. An approach that disregards this difficulty in constraint programming is known as Large Neighborhood Search (LNS). LNS does not prioritize completeness, where optimality can be proven, but focuses on rapidly finding an acceptable solution, which holds significance in real life scenarios. This is achieved by using parts of the previous solution found during the search to guide the next search towards a potentially promising area of the search space. While this strategy is effective, the guided search could be misled if the prior solution used bears no resemblance to the optimal solution. Hence the solutions found could be trapped in a local optima and have difficulties getting out of it. LNS works in two iterative phases, the relaxation phase and the reconstruction phase[8].

The relaxation phase starts by taking the current best solution and selectively resetting some parts to their initial state as defined in the original COP. The selection of destroyed parts is often decided by heuristics using some elements of randomness. Hence allowing for distinct relaxation phases originating from the same solution, in the case where the reconstruction phase does not find a better solution on the first try. Each relaxation phase yields a new COP which can be considerably smaller than the original COP when most parts of the solution are left untouched.

The reconstruction phase simply attempts to solve the new COP derived from the relaxation phase. Failures and solutions can be found faster with the reduced search space.

In the case where, the reconstruction phase is over and a new improved solution is found⁵, the iteration of relaxation-reconstruction goes on with this new solution instead. If no new solution was found, the next relaxation phase will use a different selection strategy on the same previous solution.

LNS Example: JobShop Problem

Continuing from the solution found in Figure 2.4.a, a large neighborhood search can be started. The only specification needed is what parts are reset in the relaxation phase and when to stop searching during the reconstruction. The example from

⁵Both need to be specified since the reconstruction phase may work until reaching a defined time or failure limit instead of a solution found. So the reconstruction phase can come across multiple solutions with the best one kept.

Figures 2.9 and 2.10 keeps around 60 % of the solution and stops the reconstruction search once the whole search tree has been traversed. Having only three tasks to move around, the search tree is considerably smaller and the full traversal of the tree becomes possible. With these parameters, a new solution of better quality was found and is in fact one of the best solutions so new iterations would not help further decrease the makespan, even though the algorithm does not know this.

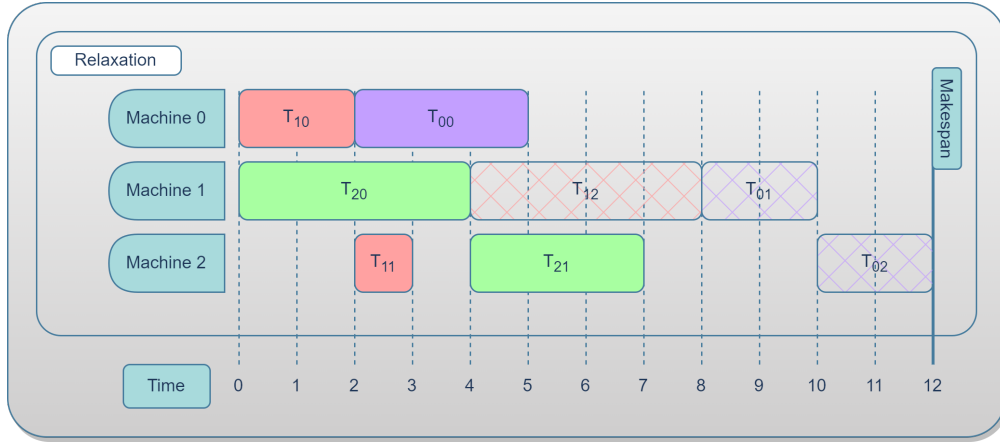


Figure 2.9: relaxation phase of LNS from the solution given in Figure 2.4.a.

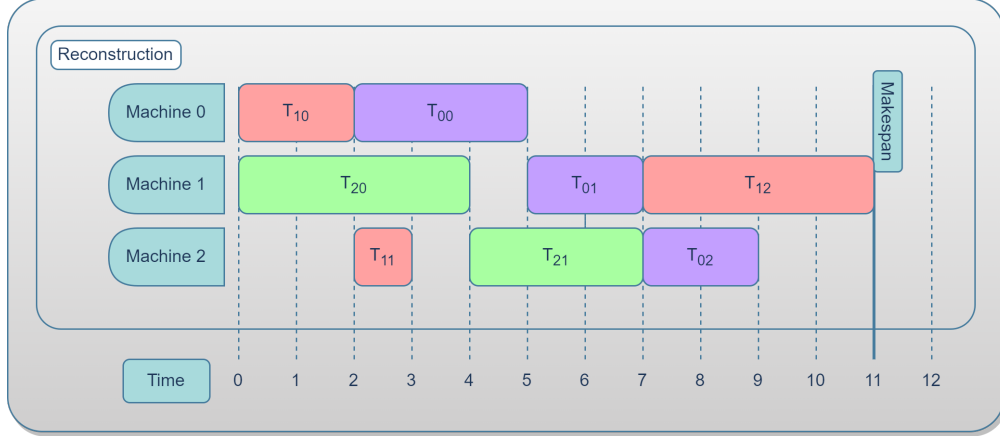


Figure 2.10: Reconstruction phase of LNS following the relaxation from Figure 2.9

2.4 Nogoods

While restarts are an effective mechanism to avoid heavy-tailed behavior by aiming to explore different regions of the search space, they lack the ability to incorporate learning and prevent revisiting these regions[12]. Consequently, systematicity, which ensures that no leaf node in the search tree is visited more than once, is not guaranteed[13].

One approach to tackle this challenge is to identify the decisions that led to such regions and avoid their repetition. This is where nogood recording comes into play. Initially introduced in 1977[14] and 1980[15] for solving constraint-based systems, its application to enhance CSP solving was proposed in 1990[16]. A nogood is the set of decisions leading to an inconsistency⁶ and is used by including a clause that prevents its reexploration during the subsequent restarts' searches. Originally, to ensure completeness and systematicity, the idea was to record a nogood for each encountered inconsistency and consider them during the restart. However, due to the exponential growth in the number of registered nogoods, an alternative idea, adopted in this thesis, involves extracting nogoods only from the last branch of the search tree, starting from the root, before each restart. This approach aims to block the same regions but with fewer nogoods and has been referred to by various names, including search signature[17], path recording[13], and restart nogoods[18, 12]. Only the latter will be used to describe the approach in this paper. Nevertheless, nogood recording also has a drawback: their recording and usage represent a significant additional workload. Therefore, it might be necessary to limit the length of registered nogoods to avoid overburdening the solver. Although it loses systematicity, it might be a needed trade-off in some cases.

This part will introduce some theoretical aspects of nogoods and address how these are created, stored and used in order to avoid reexploring some regions of the search tree in such a way that prevents overprocessing. Note that it is mainly based on the work of C. Lecoutre in *Constraint Networks: Techniques and Algorithms*[3].

2.4.1 Nogood Recording

To explain how nogoods are recorded from restarts, some assumptions must be done. Throughout this thesis, only backtrack search algorithms with binary branching will be considered. Additionally, only binary labeling, consisting of assignment and rejection decisions, is tackled in this section. An assignment is represented in

⁶Of course, for COP the concept is extended to all regions already visited including inconsistencies and already found solutions.

Now, generalized nogoods can be defined. Given a CSP P defined by $\langle X, D, C \rangle$ and Σ a sequence of decisions made along a branch of the search tree constructed for P , then for any nld-subsequence $\langle \delta_0, \dots, \delta_i \rangle$ of Σ , the set $\Delta = \{\delta_j \in \Sigma \mid 0 \leq j \leq i\} \cup \{\neg\delta_i\}$ is a generalized nogood of P , called nld-nogood. $\neg\delta_i$ denotes the positive decision opposite to the negative decision δ_i .

For example, Table 2.1 shows all the nld-nogoods that can be extracted from the last branch of Figure 2.11 resulting from corresponding nld-subsequences.

nld-subsequences	nld-nogoods
$\{a = 0, b \neq 1\}$	$\{a = 0, b = 1\}$
$\{a = 0, b \neq 1, d \neq 1\}$	$\{a = 0, b \neq 1, d = 1\}$
$\{a = 0, b \neq 1, d \neq 1, c = 3, b \neq 0\}$	$\{a = 0, b \neq 1, d \neq 1, c = 3, b = 0\}$
$\{a = 0, b \neq 1, d \neq 1, c = 3, b \neq 0, e \neq 1\}$	$\{a = 0, b \neq 1, d \neq 1, c = 3, b \neq 0, e = 1\}$

Table 2.1: Nld-subsequences and resulting nld-nogoods extracted from the last branch of Figure 2.11.

As it will be demonstrated later, the generalized nogoods are all managed together to prune the search tree, hence current nld-nogoods have redundancies. In fact, the negative decisions of a nld-nogood can be omitted since they are induced by their upper nogoods. Upper nogoods are the nogoods made from nld-subsequences of a same branch with a negative last decision closer to the root. Such nld-nogoods without negative decisions are called reduced nld-nogoods and can be defined as follows. Given a CSP P defined by $\langle X, D, C \rangle$ and Σ a sequence of decisions made along a branch of the search tree constructed for P , for any nld-subsequence $\Sigma_i = \langle \delta_0, \dots, \delta_i \rangle$ of Σ , the set $\Delta = \Sigma_i^+ \cup \{\neg\delta_i\}$ is a standard nogood of P , called reduced nld-nogood.

Table 2.2 extends the example of Table 2.1 by displaying the final reduced nld-nogoods.

nld-nogoods	reduced nld-nogoods
$\{a = 0, b = 1\}$	$\{a = 0, b = 1\}$
$\{a = 0, b \neq 1, d = 1\}$	$\{a = 0, d = 1\}$
$\{a = 0, b \neq 1, d \neq 1, c = 3, b = 0\}$	$\{a = 0, c = 3, b = 0\}$
$\{a = 0, b \neq 1, d \neq 1, c = 3, b \neq 0, e = 1\}$	$\{a = 0, c = 3, e = 1\}$

Table 2.2: Nld-nogoods and resulting reduced nld-nogoods coming from Table 2.1

The reduced nld-nogoods decrease the necessary storage space and improve pruning capability by shortening the nogoods. Since only reduced nld-nogood will be used in this paper, the term *nogoods*, if not specified, will refer to *reduced nld-nogoods* from now on.

2.4.2 Restarts with Learning

So far, the process of extracting nogoods from a branch of a search tree has been demonstrated through examples. This section will now outline the methodology for extracting them after each restart, thereby incorporating the desired learning to the restart process.

Algorithm 2.1 describes the nogood extraction process. It takes as input the sequence of decisions forming the last branch visited before a restart, i.e. the rightmost branch. The algorithm simply iterates on the sequence and performs two different actions given the type of decisions. If the current one is positive, the algorithm adds it in a temporary set of all positive decisions encountered until there. In the case where it is negative, a nogood is made from the union of the set of positive decisions and the opposite of the current decision. Then the nogood is added in a set to record it. This set is then returned at the end of the process.

Algorithm 2.1: Extracting nogoods

Input:

$$\Sigma = \langle \delta_0, \dots, \delta_{m-1} \rangle$$

begin

nogoods $\leftarrow \emptyset$

positiveDecisions $\leftarrow \emptyset$

for $i \leftarrow 0$ **to** $m - 1$ **do**

if δ_i **is positive** **then**

positiveDecisions $\leftarrow$ *positiveDecisions* $\cup \{\delta_i\}$

else

$\Delta \leftarrow$ *positiveDecisions* $\cup \{\neg\delta_i\}$

nogoods $\leftarrow$ *nogoods* $\cup \{\Delta\}$

end

end

end

return *nogoods*

The returned nogoods are then stored into a base with their specific usages detailed in the upcoming sections. For now, it can be seen as a set storing the nogoods generated after all previous restarts. The utilization of stored nogoods

prevents redundant explorations of the search space in different runs by ensuring that the set of decisions along the current branch is not covered by any previously recorded nogood. This is achieved by avoiding such sequence of decisions to be taken by making inferences. For example, if $\{a = 0, b = 1\}$ is a registered nogood in the base, and the decision $a = 0$ is taken in the current run of the search tree, the program can remove 1 from the domain of b ensuring that $b \neq 1$. This action is called an inference.

2.4.3 Nogood Constraints

This section will discuss how to manage nogoods as constraints. Nogoods have been created to indicate already explored states if the decisions constituting the nogoods are retaken. Therefore, if a constraint needs to be added so that it prohibits these reexplorations, it is necessary that the conjunctions made of the positive decisions are not met, i.e. that its opposite is satisfied. Given a nogood $\Delta = \{\delta_0, \dots, \delta_{m-1}\}$, the nogood constraint of Δ is defined by $c_\Delta : \neg(\delta_0 \wedge \dots \wedge \delta_{m-1})$.

Until now, nogoods have been seen as conjunctions of positive decisions. But those can also be modeled in strictly equivalent forms as disjunctions of negative decisions following the De Morgan's law⁸. Therefore, $c_\Delta : \neg(\delta_0 \wedge \dots \wedge \delta_{m-1})$ is strictly equivalent to $c_\Delta : \neg\delta_0 \vee \dots \vee \neg\delta_{m-1}$ and will be preferred to make later modeling easier. To illustrate this, the nogood $\Delta = \{a = 0, b = 1\}$ generates the nogood constraint $c_\Delta : a \neq 0 \vee b \neq 1$ implying that at least a must be different from 0 or b different from 1 to explore new states.

To use these new constraints efficiently, watched literals are implemented to enforce generalized arc consistency on nogood constraints. Since the method of watched literals, used in Boolean satisfiability problem (SAT) solvers[19], involves literals as indicated by its name, it will be addressed here as watched decisions. But some essential concepts still need to be defined first. The entailment of problems with nogoods, as well as the states of negative decisions will be discussed below.

Given a CSP $P = \langle X, D, C \rangle$ and Δ a nogood of P , P entails Δ iff $P|_\Delta = \langle X, D, C \cup \{\Delta\} \rangle = P$. In other words, adding the conjunction of the nogood's positive decisions, to the problem leaves it unchanged, as the variables concerned are already set to these particular values. Moreover, under the same assumptions, P can be said to almost entail Δ iff $\exists \delta_{(x,a)} \in \Delta$ s.t. $a \in D(x)$, $D(x) \neq \{a\}$ and P entails $\Delta \setminus \{\delta_{(x,a)}\}$.

⁸De Morgan's law states that $\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$ and $\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$ where P and Q are propositions. It is the second statement that is used here.

Watched decisions will serve as a pair of pointers to some decisions of the constraint nogood, allowing the decisions to be traversed and to determine if the problem (almost) entails the nogood or not. To this end, a negative decision $x \neq a$ of the constraint nogood c_Δ over a CSP $P = \langle X, D, C \rangle$ is said to be in one of the three following states:

- Satisfied iff $a \notin D(x)$
- Falsified iff $D(x) = \{a\}$
- Free iff $a \in D(x) \wedge |D(x)| > 1$

Based on the previous concepts, a constraint nogood can be categorized in four states that determine when actions can be taken, to prevent redundant explorations, and when it has become useless, meaning attention can be removed from it. Given a CSP $P = \langle X, D, C \rangle$, a nogood Δ and its constraint nogood c_Δ the four cases are the following:

- 1) c_Δ is entailed iff $\exists \delta_{(x,a)} \in c_\Delta$ s.t. $a \notin D(x)$ meaning that at least one decision of c_Δ is satisfied. P does not entail Δ .
- 2) c_Δ is disentailed iff $\forall \delta_{(x,a)} \in c_\Delta : D(x) = \{a\}$ meaning that all decisions of c_Δ are falsified. P entails Δ . This indicates that the algorithm needs to backtrack.
- 3) c_Δ has no satisfied decisions and exactly one free decision. P almost entails Δ . In this case, making an inference by satisfying the free decision would prevent a potential further backtracking.
- 4) c_Δ has no satisfied decisions and at least two free decisions. P does not entail nor almost entail Δ .

With these four cases, search algorithms will only need to prevent the second case by enforcing generalized arc consistency when the third case is reached. It happens when the penultimate free decision becomes falsified, i.e. when the variable of this decision becomes fixed by a positive decision or by propagation. To summarize, for nogood constraints to prevent reexploration, attention needs to be paid when fixes⁹ arise since they are the only actions that will possibly lead to inferences.

⁹Fixes denotes the actions that lead to a fixed variable.

2.4.4 Watched Decisions

Now that constraint nogoods have been detailed, a simple solution to make inferences could be to pay attention to all decisions of the stored constraint nogoods and take actions based on the fixes that occur. However, a more clever solution is able to drastically lower the computation for the same result. The idea is to only watch two decisions per nogood constraint to be able to identify when it falls into the case **3**). For example, such watched decisions are illustrated in Figure 2.12 where attention is paid to $a \neq 0$ and $d \neq 2$. In order to designate them easily, the first and second watched decisions are referred to as W_1 and W_2 respectively.

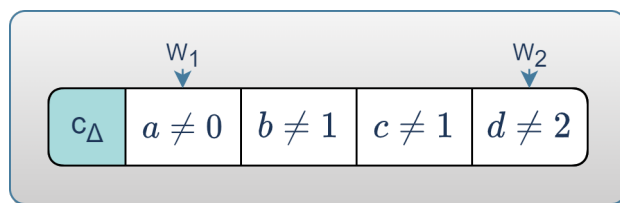


Figure 2.12: Example of initializing watched decisions W_1 and W_2 on a nogood constraint $a \neq 0 \vee b \neq 1 \vee c \neq 1 \vee d \neq 2$.

However, watched decisions on nogood constraints alone are not enough. They need an efficient data structure to be stored in, where they are accessible via a pair of variable and value. Hence, when a fix on variable x occurs, changing its domain to $D(x) = \{a\}$, all watched decisions $x \neq a$ can be accessed directly. This structure will be denoted as the base $\mathcal{B}$. It is composed of all the registered nogood constraints on the right side and of an array on the left. The array has an entry for each possible pair made of a variable and one of its initial domain value¹⁰. Each entry can either remain empty or serve as the head of a linked list of watched decisions having the same variable and value. The notation $\mathcal{B}_{(x,a)}$ will be used to indicate such a list for a pair of variable x and value a with each individual nodes pointing to a nogood constraint containing a decision $x \neq a$.

During a restart, the nogoods extracted from Algorithm 2.1 are stored in $\mathcal{B}$ as nogood constraints. For each added constraint, the first and last decision are chosen to be watched¹¹ and a node is added to the corresponding linked lists for each watched decision. Note that single nogoods, i.e. nogoods containing only one decision, that are extracted will not be added in $\mathcal{B}$ since they can be directly and permanently inferred. Consequently, at the beginning of the run following the single

¹⁰For example, given a CSP $P = \langle X, D, C \rangle$, the array has an entry for each pair of variable and value of the set $\{(x, a) \mid \forall x \in X, \forall a \in D(x)\}$.

¹¹Their location does not matter as long as it does not point to the same decision.

nogood inference, the nogood constraints falling into the case **1)** can be removed from $\mathcal{B}$ since they will no longer be of any help. Additionally, at that same moment, the falsified decisions can be removed from the nogood constraints since such decisions will never change their state again, and thus can no longer be free or inferred.

Figure 2.13 illustrates an example of a base $\mathcal{B}$ that just recorded $c_{\Delta_1} : x \neq 0 \vee y \neq 1 \vee z \neq 2$ and $c_{\Delta_2} : x \neq 0 \vee w \neq 1$ during a restart and that will be used for the next run.

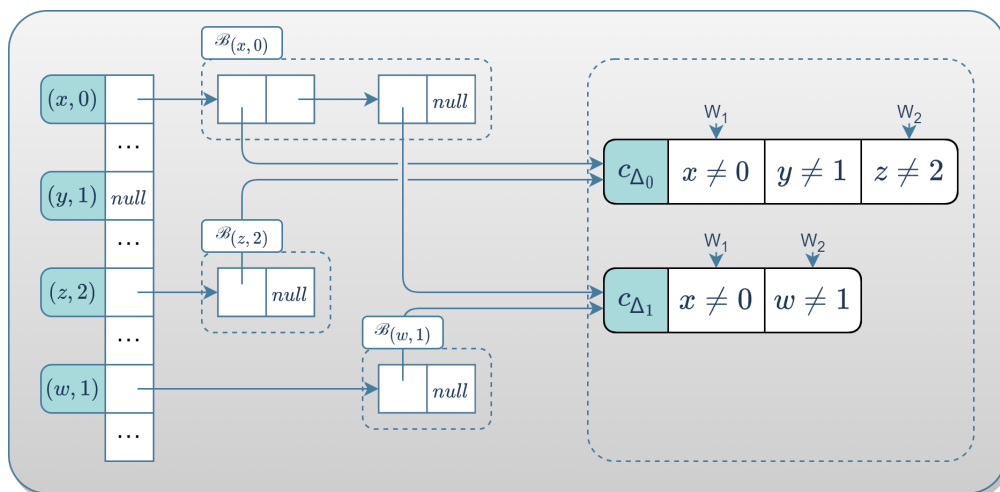


Figure 2.13: Representation of a nogood constraint base $\mathcal{B}$ storing c_{Δ_1} and c_{Δ_2} .

With this data structure, when a variable x is restrained to a sole value a , i.e. a fix occurs, the list $\mathcal{B}_{(x,a)}$ is accessed in constant time via the array. If it is empty, meaning there are no decisions $x \neq a$ being observed, it stops as there is nothing more to do. Otherwise, each of its nodes are visited to access the constraints with a watched decision $x \neq a$. Since $x \neq a$ became falsified, an action can be taken. First, it checks if the other watched decision is satisfied because nothing can be done in that case as the nogood constraint falls into the case **1)**. If not, the algorithm needs to establish if the case **3)** arises. To this end, it searches for another decision that is not falsified to which the pointer can be moved onto so that it becomes watched. When no other decision is found, the nogood constraint falls into the case **3)** and an inference is performed on the other watched decision to enforce generalized arc consistency. Note that since the decision on which the inference is performed could be falsified at the same time, the inference would lead to an empty domain requiring a backtrack¹².

¹²This is the explanation given by Lecoutre[3] although it is simply avoided in his implementation by checking this situation beforehand.

For the purpose of illustrating how the base works, Figure 2.14 continues the example from Figure 2.13 where the decision $z = 2$ is taken. When z becomes fixed, the algorithm accesses $\mathcal{B}_{(z,2)}$ which only points to c_{Δ_1} . There, since $z \neq 2$ is falsified and W_1 is not satisfied, it searches for another watchable decision in c_{Δ_1} and finds $y \neq 1$ which is still free. Hence, this decision becomes the new watched as W_2 . This is done by removing the node of $\mathcal{B}_{(z,2)}$ pointing to c_{Δ_1} and adding one to $\mathcal{B}_{(y,1)}$ so that the data structure remains valid. Here, $\mathcal{B}_{(z,2)}$ becomes empty.

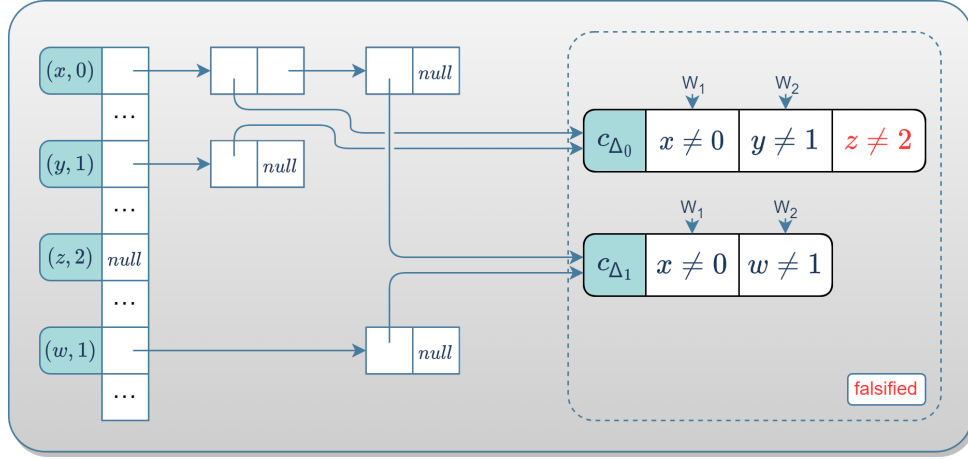


Figure 2.14: $\mathcal{B}$ of Figure 2.13 after taking the decision $z = 2$.

Afterward, Figure 2.15 shows the state of $\mathcal{B}$ after the branching on $x = 0$. Just as explained previously, it accessed the list associated with the entry $(x,0)$. The first node redirects to c_{Δ_1} where $x \neq 0$ becomes falsified with W_2 still free. It looks for another watchable decision but none are found. Thus, c_{Δ_1} falls in case **3**) and $y \neq 1$ can be inferred by removing 1 from $D(y)$. Finally, it goes on the second node of $\mathcal{B}_{(x,0)}$ and reaches c_{Δ_2} . Since W_2 is free and there are no other watchable decisions, it can directly infer W_2 by removing 1 from $D(w)$.

Finally, it is important to note that watched decisions keep their place after backtracking because they are still valid. Indeed, when needing to cancel the decision $x = 0$ upon finding inconsistencies further into the search following 2.15. The two W_1 will become free, as well as both W_2 that will no longer be inferred.

Algorithm 2.3: findWatchableDecision

Input: c_Δ // nogood constraint with at least one falsified
 // watched decision**begin****foreach** $\delta_{(z,c)} \in c_\Delta$ **do**
 if $\delta_{(z,c)}$ *is not watched in* c_Δ **then**
 if $\delta_{(z,c)}$ *is free or* $\delta_{(z,c)}$ *is satisfied* **then**
 return $\delta_{(z,c)}$
 end
 end**end****end****return** *null*

2.4.5 Minimizing Nogoods

As the type of nogoods is important here, the assumption that the term nogoods refers to reduced nld-nogoods is dropped for this section and all types will be explicitly mentioned.

Taking a step back, shorter nogoods are preferable as they are more suitable for inferences and require less computation. Therefore, it could be interesting to minimize them beyond the reduction of nld-nogoods. Minimizing a nogood involves attempting to reorder its decisions to determine if an inconsistency could occur earlier with fewer decisions. As a result, this approach is limited to nogoods ending with an inconsistent leaf node¹³, as the nogoods overlooking a subtree do not directly create an inconsistency with their last decision. This is because they share a common prefix with the nogoods that would be generated by all the leaf nodes within the subtree. In other words, the nogoods finishing by a leaf node causing an inconsistency do not respect the consistency.

Recall that consistency is a property that maintains coherence in a constraint network, a structure composed of variables and constraints. Lecoutre states that, given a constraint network P and any stable consistency ϕ , there exists a unique greatest constraint network where ϕ holds on P , denoted by $\phi(P)$, which is smaller than or equal to P and equivalent to P . This particular consistency is then considered well-behaved, and enforcing it on P means applying $\phi(P)$. To facilitate

¹³As the order of decisions in a nogood is not necessarily respected, the last decision of a nogood is a simplification to refer to the last decision of the nld-subsequence from which it originates.

the notation in the paper, the notion of consistency over a constraint network is enlarged to a CSP since it is defined by a set of constraints along with sets of variables and domains.

Now that ϕ is properly formulated, ϕ denotes the consistency involved in our problems, and consequently the nogoods breaking the consistency ϕ will be called ϕ -nogoods. Those nogoods are defined as followed. Given a CSP P defined by $\langle X, D, C \rangle$, a well-behaved consistency ϕ , and Δ a set of decisions on P , Δ is a ϕ -nogood of P iff $\phi(P|_{\Delta}) = \perp$. Also Δ is said to be a minimal ϕ -nogood of P iff $\nexists \Delta' \subset \Delta$ such that $\phi(P|_{\Delta'}) = \perp$.

It is important to note that when obtaining a reduced nld-nogood from a nld-nogood that is a ϕ -nogood, the reduced one may not be a ϕ -nogood itself. This is because the negative decisions removed during the reduction process may play a crucial role in the inconsistency¹⁴.

The idea in the extraction of nogoods is then to keep non ϕ -nogoods as reduced nld-nogoods and attempt to minimize the reduced ϕ -nogoods that are ϕ -nogoods as well. A nogood can be identified as a ϕ -nogood if it contains a transition decision. Given a CSP P and a ϕ -nogood $\Delta = \{\delta_0, \dots, \delta_{m-1}\}$ with an ordering given by indices, a transition decision is a decision $\delta_i \in \Delta$ such that $\phi(P|_{\{\delta_0, \dots, \delta_{i-1}\}}) \neq \perp$ and $\phi(P|_{\{\delta_0, \dots, \delta_i\}}) = \perp$.

To minimize a ϕ -nogood, an iterative process is followed. First, the decisions are reordered, with the transition decision δ_i placed in the first position, and decisions δ_j with $j > i$ are disregarded. This results in a modified nogood $\Delta' = \{\delta_i, \delta_0, \dots, \delta_{i-1}\}$. Then, the process is repeated, each new transition decision being placed after the one from the previous iteration. The process continues until the ϕ -nogood is made up only of past transition decisions. In such a case, it is referred to as a minimal ϕ -nogood, denoted as Δ^m .

In conclusion, while minimization can lead to smaller nogoods in some cases, it comes at the cost of extensive computation. Therefore, the trade-off between increased computation for smaller nogoods may not always be beneficial. Since its practicality is not yet firmly established, this paper will not make use of minimal ϕ -nogoods. Although not explored further here, the theoretical aspects introduced in this section will be relevant for the next chapter.

¹⁴As a result, another approach would be to minimize nld-nogoods instead of reduced nld-nogoods but it would require to manage generalized nogoods instead of standard nogoods in constraints, which is not addressed in this paper.

2.5 MiniCP

MiniCP[1] is a lightweight open-source constraint programming solver developed for educational purposes. It comes from the observation that many undergraduate students in computer science are not familiar with CP. The root of the problem is that it is simply not taught in universities and that the current solvers are focused on speed and functionalities at the expense of accessibility. MiniCP instead prioritizes simplicity, clarity, and ease of learning making it suitable for beginners. MiniCP does not explore all dimensions of constraint programming, but thanks to a minimalist and thoroughly documented architecture, it is made to be extendable. The purpose of the paper is in fact to explore one of these aspects which is the learning-based CP solver.

This section describes how the previous theoretical aspects are implemented, allowing MiniCP to model and solve CSP or COP. It includes how backtracking is managed, as well as how variables and their domains are represented with efficient objects to avoid burdening the solver. Furthermore, it describes how constraint propagation is handled, along with how the search and branching are performed to solve a problem. Ultimately, the way problems are modeled practically in MiniCP is examined. Along with the explanations, the names of the main classes and essential functions are introduced for a deeper understanding of the architecture. These will be useful to explain the additions in the next chapter.

2.5.1 State variables

State variables are restorable objects that can recover their previous values upon modifications. Several of these types of objects are implemented in MiniCP but, as the logic remains the same, only the `StateInt` object representing stateful integers will be detailed. The `StateManager` is the key factor behind this stateful property. Besides creating the objects, it performs saves and restores of those integers by using a stack that records each save. Consequently, each save is assigned to a position in the stack denoted as the level of the state manager. At first, the level starts at 0 and increases by 1 for each backup added. To restore a state backed up at a given level l , the `StateManager` pops every state from levels greater than l , ultimately forgetting them.

Saves and restores are essential to model a DFS. If a state manager handles a search, a save can be established at each node, which is its own state, so that the LIFO¹⁵ property of the stack follows the search synchronously. This allows for chronological backtracking since, in the event of an inconsistency, the most recent

¹⁵Last in First Out refers to the order in which items are added and removed from a data

decision can be canceled by restoring the state of the parent node. `saveState` and `restoreState` are the methods that perform these so-called operations in MiniCP.

In MiniCP, such state saves can be made in two ways. The simplest one is to copy the entire state storing the values of all state variables in each save. This approach is named the copying state management strategy and is carried out by the class `Copier`. Of course, this method is not optimal since it is more likely that only a few of the variables will be modified via branching. To address this issue, the second method, called the trailing state management strategy, lazily copies states such that only modified state variables are recorded. This strategy, implemented by the `Trailer` class, will be the only one used to make benchmarks in Chapter 4.

2.5.2 Domains

In MiniCP, domains are implemented with sparse sets which are data structures that allow to store subsets of numbers between a lower bound v_0 and an upper bound v_{n-1} . A sparse set is preferred over a classical set because it enables efficient removal of elements in the domain while staying backtrackable. This efficiency remains even with big gaps in the domain, hence why it is named sparse set. It is composed of two arrays, one of which is a permutation of the values forming the domain and is denoted by V . The second one is denoted by I and monitors the positions of values in V . For any domain operations, these two arrays follow the invariant $\forall v \in D(x) = \{v_0, \dots, v_{n-1}\} : V[I[v - v_0]] = v$ for a variable x ¹⁶. Being the lower bound, v_0 is used as an offset to keep I zero based indexed¹⁷. In addition to these arrays, the sparse set maintains a variable *size* updated to represent the current size of the domain. The domain operations introduced above are the following:

- *contains*(v) : checks if the value v is still part of the domain by verifying that $I[v - v_0] < \text{size}$. It is performed in $O(1)$.
- *remove*(v): takes away the value v of the domain by swapping the value v and w respectively at index $I[v]$ and $\text{size} - 1$ in V and adapts I by swapping $I[v]$ and $I[w]$. Then *size* is decremented by 1. It is performed in $O(1)$.
- *removeBelow*(v)¹⁸: removes values smaller than v from the domain by executing *remove*(w) for each value w of the domain such that $w < v$. For R

structure.

¹⁶ $A[i] = v$ is the notation to denote the access of i^{th} element of array A which has value v .

¹⁷In MiniCP, V is also zero based indexed so the invariant becomes $\forall v \in D(x) = \{v_0, \dots, v_{n-1}\} : V[I[v - v_0]] = v - v_0$.

¹⁸*removeAbove*(v) can be deduced from *removeBelow*(v).

being the set of removed values, it is performed in $O(|R|)$.

- *removeAllBut(v)*: equivalent to fixing the domain to v , it removes all other values. To achieve this, it swaps the values v and w respectively at index $I[v]$ and 0 in V and adapts I by swapping $I[v]$ and $I[w]$. Then it sets *size* to 1. It is performed in $O(1)$.
- *removeAll()*: empties the domain by setting *size* to 0. It is performed in $O(1)$. This operation raises an inconsistency.

No operations consisting in adding values to the domain were shown above as it does not happen directly in a search tree since the branching can only, at most, remove values from variables. However, when backtracking, the impact of canceled decisions need to be forgotten as well, thus adding the values that were previously withdrawn by these decisions. With the sparse property, it can simply be achieved by resetting the *size* to its previous value. Hence, *size* is a **StateInt** object. The class implementing this data structure is **StateSparseSet**.

Figure 2.16 illustrates the previously explained operations on a state sparse set representing the domain $\{0, 1, 2, 3, 4\}$.

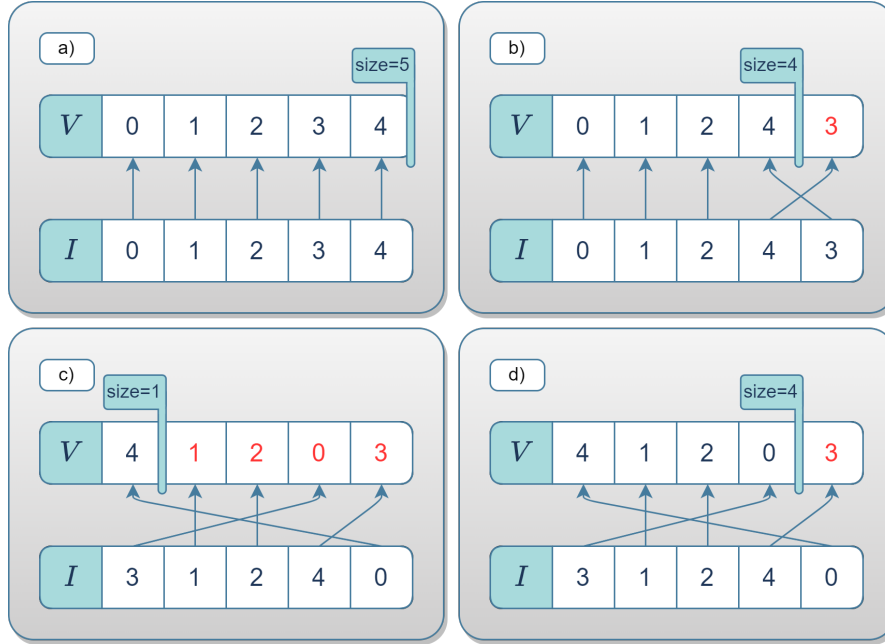


Figure 2.16: Different states of a sparse set representing domain $\{0, 1, 2, 3, 4\}$ after operations *remove(3)*, *removeAllBut(4)* and a backtrack on the previous operation.

Subfigure **a)** introduces the initial object with a complete domain. Then, the domain parts with the value 3 in **b)** and ends up being fixed to 4 by using a *removeAllBut*(4) in **c)**. Finally, **d)** displays the object after the fix is canceled, and the variable *size* is restored to its previous value.

2.5.3 Variables

Continuing to focus on integer domains, variables with such domains are represented with the class `IntVarImpl` which implements the interface `IntVar`. The class contains a reference to the solver, a domain and three stacks¹⁹ of constraints which involve the variable. Each stack handles a particular propagation situation. These situations are when the domain is modified, when at least one of the bound of the domain is modified and when the domain is fixed. Consequently, the stacks are named `onDomain`, `onBound`, `onFix` and serve to easily schedule constraints for propagation.

Therefore, to link a constraint and a variable together based on an event, the constraint needs to be added in the according stack. In addition, the stacks are not limited to problem constraints. They can also contain procedures, in a constraint form, that should be called when the event occurs. The methods allowing this are `whenDomainChange`, `whenBoundChange` and `whenFixed`. Lastly, since the class represents a variable, it has the domain operations discussed above. However, the removal operations are delegated to the domain objects with a twist: a domain listener is added with each operation. `DomainListener` is an inner class of `IntVarImpl` containing relay functions that notify the upper class on the impact of the domain modification. The operations are thus modified to include some checks and whether they lead to a domain change, a bound change, a fix, or an empty domain. Then the domain listener either schedules the constraints of the appropriate stack(s) or raises an inconsistency by throwing an `InconsistencyException`.

2.5.4 Constraints and Propagation

In MiniCP, `Constraint` is an interface implemented by the `AbstractConstraint` class. This abstract class enables specific constraints to extend it using dedicated classes, thus inheriting common functions. Among the common functions, `post` and `propagate` need to be overridden by subclasses because they heavily rely on the type of constraint. Except for these functions, the others are methods that increase the efficiency of the fixpoint algorithm²⁰ by adding two properties.

¹⁹These stacks are `StateStack` objects that are actually implemented with an `ArrayList` behind, which is why iteration is no problem.

²⁰The fixpoint algorithm is implemented in the function `fixpoint` of `MiniCP` class.

The first is that constraints can be disabled via a stateful Boolean. When the constraint is verified, no more propagation is needed since it would be useless.

The second property arises from the way constraints are propagated. When a constraint needs to be propagated, which comes from a domain modification, it is added to a propagation queue as if it was scheduled. Once evaluated by the fixpoint algorithm, this constraint can itself add other constraints to the queue. Putting things into context, when a decision is taken during the search, it is posted as a trivial constraint enforcing the decision. However whenever a constraint is posted, the fixpoint algorithm is executed, causing propagation. This algorithm iterates over the queue of triggered constraints and propagates each of them. It terminates either if the queue is empty signifying the end of propagation or if an `InconsistencyException` is caught. In that case, the queue is first emptied and the exception is raised. With this configuration, some constraints could appear multiple times in the queue, which is useless and therefore not wanted. Thus, the second property added by the functions of `AbstractConstraint` is the management of a flag `scheduled` indicating if the constraint is already in the queue.

2.5.5 Branching

MiniCP takes advantage of closures introduced in Java 1.8[20] which is essential for functional programming to model the branching. A branch corresponds to a closure²¹ posting a constraint. Since it is encapsulated, the closure is only executed when the branch is visited during the search. `Procedure`²² is the functional interface storing that kind of closure with the method `call`. A branching scheme, generating the branches from a node, is then a supplier of `Procedure[]`²³, ordering them according to the variable and value ordering strategies. A trivial example of such a strategy could be to always branch on the variable with the smallest domain which is not fixed, and select its minimal value. Such strategies are made up of variable and value selectors.

2.5.6 Search

To solve a problem in MiniCP, different initial functions are called depending on the kind of problem. If it is a CSP, `solve` will be the method executed whereas if it is a COP, it will be `optimize`. The last includes an objective to optimize that is tightened each time a new solution is found. Since the goal is to find

²¹A closure is a first order function, i.e. a lambda function.

²²A functional interface is an interface having only one abstract method.

²³This is achieved with `Supplier<Procedure[]>` where `Supplier` is a functional interface with an abstract method `get`.

the best solution, only new solutions better than the best found so far will be relevant. MiniCP has implemented a way to be notified of such solutions, allowing to use them in a LNS for example. Also, it is worth mentioning that `optimize` is implemented by calling `solve` internally. This is possible because MiniCP does not terminate the search when a solution is found for a CSP, or at least if it has not been informed to do so. This allows `solve` to explore the entire search tree and record all the solutions. In addition, both methods can include a stopping criterion informing the search to stop in a particular situation. For example, this can be when a certain number of inconsistencies is reached, or a time limit has elapsed, or again when a number of solutions is found²⁴. To this end, some statistics are recorded for a stopping criterion to be able to take action, such as counting the number of failures, i.e. the number of inconsistencies.

In addition to this, these methods can also be subject to a closure to be executed before the start of the search. In that case, the methods are `solveSubjectTo` and `optimizeSubjectTo`. This closure can be used to model a LNS for example. In that case, a loop iterates over a number of restarts where the problem is implemented and execute `optimizeSubjectTo` for each repetition by fixing some parts of the variables based on the best solution found so far in the closure.

All these methods are defined in the `DFSearch` class. The class also includes the `dfs` method, performing a DFS, which is called during the execution of `solve`. Examining its implementation reveals that it creates an initial state, which forms the root state of the search, before executing `dfs` in a `try` statement. If the entire search tree is explored, the method ends normally and if the stopping criterion is reached, the `catch` statement neutralizes the `StopSearchException` before finishing the method.

The method `dfs` can now be developed further. It possesses a stack that is supplied with closures that either save a state, restore a state, or post a constraint. This is done via the method `expandNode` which generates closures that post constraints from a branching scheme based on the given state of the node. They are then added in the stack with each being surrounded between a restore state and a save state closure. The order is backward so that when popping the stack, the saves appear before posting the constraints and the restores after. Also, the branches include a recursive call to `expandNode` according to the DFS approach. If the branching scheme does not provide any branches, it indicates that the node is a solution since all variables are fixed. In that case, the solver is informed after updating the

²⁴This implies that a solver solving a CSP can still be stopped after a solution is found if the stopping criterion says so.

statistics, allowing a stopping criterion based on statistics to operate accordingly. When `dfs` has updated its stack, it then iterates over it. For each popped element, it checks the stopping criterion and stops if it is reached. Otherwise, it tries to successfully execute the closure. If an `InconsistencyException` is raised, it is caught, the statistics are updated in consequence and the solver is notified. Else, the algorithm continues.

2.5.7 Practical Problem Modeling

The final thing to expand on is how problems are modeled in practice. All previous classes and functions are essential to run the constraint propagation solver. However, they are not all involved in problems modeling. Remembering the Sections 2.1.1 and 2.1.2, a CSP is defined by $\langle X, D, C \rangle$ and a COP is defined by $\langle X, D, C, f \rangle$. Therefore, only these components need to be declared for modeling.

Factory is a class simplifying this process. Indeed, it serves as a factory that creates the solver, variables with domains, constraints, and search instances respectively having classes `MiniCP`, `IntVar`, `Constraint` and `DFSearch` while making the necessary links between them²⁵.

Factory achieves this mostly by itself since it creates these other classes. Some have the prefix `make` and are followed by the name of the created objects, while others are simply named from the basic constraints they create. As a result, there are `makeSolver`, `makeIntVar`, `makeDfs`, as well as `equal`, `notEqual`. These last functions return `AbstractConstraint` objects where the `post` method is overridden to perform the straightforward operation with a domain operation.

Listing 2.1 shows how the methods of **Factory** intervene in the modeling of N-Queens problem in `MiniCP`. There are only three things that have not been discussed yet:

- `Solver` is the interface implemented by `MiniCP` and has a `post` method which serves as a relay to call the constraint `post` function and launches `fixpoint`.
- `makeIntVarArray` is a shortcut method to define an array of similar `IntVarImpl` objects. Here, the first n precises the size of the array, and the second n defines the size of domain variables.
- `onSolution` is a method allowing to define a closure to be executed when a solution is found during the search.

²⁵Objective functions are not provided by **Factory** but by `MiniCP`.

Considering all the discussed aspects so far, implementing the problem becomes straightforward, thanks to the declarative nature of constraint programming. The various steps involved can clearly be observed, such as creating the solver, declaring variables with domains, adding constraints to the problem, specifying the action on solutions, and initiating the search. The only slightly less trivial part here is the declaration of constraints. It is important to remember that each `IntVarImpl` represents a queen on a different column, while the domain variable represents the row indices. As a result, the first constraint ensures that two queens are not placed on the same row, while the other constraints prevent them from sharing a diagonal.

```
Solver cp = Factory.makeSolver(false);
IntVar[] q = Factory.makeIntVarArray(cp, n, n);
for (int i = 0; i < n; i++){
    for (int j = i + 1; j < n; j++) {
        cp.post(Factory.notEqual(q[i], q[j]));
        cp.post(Factory.notEqual(q[i], q[j], j - i));
        cp.post(Factory.notEqual(q[i], q[j], i - j));
    }
}
search.onSolution(() ->
    System.out.println("solution:" + Arrays.toString(q))
);
DFSearch search = Factory.makeDfs(cp, firstFail(q));
SearchStatistics stats = search.solve();
```

Listing 2.1: N-Queens problem modeled with **Factory** methods.

Chapter 3

Integration of Nogoods in MiniCP

Now that the necessary theoretical background has been covered, it is time to look at how the nogoods²⁶ are integrated into MiniCP. This chapter describes step by step what had to be added in order to achieve this. Section 3.1 explains how the decisions are recorded, Sections 3.2 and 3.3 how the nogoods are created and stored, and Section 3.4 how they are used to make inferences. For each part, some explanations will outline implemented classes and methods as well as the implementation choices. Moreover, since the implementation is not generic, the supported branchings for nogoods management in MiniCP will be examined in Section 3.5. Then, Section 3.6 will detail the way in which nogoods have been integrated to also enable LNS. Lastly, since the integration has been performed in order to minimize the change in the user's experience of MiniCP, Section 3.7 will discuss the use of nogoods to solve problems in this solver. This goes hand in hand with an explanation on how all separated parts interact with each other during the resolution of a problem.

3.1 Decision Path Saving

As described in Section 2.4.1, nogoods are composed of the decisions taken by the last branch of the searched tree before a restart. Therefore, it is necessary to record these decisions made by the DFS in MiniCP. Saving them throughout the algorithm allows to create the nogoods after a restart by simply iterating over the last decisions. So, it is crucial that this part of the code is well implemented to avoid potential slowdowns in the algorithm.

²⁶As explained in Section 2.4.1, nogoods refer to reduced nld-nogoods.

3.1.1 Decision

To achieve this, the first step is to be able to store a decision. A decision taken during the search consists of posting a constraint made out of a variable, a value, and an action²⁷. However, the specific action, represented by the constraint itself, is not what needs to be known in a nogood since the implementation uses different classes for the different decision branchings. Indeed, it is rather the type of the decision that is important to save in order to know if the binary search tree branched on the left or the right.

To make it possible to save the essential information, a new object named **Decision** has been implemented. The class specification is shown in Listing 3.1.

```
public class Decision {
    IntVar x;          //The variable's identifier
    int v;             //The value concerned
    boolean assign;    //The type of decision
    ...
}
```

Listing 3.1: **Decision** class.

In the following sections, Lecoutre's wording introduced in Section 2.4.1 concerning the type of decisions is used. Consequently, a decision stored as an object **Decision** having an **assign** attribute set to **True** will be referred to as a positive decision whereas if set to **False**, it will be called a negative decision.

3.1.2 Path saving

With the capability to store a decision, it is now possible to save the path of a branch in the search tree. For that, each time a decision is taken in the search tree, it is added to an **ArrayList** as a **Decision** object.

The **ArrayList**, while dynamically increasing its size, allows to reset elements at specified indexes and to add elements at the end in $O(1)$ ²⁸. These properties are interesting because they eliminate the need for a stack, as decisions can be stored without the use of push and pop operations when backtracking. Actually, outdated decisions will never be removed upon backtracking but will remain in memory. These decisions will eventually be lazily changed when a new decision

²⁷Section 3.5 will describe the supported branchings implemented.

²⁸It is the complexity in average time. The worst case is in $O(n)$ when the list has to be reallocated due to its size.

overrides its place in memory during the search.

The key is then to know at which position the decision should be stored, which can be easily obtained from the level of the **StateManager**. The **StateManager** stores the states of the objects it manages at each node in the tree and increments a level variable accordingly. Conversely, it decrements the level when backtracking in order to restore a previously saved state. To summarize, the **StateManager** can indicate the depth of the search, which is used as the index to fill the corresponding decision in the path saving list. This eliminates the need for unnecessary deletion of list elements.

To better understand how the implementation works, Figure 3.1 is an example that illustrates how the decision path, made from the list named **Decisions** paired with the level, is updated after backtracking and taking a decision.

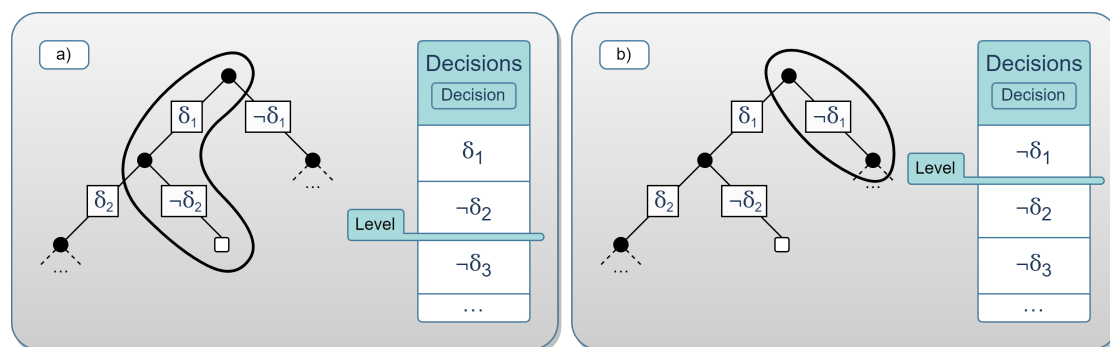


Figure 3.1: Example on saving the decision path as **Decisions**

3.2 Creating the Nogoods

With the help of Section 3.1, a list maintains a record of decisions made up to the current branch of the search tree, which is continuously updated. This will be useful to generate nogoods at the end of each restart of the search tree. Hence, the following section builds upon Lecoutre's approach of nogood creation, as described in 2.4.2. Furthermore, it introduces the possibility of defining a length threshold.

At the start of a search, a stopping criterion is specified, which may be based on a cutoff value such as the number of failures or a time limit. When the criterion is reached, the search performs a restart. Prior to the start of a new search, a function is called to generate nogoods based on the current branch of the search tree. This branch corresponds to the last branch for this iteration and is stored in **Decisions**. In addition to the list, the function needs the depth level reached so that outdated

decisions saved further into the list will not be taken into account. Since multiple nogoods can be produced from it, they are generated by increasing size starting from the root of the tree. Algorithm 3.1 describes the strategy applied to produce them. Note that the red section is an added feature that will be explained at the end of this section.

Algorithm 3.1: Generate nogoods from Decisions

```

Input:
    decisions           // list of Decision of last branch
    depth               // level before restart; root is zero
    threshold           // maximum length threshold

begin
    last  $\leftarrow$  depth - 1           // index of last negative decision
    for last to 0 do
        if decisions[last].assign = False then
            break
        end
    end
    nogood  $\leftarrow$  List()
    for i  $\leftarrow$  0 to last do
        if decisions[i].assign = True then
            nogood.add(decisions[i])
            if size(nogood)  $\geq$  threshold then
                return
            end
        else
            new  $\leftarrow$  copy(nogood)29
            new.add(decisions[i])
            storeNogood(new)
        end
    end
end

```

Since the type of stopping criterion is unknown, the final state reached at the end of a restart might lead to solutions. Therefore, no nogood that blocks this particular state in the tree should be generated, as it has not been fully explored yet. Consequently, any trailing positive decisions are disregarded as they cannot

²⁹When $i = \text{last}$, execution of *copy* is not necessary and one can add *nogood* instead of *new*. Obviously, this kind of minor optimization is carried out in the implementation but does not seem adapted to the pseudo algorithms discussed.

be used to form a valid nogood. The first loop achieves this by finding the index of the last negative decision.

Afterward, the second loop iterates until it reaches the index found in the previous loop. Each positive decision is added to a common base named `nogood` which will remain in nogoods of greater size. On the other hand, the negative decisions are not added to the list since the algorithm generates reduced nld-nogoods. Instead, a negative decision will only be added to a copied version of the list so its nogood can be stored via the function `storeNogood`, which sends the nogood to be stored in a database. Making nogoods out of positive and a final negative decision is not a problem because the `assign` attribute of the decision is no longer required for the rest of the implementation as the nogood is already reduced. Additionally, this is why the algorithm generates nogoods and not constraint nogoods to be stored in the base as opposed to Lecoutre’s approach seen in Section 2.4.2.

While nogoods have the potential to reduce the search space, they can also become quite large, which can lead to some drawbacks. Indeed, the longer the nogood, the longer the process to make an inference. In the best case, it results in a low gain for a heavy calculation and in the worst case, in wasted time. As a result, their usefulness can be limited. To address this issue, a maximum length threshold parameter has been introduced, which allows the user to set a limit on the size of the generated nogoods. To accomplish this, the red part of Algorithm 3.1 comes into action.

As you may have noticed, Algorithm 3.1 leaves the possibility of revisiting a previously made decision path from a previous restart due to the unknown stopping criterion. At first, it could seem to contradict the very concept of nogoods. However, it only happens at the end of the restarts decision path, either being the last state after a negative decision or multiple states from the last positive decisions. So, even if the stopping criterion used was a failure limit, meaning there was nothing after the final node, the nogood not generated would be the longest one out of the given path. Since the longest nogoods have a lower probability of making inferences, this drawback should not have a significant impact. Nevertheless, a variation of Algorithm 3.1 with such a stopping criterion is in Appendix A and avoids the mentioned issue by never returning to a previously explored state.

3.3 Storing the Nogoods

Once nogoods are generated, they need to be stored in an efficient manner to minimize computing time when using them as a general constraint. What follows is the implementation of Lecoutre’s constraint base, as explained in Section 2.4.4, which has been adapted for use in MiniCP. This section consists of explaining each component in the base: the variable base, the watched literals and nogoods base. It thoroughly examines the implementation decisions made to enable the base’s functionality in MiniCP. Figure 3.2 provides an example of a functional base with the addition of several nogoods and watched decisions.

It is important to note that the following implementation is designed to remain unaffected by restarts. Therefore, a restart will not prompt the recalculation of any component of the base, as it remains static and beneficial for all subsequent restarts. The only modifications to the base arise from decisions that require the update of watched decisions and the addition of new nogoods.

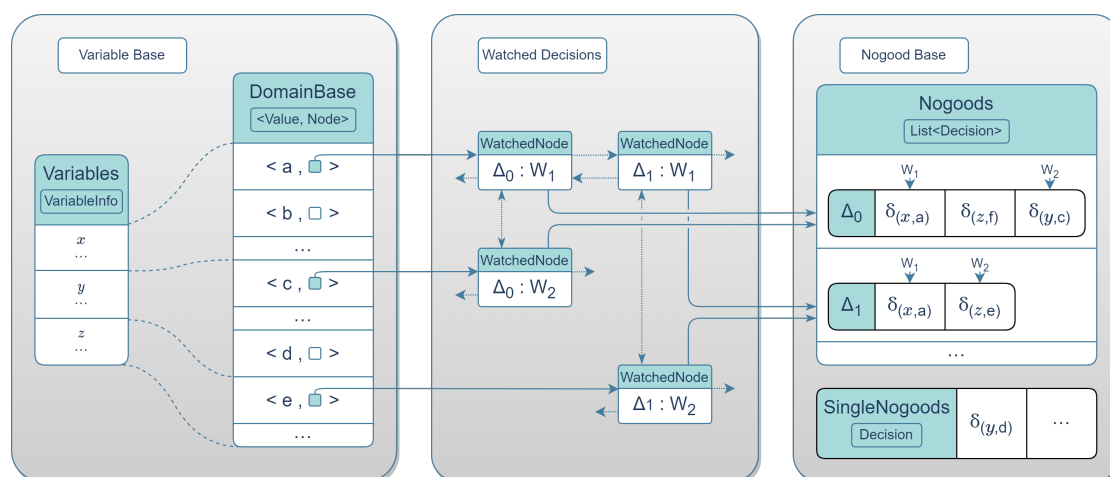


Figure 3.2: Nogood constraint base in MiniCP

3.3.1 Nogood Base

Each generated nogood, represented as a list of **Decision**, is added to an **ArrayList** via the function **storeNogood** used in Algorithm 3.1. Once it is added to that list, the index of the nogoods position becomes its unique identifier for use as a reference by the watched decisions. This is the most straightforward component of the base since the nogoods have already undergone reduction and filtering using a max length as stated earlier.

A specific case arises when a nogood consists of only one decision. In such cases, the notion of watched decisions becomes redundant as the decision in that nogood directly leads to an inference. Since these single nogoods are distinct from longer ones, their decisions are stored in a list for future reuse during the inference step.

3.3.2 Watched Decisions

Upon adding a nogood to the `ArrayList`, two watched decisions are created for the new entry. To organize and store all of these watched decisions in an orderly manner, along with additional information for enhanced optimizations, a new `WatchedNode` object is used. Its class specification is shown in Listing 3.2.

```
public class WatchedNode {  
  
    int nogoodIndex;           // The index of the nogood  
    int decisionIndex;        // The index of the decision  
    int direction;            // The direction of the update  
  
    WatchedNode previous;     // The previous node  
    WatchedNode next;         // The next node  
    WatchedNode otherWatched; // The other watched decision for  
    ...                        // this nogood  
}
```

Listing 3.2: `WatchedNode` class.

The `WatchedNode` object is implemented to construct double-linked lists of watched decisions, which will henceforth be referred to as nodes in the subsequent section. The choice of a double-linked list allows for an easy removal or addition of a node when traversing it during the inference step. As a reminder from the previous chapter, each linked list corresponds to a specific decision with its head stored in the `DomainBase` for fast access.

Watched decisions are created in pairs for each nogood addition. Both decisions store the index of the new nogood and an index for the decision they handle. Initially, the decision index is set to the first and second elements in the list. As the watched decisions are subject to change throughout the restarts, the starting point is not crucial, but the direction is critical when searching for a new decision to watch. Consequently, to minimize collisions during updates, for each pair of watched decisions, one is traversing to the left through the nogood decisions, and the other is traversing to the right. The presence of the `direction` attribute in the node object facilitates this feature.

The final variable in the `WatchedNode` object that has yet to be clarified is `otherWatched`. This variable links pairs of nodes that are watching the same nogood, enabling them to locate each other quickly. It improves efficiency when updating a watched decision, as its paired node may already possess information to halt the update. The usage of the `otherWatched` variable will be demonstrated when making inferences in Section 3.4.

3.3.3 Variable Base

To store the linked lists of watched decisions, a fixed size `ArrayList` named `DomainBase` is used. Its structure is as follows: each element corresponds to a possible decision, storing a pair of value and a link to the corresponding linked list's head. Several implementation choices have been made to minimize the size and optimize the lookup time of the `DomainBase`, which will be detailed below.

The first implementation that allows for these two improvements involves using a second list in conjunction with the `DomainBase`. This list, named `Variables`, contains additional information on each variable which is essential for rapidly locating the appropriate index for a specific decision in the `DomainBase`. To handle the extra information, a new `VariableInfo` object is implemented, and its class specification is in Listing 3.3.

```
public class VariableInfo {  
  
    IntVar variable; // The variable  
  
    int initSize; // The variable's domain size before the search  
    int firstIndex; // The first index given in the DomainBase  
  
    int minValue; // The lowest domain value before the search  
    int maxValue; // The highest domain value before the search  
    ...  
}
```

Listing 3.3: `VariableInfo` class.

Initially, the new object may seem excessive just for locating the appropriate decision index in the `DomainBase`. However, there is a reason for this implementation choice. Not every decision of every variable requires saving, as some are guaranteed never to occur. This is due to the fact that initial constraints can decrease the size of a variable's domain prior to the search. Thus, there is a `findPosition` function that uses the `VariableInfo` object to efficiently find the right index.

The `findPosition` works in 3 steps, it first makes two informed guesses using the information stored, and if none of those are the right index, it performs a binary search between the two informed guesses.

Since most variable domains can be reduced prior to starting the DFS, the user must manually add the variables used for the search to the `Variables` list when implementing their constraint programming problem. The implementation ensures that the `DomainBase` is only created from the added variables domains once the search is initiated. This feature allows the constraints to be posted at any time, before or after specifying which variables the nogoods will operate on. Since the constraints may reduce the domain of some variables, the `DomainBase` will store fewer pairs and, therefore, have the smallest possible size as soon as the search begins. Larger constraint programming problems can be particularly affected since the variable domains may increase significantly.

As an alternative solution, a hashmap was used to replace the whole variable base by mapping hashed decisions to the list of their `WatchedNode`. Besides clearly simplifying the code, it also featured a lazy addition of decisions in the variable base to minimize memory. The reason it is not used in this implementation is because, in some cases, multiple consecutive elements of the `DomainBase` need to be traversed. Meaning that having an index iterating over the list would be preferred.

The addition of notifiers is the final element that makes the entire base functional. MiniCP offers a function that allows any procedure to be executed when a variable's domain is modified. Consequently, when a variable is included to the nogood variables, a notifier is added for that particular variable domain change, which includes a procedure to make inferences under the appropriate circumstances.

3.3.4 Example

The initial step is to start the variable base. First the variables are added, then constraints may reduce their domains, and when the search is finally started the `Variables` becomes fixed and the `DomainBase` is created. Then, for each restart, the generated nogoods are added to the base with their watched decisions. Below is an example that summarizes this chapter in steps of actions followed by the state of the base at that moment.

- a) When starting the resolution of a CSP, variables w , x , y and z with domains initialized to $\{1,2,3,4\}$ are created and then added to the nogood variables. For this example, $w_{\{1,2,3,4\}}$ represents the variable w with its domain as $\{1,2,3,4\}$. For each variable added, a new object named **VariableInfo** is created and a notifier for that variable is also made.
- b) Some constraints are posted and it affects the domain of the variables. Since the search has not started, the notifiers will not do anything yet even though some domains have changed.
- c) As soon as the DFS is started, the variable base will take shape. Each **VariableInfo** can be completed in conjunction with the **DomainBase** pairs. At this point the nogood base is also created but remains empty.

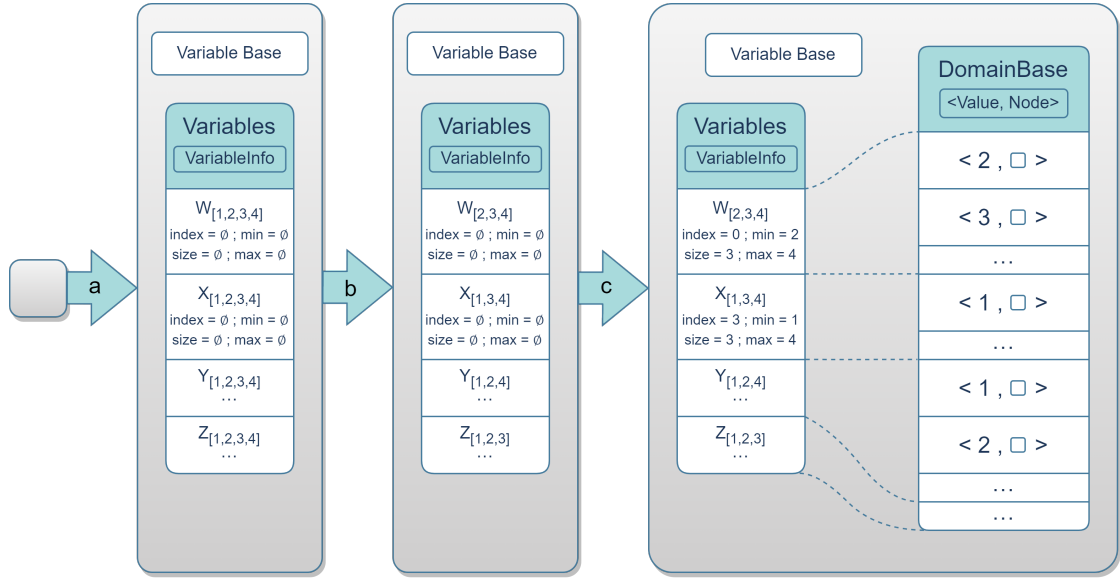


Figure 3.3: Steps **a-b-c** of the nogood base example

- d) Then the first restart reaches its stopping criterion and is able to generate a nogood which is directly added to the base. The list of decisions Δ_0 is stored in the **Nogoods** list and the pair of watched decisions are created. They refer to the first two decisions of the nogood and are also linked together. Once all the generated nogoods are stored, in this case only one, a new restart may begin.

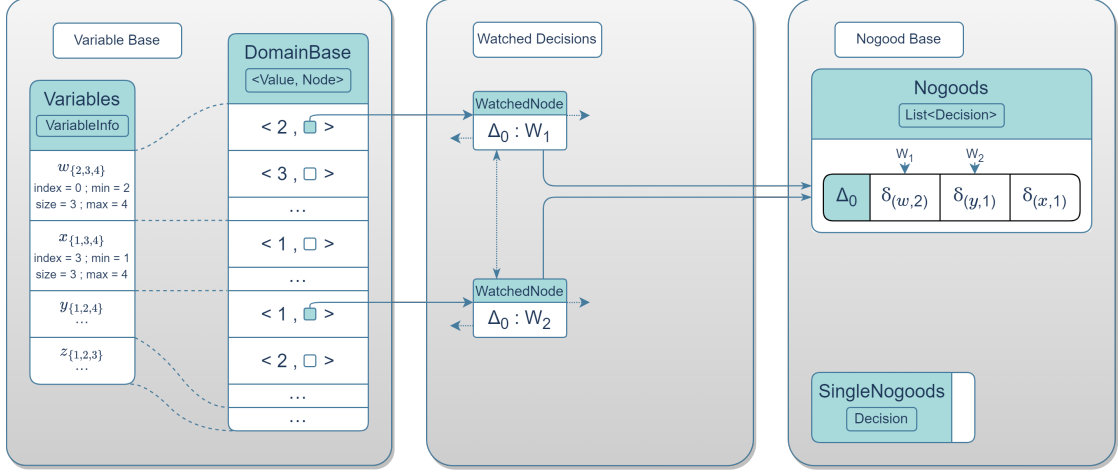


Figure 3.4: Step **d** of the nogood base example

- e) The following restart is able to generate two nogoods, a single nogood and one of longer size. They are added one by one starting from the smallest which has its decision added in the **SingleNogood** list. The second one Δ_1 is then processed the same way as Δ_0 with its watched decisions being added.

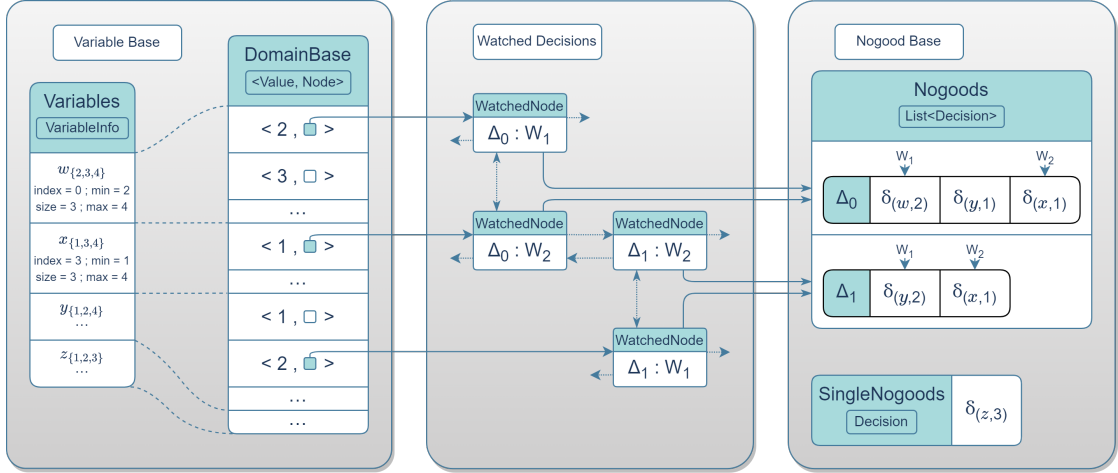


Figure 3.5: Step **e** of the nogood base example

In this example, the second watched decision of Δ_0 was moved in Figure 3.5. This corresponds to the inference process being made during restarts. Its implementation in MiniCP will be detailed in the following section.

3.4 Inferences with Nogoods

With a few nogoods in the base, the inference process will begin its operation. To clarify, inference process does not just mean making inferences, there are several steps that must be followed before a domain can be reduced. Therefore, the steps along with the implementation choices will be explained chronologically in this section. Note that explanations will remain general until Section 3.5 dives into specific cases.

3.4.1 Notifier Procedure

The first step is starting the inference process. As mentioned in Section 3.3.3, a notifier is created after each addition of a variable to the base. This is the mechanism that triggers the inference process whenever the variable's domain is altered. It cannot be linked to decisions because other constraints can also reduce the domain during the run. Algorithm 3.2 shows the procedure that the notifier executes when called.

Algorithm 3.2: Notifier procedure to begin the watched decisions update

```

Input:
    var                // variable which had its domain change
    oldDomain          // var's domain before the domain change

begin
    if  $\exists a : oldDomain \cap Dom(\delta_{(var,a)})^{30} = var.domain$  then
        info  $\leftarrow Variables[var.id]$ 
        initialDomain  $\leftarrow DomainBase[info.index, info.index+info.size]$ 
        foreach  $x \in initialDomain : Dom(\delta_{(var,x)}) \subseteq Dom(\delta_{(var,a)})$  do
            index  $\leftarrow findPosition(var, x)$ 
            node  $\leftarrow DomainBase[index].second$ 
            while node  $\neq null$  do
                updateWatched(node)
                node  $\leftarrow node.next$ 
            end
        end
    end
end

```

³⁰ The function *Dom* creates a corresponding set of integers from a decision given. For example *Dom*($x > 3$) gives 4,5,6,...

Algorithm 3.2 uses several concepts explained in Section 3.3 such as the **VariableInfo** object, the lists from the variable base, and the **findPosition** function. It starts by verifying whether the domain change corresponds to a new decision $\delta_{(var,a)}$ or not. If so, for any $\delta_{(var,x)}$ that may be created from the variable's initial domain prior to the search and that is implied from $\delta_{(var,a)}$, the algorithm executes the following steps. Locate the index in the **domainBase** corresponding to the decision $\delta_{(var,x)}$. Then if the resulting pair has watched decisions, iterate through the nodes to execute the function **updateWatched** with the node as argument. Hence, this function is the next section's subject.

3.4.2 Update Watched Decisions

The objective of the notifier procedure is to determine the watched decisions requiring update. By passing these nodes individually as arguments to the **updateWatched** function, attention can now be directed towards the associated nogood and its pair of watched decisions while explaining the function's implementation. Algorithm 3.3 describes how watches are updated. To facilitate the reading, some terms introduced in Section 2.4.3 such as satisfied, falsified, and free decisions have been used.

Algorithm 3.3: updateWatched

```

Input:
    node                                // watchedNode that needs updating

begin
    nogood  $\leftarrow$  Nogoods[node.nogoodIndex]
    firstDecision  $\leftarrow$  nogood[node.decisionIndex]
    secondDecision  $\leftarrow$  nogood[node.otherWatched.decisionIndex]
    if secondDecision is free then
        newDecision  $\leftarrow$  nextDecision(firstDecision, node)
        while newDecision  $\neq$  firstDecision do
            if newDecision is satisfied then
                return
            end
            if newDecision is free then
                updateNode(node, newDecision)
                moveNode(node)
                return
            end
            newDecision  $\leftarrow$  nextDecision(newDecision, node)
        end
        makeInference(secondDecision)
    end
end

```

Since this is the core of the inference process and the nogood constraint's implementation, every optimization matters. The main idea followed in Algorithm 3.3 is to minimize modifications to the watched decisions. Consequently, during execution, if it notices that the nogood is no longer relevant, the process is halted with a return statement. Here is a quick rundown of this algorithm.

First, it verifies that the second decision is free in order to continue. If not, it can mean two things. Either the second decision is satisfied and the nogood is already validated, or it is falsified meaning that at some point it tried to update but halted during its own execution of `updateWatched`³¹. Following that logic, no possible inference can be made so the algorithm is stopped.

After passing the verification, a `newDecision` variable is used to iterate over the other nogood decisions with the help of the `nextDecision` function. This function simply gives the next decision in a cyclic order with the direction attribute given by the node, while also skipping the `otherWatched` decision. The `newDecision` will be checked for each iteration. If it turns out to be satisfied, then the nogood is already validated and the algorithm is halted without any node modification. If it turns out to be free, then a new decision to watch is found meaning the node is updated, moved in the base and the algorithm returns.

If the iterator is able to traverse through the entire set of nogood decisions without finding a candidate, it implies that an inference on the `secondDecision` is necessary. To make this inference, appropriate actions are taken to remove elements from the domain based on the given decision.

3.4.3 Single Nogoods

Single nogoods are a special case, they are decisions that should immediately be inferred upon generation. Meaning that, based on the $\delta_{(variable, value)}$, certain values can be removed from the variable domain for the remainder of the search.

However, in MiniCP the inference cannot be made at any time. Removing an element from a domain will only be temporary because backtracking will restore previously saved state. Thus, the single nogoods generated can only be inferred when at the state of the root node.

As a result, a `singleNogoods` list is used to store each generated single nogood at the end of a restart. Then the restore states are executed to return back to the root node and the single nogood inferences can finally be made. Once done, the list is cleared.

³¹This last fact is only possible due to the backtracking order. If the `otherWatched` halted because it saw a satisfied decision, the backtrack will first change the `otherWatched` variable before the satisfied decision's variable. So it remains correct.

3.4.4 Example

To summarize, an example will provide a clear demonstration of when and how the inference process occurs. This example is a follow up from the previous one provided in Section 3.3, but takes place in the middle of a restart. During the previous, unmentioned restarts, additional nogoods have been added and watched decisions have been rearranged. As previously illustrated, the blue squares in the **DomainBase** represent a connection to a **WatchedNode**.

At a certain point in the search, the domain of x changes from $\{1, 3\}$ to $\{1\}$, triggering the inference process. Figure 3.6 illustrates the state of the variable base at this moment.

Initially, the algorithm identifies the decision that is responsible for the domain modification. For this example, let's say that it corresponds to $\delta_{(x,1)}$ without having other decisions that can be implied from it. Therefore, it is the only decision that will need the watches to be updated. The corresponding pair is found in the **DomainBase** and the nodes of the linked list are updated one by one.

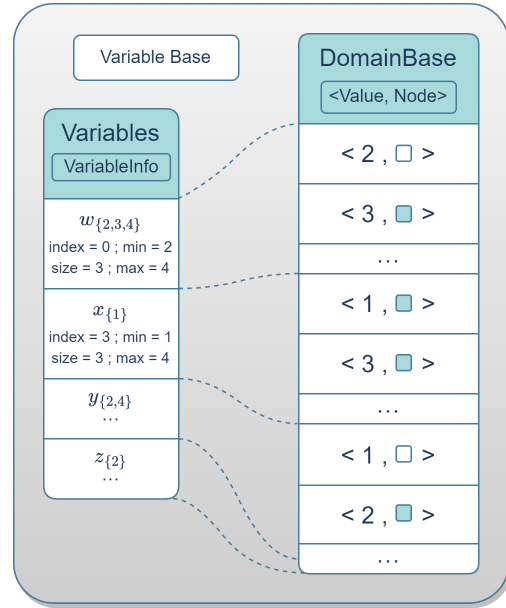


Figure 3.6: Initial state after modification of domain of x

Let the aforementioned linked list consist of four distinct nodes, each of which will be examined separately. These nodes lead to the nogoods depicted in Figure 3.7. To simplify the presentation of the decisions, the following color code will be used: blue for free decisions, red for falsified decisions, and green for satisfied decisions.

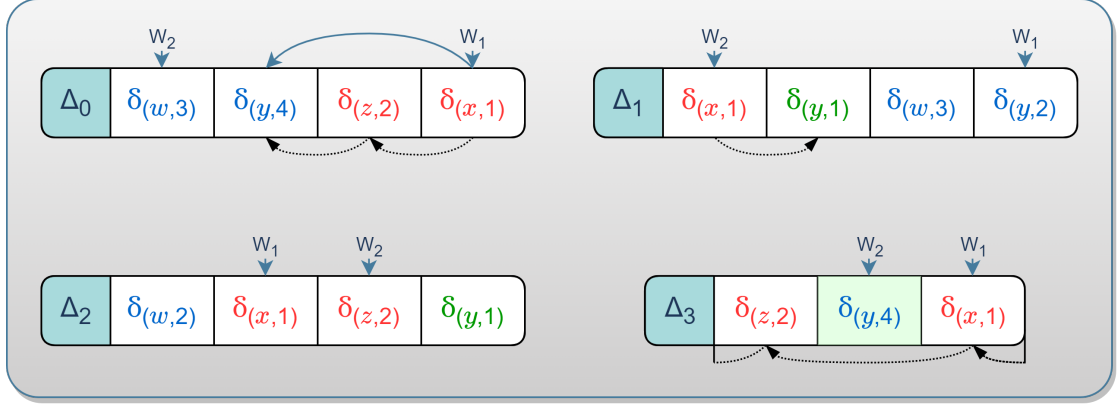


Figure 3.7: Example of the inference process followed for $\delta_{(x,1)}$ given 4 different nogood scenarios

The first node in the linked list leads to Δ_0 and is named W_1 . It is a watched decision that moves left, as it is the first watched registered during creation. Before checking for a new decision to watch, the algorithm verifies if W_2 is still free, which is the case. Subsequently, it starts testing for a new candidate until one is found or other events halt the algorithm's update. In this instance, the algorithm only encounters a falsified decision while searching for a free decision, and as a result, the watched decision is moved to $\delta_{(y,4)}$. This is done by moving the node to the corresponding linked list and updating its `decisionIndex`. No inference is made since another decision remains free.

The second node is a W_2 type that updates by going right and watches Δ_1 . After successfully performing the **otherwatched** check, it looks for a new candidate. However, the algorithm is immediately halted upon discovering that, while searching through candidates, $\delta_{(y,1)}$ had already been satisfied. Since it would be a waste of resources to move the watched there, no nodes are moved, and no inference is made since the nogood is not relevant in this state.

The third node watches a decision from Δ_2 and is a W_1 type, indicating that it should be searching for candidates on its left. However, while checking the state of the second watched decision, it notices that it is already falsified. This situation can only occur if, at some point, the second watched decision required updating and found a satisfied decision that immediately halted the update. As a result, no inferences or changes to the watched decisions have been made yet again.

The final node in the linked list is a W_1 and watches a decision from Δ_3 . This time, the other watched decision is free, and every other decision in the nogood is falsified. Therefore, an inference can finally be made using the decision $\delta_{(y,4)}$. Depending on the type of decision, one or multiple values may be removed from the domain of y .

Once all the decisions in the linked lists have been handled, the inference process is finally over and either more inference processes may result from it or a new decision in the tree is made and the program goes on.

3.5 Supported Branchings

The notation used so far follows Lecoutre’s generalized notation, which refers to positive decisions and their opposite, negative decisions. Additionally, as per Lecoutre’s approach, each positive decision will always branch to the left in the binary search tree, and to the right for negative decisions. This notation facilitates a general illustration of the algorithms involved, but the implementation itself is not as generalized. Therefore, for specific branching performance reasons such as different properties that can be used, it is more logical to have customized classes for each branching. However, since most functions can remain the same, a global interface `NogoodConstraint` is created, with an abstract class implementing it named `AbstractNogoodConstraint`. Then, each specific branching can become its own class that extends `AbstractNogoodConstraint`. For the purposes of this paper, the two types of branching discussed in Section 2.1.4 were implemented.

Recall that the first, used by Lecoutre, is the binary labeling branching where positive decisions are assignments (e.g. $x = 5$), while negative decisions involve rejections (e.g. $x \neq 5$). The second type of branching is the binary split where positive decisions apply a less than or equal to inequality (e.g. $x \leq 5$), while negative decisions apply a greater than inequality (e.g. $x > 5$). To implement these two types of branching, two additional classes named `NogoodConstraintEqual` and `NogoodConstraintLessOrEqual` were created by extending the `AbstractNogoodConstraint` class. The overall architecture is illustrated in Figure 3.8.

This section will describe each branching’s specificities, the way in which they were modeled, as well as their implementation.

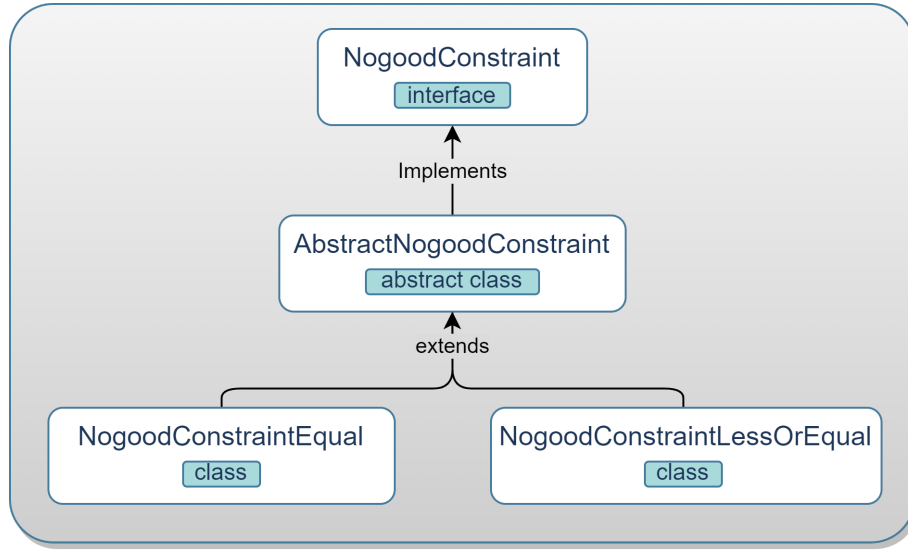


Figure 3.8: Diagram of classes used to model both branchings

3.5.1 Binary Labeling

Before implementing the binary labeling branching, it is necessary to determine when the base should be notified of any variable domain modification. As described in Section 3.3.3, a notifier was added for each variable stored in `Variables`. However, for this specific branching approach, the base only needs to be notified when the domain of a variable is restricted to a single value, which indicates that it is fixed and equivalent to an assignment. Only in this case can a negative decision of a constraint nogood become falsified. As a reminder from Section 2.4.4, only falsified decisions trigger the update of watched decisions in the hope of leading to an inference. Therefore, the general function `whenDomainChange` used to notify any modification to the domain of a variable is restricted to `whenFixed`³².

Now that the base is notified for the right reasons, the rest of the inference process can be focused on. Since it is started when a domain is fixed, the program receives the variable and value in one go, as it is the only one left in the variable's domain. These two parameters allow the `findPosition` function, introduced in Section 3.3.3, to identify the corresponding pair in `DomainBase`. If there are elements in the pair's linked list, the nodes will be iterated over and the update will work the same way as the generalized version.

³²For more information on the functions, refer to Section 2.5.3

The final part of the inference process that needs specifying for this branching is the inference itself. An inference is made when a free decision has to be satisfied since all the other decisions in the nogood are falsified. This is accomplished by removing the decision's value from the variable's domain in the current state. This cannot be forgotten because backtracks need to be able to undo these changes. The inference is achieved using the `IntVar.remove(int)` function in MiniCP, seen in Section 2.5.2. For example, when the free decision $x \neq 5$ of a constraint nogood needs to be inferred, the program executes `x.remove(5)`.

3.5.2 Binary Split

Compared to the previous branching option, the binary split branching approach is less intuitive to implement. This is primarily because the branching's properties require a more extended thought process to determine the circumstances under which the program should be notified of a domain change. Additionally, once notified, it is necessary to determine the appropriate action to take. Finally, it will be shown that the generation of nogoods for this branching can be adapted to act more efficiently.

In the same order as the binary labeling branching, the first step is to understand how a decision from a constraint nogood can become falsified since this is when the notifier should come into action. To come to the right conclusions, the decision $x > 5$ is used as an example. This decision becomes falsified if $x \leq 5$, meaning x does not have any values above 5. Note that the previous statement can be reduced to $x.max() \leq 5$ since the maximum of the domain is enough to include the other values. Hence, when a domain is altered, all existing decisions in constraint nogoods, associated with values between the new maximum and the old one become falsified. This functionality is implemented in MiniCP with the `whenBoundChange` function paired with a check on the maximum change. Although, the old maximum then needs to be stored in a `StateInt` object to stay valid upon backtracking.

To retrieve the pairs in the `DomainBase`, the program will first find the position of the new maximum value with the function `findPosition`, process the second element and then iterate over the following values until the old maximum is reached. It is only allowed because the `DomainBase` represents the initial domain of variables. Each second element of the pairs is managed just like they are in the binary labeling branching except for the inferences. Those are performed this time with `IntVar.removeBelow(int)` by then increasing the value by 1. For instance, to infer the free decision $x > 5$ in a constraint nogood, the program performs `x.removeBelow(6)`.

As an example, Figure 3.9 illustrates values to iterate over for variable x in which the domain changes from $\{1, 2, 3, 4, 5\}$ to $\{1, 2\}$.

The program has to iterate over pairs with values 2, 3 and 4. 2 is included since being the new maximum, no more values in x are greater than 2 and thus any watched decisions related becomes falsified. 5 is excluded for this reason since it was considered as the max_{old} of the previous domain reduction and is thus already processed.

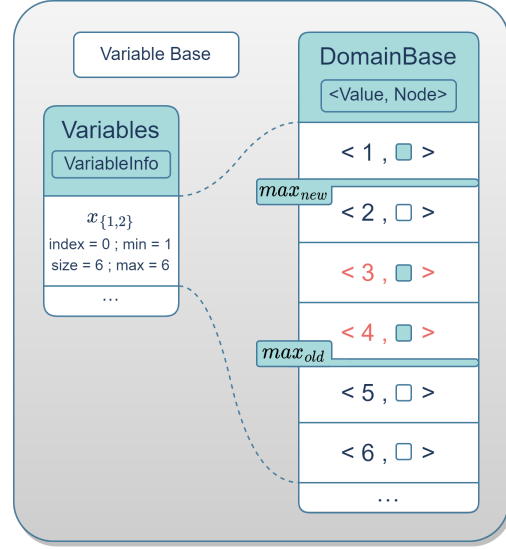


Figure 3.9: Example illustrating values to iterate over after a domain reduction

Minimized Nogoods Generation

However, out of the two branchings considered in this thesis, the binary split branching has two interesting properties. Firstly, positive decisions on the same variable can be performed multiple times within the same branch of the search tree. Secondly, the values assigned to these inequalities are in decreasing order from the root. By combining these two properties, a minimized version of nogoods can be created during their generation with only one pass over **Decisions**. Indeed, if multiple decisions of identical type on the same variable are present in **Decisions**, the one furthest from the root will include the preceding ones for a nogood. The example from Figure 3.10 illustrates this property.

The first nogood Δ_0 is generated as previously seen in Section 2.4.2 but the second nogood Δ_1 has been minimized. Indeed, if the classic rule is followed, Δ_1 should have been $\{\delta_{(x,5)}, \delta_{(z,7)}, \delta_{(x,3)}\}$ instead of $\{\delta_{(x,3)}, \delta_{(z,7)}\}$. To understand why, imagine that in a further restart where $\delta_{(x,5)}$ and $\delta_{(z,7)}$ of the classic form are respectively watched by W_1 and W_2 , decision $z \leq 6$ is taken. Then, W_2 would move to $\delta_{(x,3)}$. In that case, the only way for an inference to occur would be when a decision $x \leq a$ with $3 < a \leq 5$ is taken further in the search. In that case, $\delta_{(x,5)}$ would become falsified and $x > 3$ would be inferred. As a result, the regular form

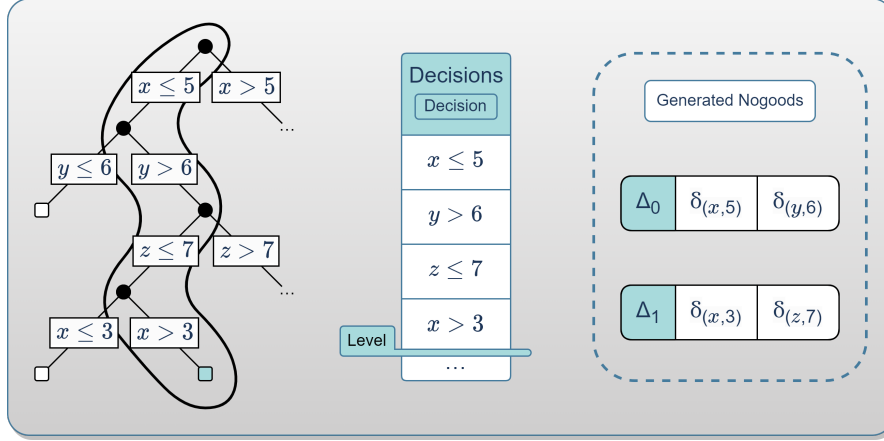


Figure 3.10: Minimized nogood generation for binary split branching

would simply be one step behind the minimized form since $x > 3$ would already be inferred after decision $z \leq 6$. The nogood constraints equivalence can also be perceived thanks to the disjunction of negative decisions where $x > 5 \vee z > 7 \vee x > 3$ is identical to $x > 3 \vee z > 7$.

In the code, this property has been implemented thanks to a **HashMap** mapping the variable identifier of decisions to their position in the base **nogood** of Algorithm 3.1. Thus it allows, when adding a positive decision to **nogood**, to update an existing decision on the same variable or simply add it. It works the same way for a negative decision except that it is the copied nogood **new** that is modified.

Lastly, this minimization is not equivalent to the one of Lecoutre, discussed in Section 2.4.5. Indeed, his approach aims to reach a minimal ϕ -nogood through multiple iterations over the same nogood. Since this process impacts performances, it will not be used and nogoods will simply be minimized as much as possible in a single pass.

3.6 Large Neighborhood Search with Nogoods

Integrating nogoods with LNS adds some complexity to the implementation, which will be explained in this section. Moreover, there are some additional restrictions that need to be imposed on the use of LNS when combined with nogoods.

As a quick summary from Section 2.3, the goal of LNS is to keep some variables assigned from a previously found solution to narrow the search tree down. This

works fine when the algorithm is allowed to run wild without saving previous restarts but not when nogoods are in play. For LNS to work with nogoods, every decision made in LNS has to be saved in the **Decisions** list just like if it was a branching with increasing depth. Without this trick, the generated nogoods from the DFS executed after the LNS fixes would be incorrect. In addition, every decision in the **Decisions** list has to be of the same type. Thus when using LNS with nogoods, the decisions used to narrow the DFS from the previous solution have to be of same type as the branching used in the DFS. Finally, to be sure that nogoods cannot block possible solutions, only positive decisions are allowed. As this previous fact is complex to explain, a proof by contradiction is shown in Figure 3.11.

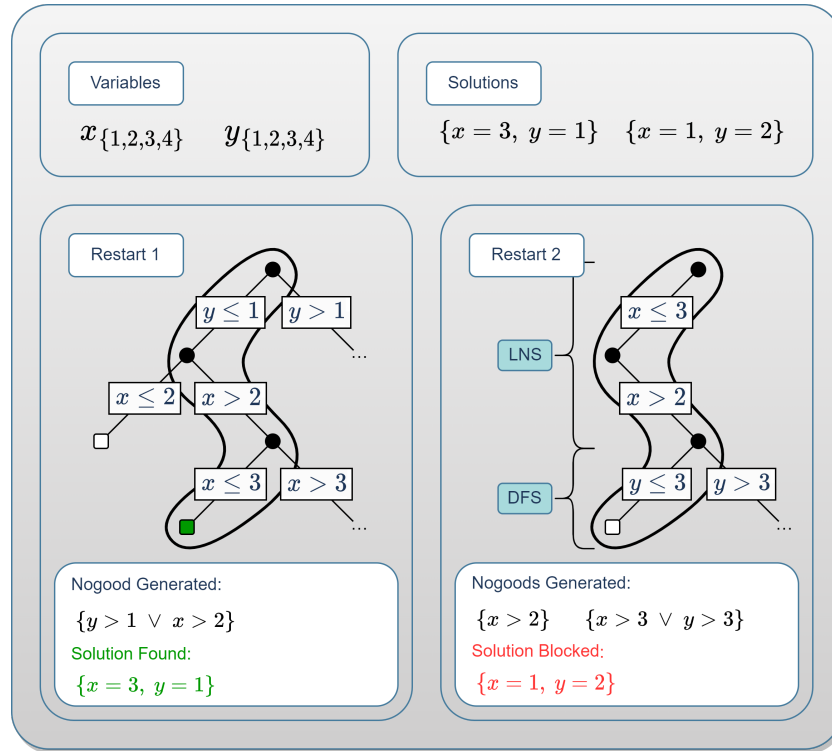


Figure 3.11: Proof by contradiction, negative decisions cannot be used with LNS

Figure 3.11 presents a problem with two variables that is solved using LNS and nogoods to identify both given solutions. These restarts end on failed or successful states so that the last nodes can be used in the nogoods. In the first restart, it discovers the first solution and generates a nogood based on the resulting path. The second restart is then able to use LNS by fixing x to 3 as per the previous solution. Then it starts its own DFS until reaching a failed state. Two new nogoods are generated, but they both prevent the second solution from being found in

subsequent restarts. Consequently, only positive decisions can be used by LNS in conjunction with nogoods to have a complete search.

The binary labeling branching is not influenced by these restrictions since posting an assignment decision will immediately lead to a fix. When using the binary split branching though, it becomes more of an inconvenience. If the user wants to fix a variable x to 5, he is only allowed to post a $x \leq 5$ constraint. It becomes clear that the mix of a binary split branching with nogoods and LNS becomes useless, hence it will not be analyzed in this paper.

With the restrictions applied, the focus can now shift to the implementation aspect of the problem. To add decisions to the **Decisions** list even though the search tree level stays at zero³³, an additional offset is used. This offset is incremented for every LNS decision made before the search has officially started. Adding the level and offset together allows for the decisions made from LNS to be saved successively. Then the root of the tree will be placed at the next available index in the list. Figure 3.12 shows an example of this concept using binary labeling branching.

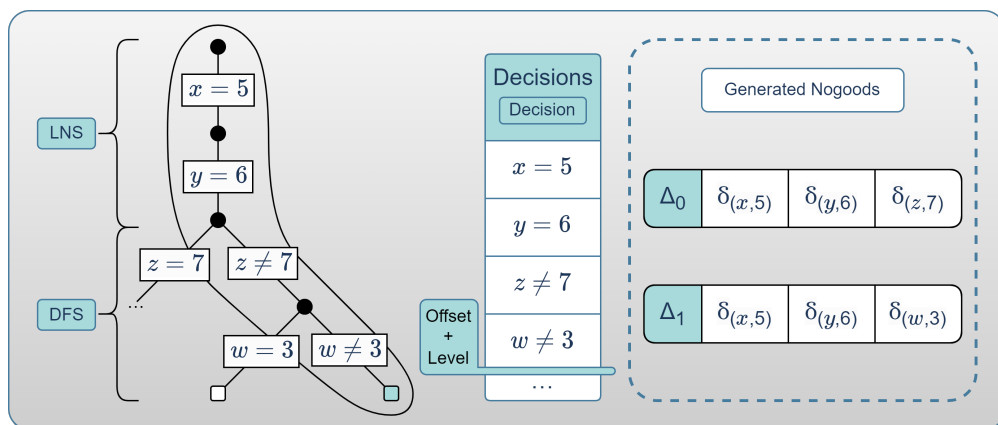


Figure 3.12: Example of nogood generation with LNS

3.7 Solving Problems with Nogoods

Having described the individual components that enable the use of nogoods in MiniCP, the focus can now shift to joining them together in order to solve problems. The following section explains their interactions with one another and the core of MiniCP.

³³As explained in Section 3.1, the level only increases when the search goes deeper in the tree.

As a reminder of Section 2.5, the main steps involved in solving a problem with MiniCP include declaring the solver and variables, adding constraints to the variables, specifying the problem's objective if applicable, and finally launching the search. Among all these, the user only needs to specify the branching operated for nogoods, the variables useful for this branching and an optional threshold value.

For this purpose, the function `Factory.makeSolver` and the constructor `MiniCP` include a new parameter specifying the branching to use for nogoods management which informs the program to use nogoods. The other classes, on the other hand, can identify the usage of nogoods through the function `MiniCP.isNogoods`. If nogoods are requested, the constructor `MiniCP` instantiates the appropriate class, among `NogoodConstraintEqual` and `NogoodConstraintLessOrEqual`, based on the specified branching type. The instance is then passed to the `DecisionPath` constructor, enabling the storage of generated nogoods in the base via the function `NogoodConstraint.addNogood`. Moreover, providing this instance to `DecisionPath` allows the class to consider the maximum length of nogoods during generation if a threshold exists.

Regarding problem's variables, it is the user's responsibility to specify the ones necessary to determine the solution's validity, which will be useful to make inferences. Consequently, it does not include the variables focused on optimizing the solution such as an objective for COP. The designation is performed via `NogoodConstraint.addVariable` function which accepts `IntVar`, `IntVar[]`, and `IntVar[][]` for user convenience. To access this function, there is a getter in `MiniCP` class that gives the base instance from the problem class through the return value of `Factory.makeSolver`. When a variable is added, the base sets its `id` attribute with an internal counter, adds a `VariableInfo` to the `Variables` list, and associates a specific notifier based on the branching. However, the attributes of `VariableInfo` are not set at that moment since the domain can always be reduced before the DFS is launched.

Although the user does not have to do anything else, it is still not functional. Indeed, in addition to the `Variables` list not being initialized, decisions taken in the DFS are not stored in `Decisions` and nogoods are not generated when restarting.

To address these issues, the `DecisionPath` instance is given to the constructor `DFSearch`, called when creating the DFS with `Factory.makeDfs`. Then, when the resolution of the problem is started with functions like `solveSubjectTo`, `optimizeSubjectTo`, or their simpler version without the closure executed before

the search starts, `Variables` list is updated. This is done with `NogoodConstraint.start` function, which is accessible via the `DecisionPath` instance.

Subsequently, regardless of the type of resolution, the `solve` function is eventually called, which triggers the execution of the `dfs` function within a `try` statement. As mentioned in Section 2.5.6, the `dfs` function performs branching by posting constraints, also referred to as decisions. To store them in the `Decisions` list, the `DecisionPath.addDecision` function is invoked when posting the constraint. This function takes the level and a `Decision` object as arguments, which is made from the constraint. The creation of this object is possible thanks to constraints in `Factory` which have been modified to include a `Decision` constructor, with the variable and value given.

In the case where a `StopSearchException` is raised, the `catch` comes into play and calls the `DecisionPath.generateNogoods` function with the according level, since it is due for a restart.

Now that every single implementation aspects have been tied together, nogoods have successfully been integrated in MiniCP.

Chapter 4

Experiments and Analyses

The aim of this chapter is to evaluate the impact of integrating nogoods on different restart strategies. These are MGAC+RST, MGAC+RST+NG, MGAC+RST+NG+TSH, MGAC+LNS and MGAC+LNS+NG. To simplify the writing, the characteristics have been abbreviated. MGAC stands for maintaining generalized arc consistency, which is done by default in MiniCP³⁴. In addition to this, RST refers to restarts, NG to nogoods and NG+TSH to nogoods with a threshold, i.e. a maximum length. LNS, as before, stands for large neighborhood search.

The main measurements are the CPU execution time, the number of nodes visited by the search tree³⁵ and the objective value reached. The benchmarks were performed with a Windows 10 OS, Intel i7-8550U processor clocked at 1.8 GHz and 16 GB RAM.

During each test, regardless of the heuristic used, randomness can influence the search positively or negatively³⁶. In addition to that, nogoods also affect the search with their inferences. Thus, comparing the performance of searches with and without nogoods is a complex task. To address this challenge, the different strategies follow the same randomness across each restart by sharing the same seed, which is renewed differently upon each restart. Appendix B demonstrates an example of this concept. This approach maximizes the similarity of the path followed. But as soon as an inference is made by nogoods, it could lead to a divergent randomness pattern for the remainder of the restart. As this final fact implies that a degree of luck-based solution finding still influences the performance

³⁴To be more precise, generalized arc consistency is done when using domain consistent constraints. However the bound consistent constraints used to model the problems of this chapter actually perform domain consistency since they are less or equal constraints.

³⁵For restart strategies, the statistics of each restart's search tree are summed.

³⁶Even if the choice of variable or value is not made randomly, the ties are broken at random.

of strategies, the tables shown in this chapter display averages from multiple runs for each instance.

Section 4.1 shows experiments done on the JobShop Problem with a restart approach while Section 4.2 show results of the Traveling Salesman Problem solved with LNS. Each section includes data shown with graphs and tables, followed by a factual analysis with additional information such as calculations from the benchmarks. Note that this chapter does not show results of CSP instances as the ones tested were not interesting. Various instances of the Graph Coloring Problem depicted in Section 2.1.1 and the N-Queens Problem depicted in Section 2.1.5 lead to an immediate solution or none at all after considerable time when using restarts with random based heuristics. The results from such observations are displayed in Appendix C.1.

4.1 JobShop Problem

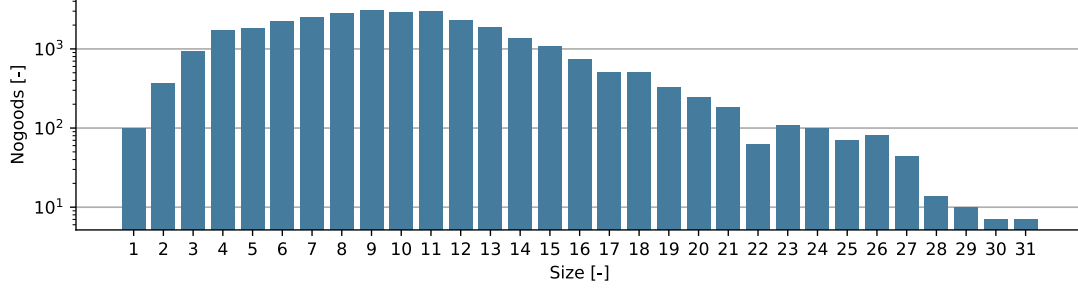
The JobShop Problem, detailed in Section 2.1.2, is used to evaluate nogoods with a basic restart strategy. To ensure that the assessment is as realistic as possible, it is carried out over 31 different instances. These originate from the JSPLIB[21] which centralizes the instances from [22, 23, 24]. For this problem, the two different branchings of Section 3.5 were used, namely binary labeling and binary split, with their own heuristic.

4.1.1 Nogoods Threshold Selection

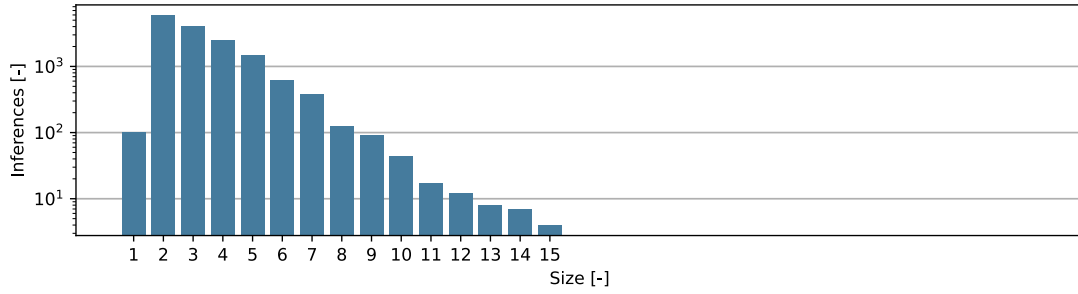
Choosing the right threshold value can be done in two ways. Choosing a fixed and arbitrarily chosen threshold, such as 10 decisions per nogood, or opting for one that varies based on the instance. For the JobShop Problem, each threshold value was chosen according to a profiling run of the instance. While profiling does introduce some overhead due to the extra data collection, it demonstrates the behavior of nogoods on that particular instance and parameters. Thus the threshold chosen with a profiling run is only valid when using the same style of heuristic and branching as an alteration in these will change the shape of the search tree.

Figure 4.1 shows the data collected during the resolution of **abz5** instance with MGAC+RST+NG, binary labeling branching, a failure limit of 50, and a random variable and minimal value ordering heuristic. Even though 31 different nogood sizes are recorded, as shown in Figure 4.1a, only the first 15 were able to make inferences, as can be observed in Figure 4.1b. In addition to that, Figure 4.1c shows

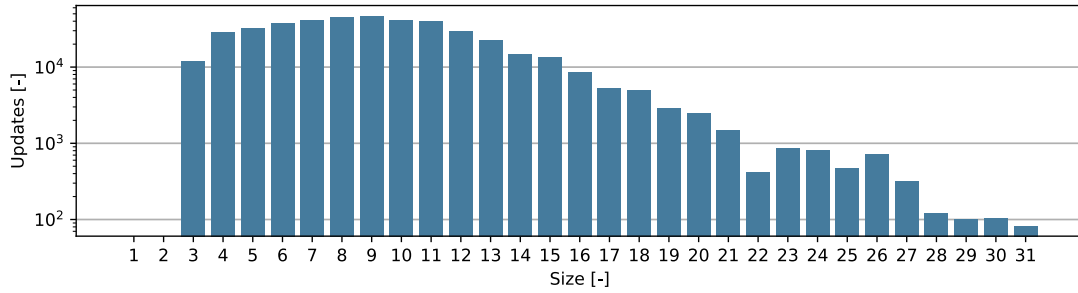
that the watched decisions are still constantly updated for all nogoods. Dividing the inferences by the number of nogoods of the same size reveals the impact that a nogood of a particular size has on the search.



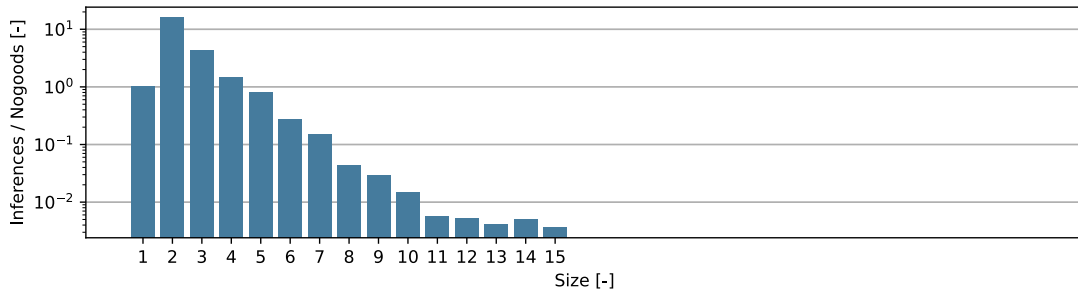
(a) Number of nogoods based on their size.



(b) Number of inferences made for each nogood size.



(c) Number of watched decisions updated for each nogood size.



(d) Impact that a nogood has on the search based on its size.

Figure 4.1: Measures obtained during resolution of **abz5** JobShop Problem instance with MGAC+RST+NG, binary labeling branching, and restarts' failure limit of 50.

Figure 4.1d shows how many inferences are made on average for a nogood depending on its size. Of course, a single nogood will only make one inference since it simply removes an element from the initial domain, as explained in Section 3.4.3.

With the collected data, making an informed choice for the appropriate threshold value of an instance becomes more manageable. Setting the threshold at 15 keeps 100 % of inferences but only removes 10 % of the stored nogoods. With a threshold of 9, 99 % of inferences are kept while 50 % of nogoods are discarded. Lastly, a threshold of 6 keeps 95 % of inferences and removes 67 % of nogoods. For each instance in this section, the threshold has been chosen so that ~ 99 % of inferences are kept.

4.1.2 Restarts with Binary Labeling

The heuristic used throughout this section, which remains the same for all strategies tested, is a random variable and minimum value ordering heuristic. In addition they all share the same branching type, namely binary labeling.

Table 4.1 shows the computation time and number of nodes visited of various restart variations to reach the optimal objective value $f(\sigma^*)$ on 5 JobShop Problem instances. The sizes of these instances are described by two numbers representing the number of jobs and machines, in that order. Note that the number of tasks per job is equal to the number of machines. The restart variations are regular restarts and learning based restarts using nogoods, either limited in size with a threshold or not. Each value in the table is an average of 10 runs and the failure limit of restarts is fixed to 50 with some variations of the number analyzed later in the section. A MGAC evaluation has also been conducted, however the corresponding data is not shown since it only consists of timeouts as the heuristic is not fit for such a search strategy.

According to the data, MGAC+RST+NG is slightly faster than MGAC+RST from the second instance onward. The real difference is on **1a18** where MGAC+RST+NG is ~ 14 % faster resulting in 34k fewer nodes visited. When comparing the threshold performance, MGAC+RST+NG+TSH is slightly faster than MGAC+RST+NG on the first instances and a bit slower on the last two. However, the nodes visited remain similar for both options.

Instances	Size	Statistics	MGAC+RST		
				+NG	
					+TSH
1a12	20 x 5	Time [s]	4.69	5.04	5.03
		Nodes	12 141	12 142	12 142
abz5	10 x 10	Time [s]	55.01	54.9	54.18
		Nodes	73 903	73 884	73 885
abz6	10 x 10	Time [s]	71.15	68.85	68.3
		Nodes	92 092	92 135	92 136
1a18	10 x 10	Time [s]	264.43	232.07	232.99
		Nodes	354 k	320 k	320 k
1a23	15 x 10	Time [s]	274.03	272.22	272.43
		Nodes	246 k	246 k	246 k

Table 4.1: Results obtained with MGAC+RST, MGAC+RST+NG and MGAC+RST+NG+TSH on 5 JobShop Problem instances with 10 runs each, binary labeling branching, restarts' failure limit of 50 and a timeout of 600 seconds.

Since measures of Table 4.1 are only performed on 5 instances, more complete analyses are displayed in Figures 4.2 and 4.3. They show the pairwise comparison of CPU time between MGAC+RST, MGAC+RST+NG and MGAC+RST+NG+TSH with the same failure limit on 310 measurements originating from 31 JobShop Problem instances with 10 runs each. The points aggregated in the upper or right borders are those that exceeded the time limit of 600 seconds.

The points of Figure 4.2 are generally distributed along the diagonal. Out of the 310 measures, MGAC+RST+NG performs better than MGAC+RST for 177 of them and vice versa for the remaining 133. When normalizing the time difference, MGAC+RST+NG is 5.06 % faster than MGAC+RST. This percentage is calculated by comparing the averaging performance ratio of each strategy. This average ratio is defined below.

Given instances $I = \{i_0, \dots, i_{n-1}\}$, strategies $S = \{s_0, \dots, s_{m-1}\}$ and $t_{i,s}$ the computing time required to solve instance i by strategy s ³⁷, $\forall i \in I, s \in S$, $r_{i,s} = \frac{t_{i,s}}{\min_{s \in S} t_{i,s}}$ is called the performance ratio of strategy s on instance i compared with others strategies[25]. The averaging performance ratio of a strategy s over all the instances is then denoted as $\bar{r}_s = \frac{1}{n} \sum_{i=0}^{n-1} r_{i,s}$. When comparing two of these average ratios, a simple subtraction is sufficient to obtain the coefficient indicated above, expressed as a percentage.

³⁷Here, $t_{i,s}$ is the arithmetic mean of the computation time needed to solve instance i by strategy s over 10 runs.

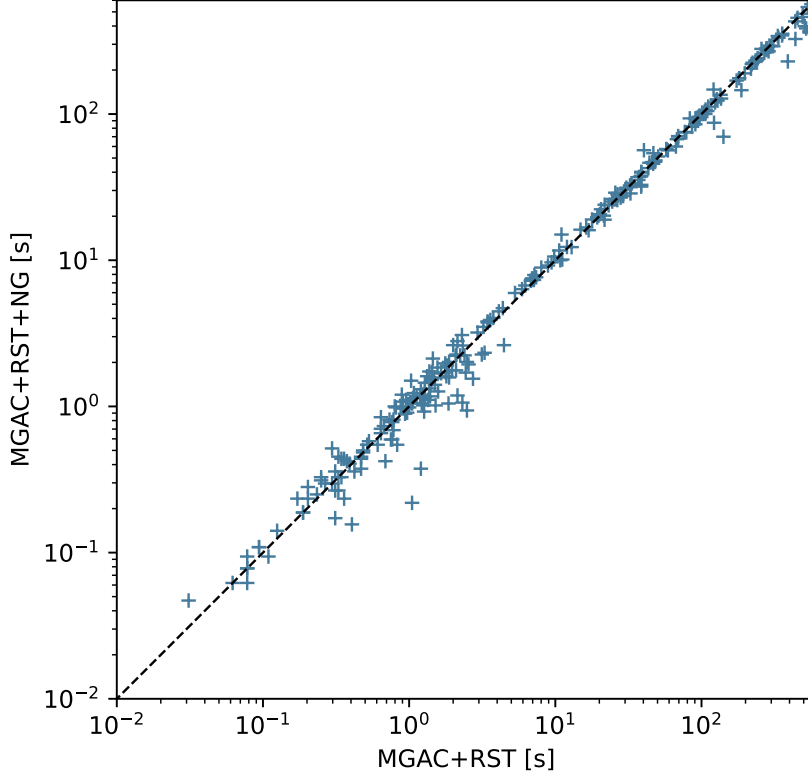


Figure 4.2: Pairwise comparison of CPU time between MGAC+RST and MGAC+RST+NG on 31 JobShop Problem instances with 10 runs each, binary labeling branching, restarts' failure limit of 50 and a timeout of 600 seconds.

One final analysis that can be done from Figure 4.2 is that there is a difference between smaller and larger instances. Points above 5 seconds tend to follow the diagonal while it is more spread out below that time. When only taking into account the points below the 5 second mark, MGAC+RST+NG is 6.98 % faster than MGAC+RST. However, for the data points after 5 seconds, MGAC+RST+NG is only 2.1 % faster than MGAC+RST.

When changing the failure limit of restarts, these values change. Since the pairwise graphs only differ slightly, Figures C.1 and C.2 of Appendix C show the same evaluation, on 28 instances however since the three remaining always produce timeouts, but using a failure limit of 10 and 100, respectively. With a failure limit of 10, out of the 280 measures, MGAC+RST+NG takes less time than MGAC+RST for 150 of them. In addition, MGAC+RST+NG is 2.21 % faster than MGAC+RST.

Finally, when using a failure limit of 100, MGAC+RST performs better on 145 out of the 280 measures and is 0.9 % faster than MGAC+RST+NG.

The pairwise comparison from Figure 4.3 shows a similar difference between MGAC+RST+NG+TSH and MGAC+RST+NG as in Table 4.1. Here, 200 measures out of the 310 perform better when using a threshold, it is also 2.79 % faster than its counterpart.

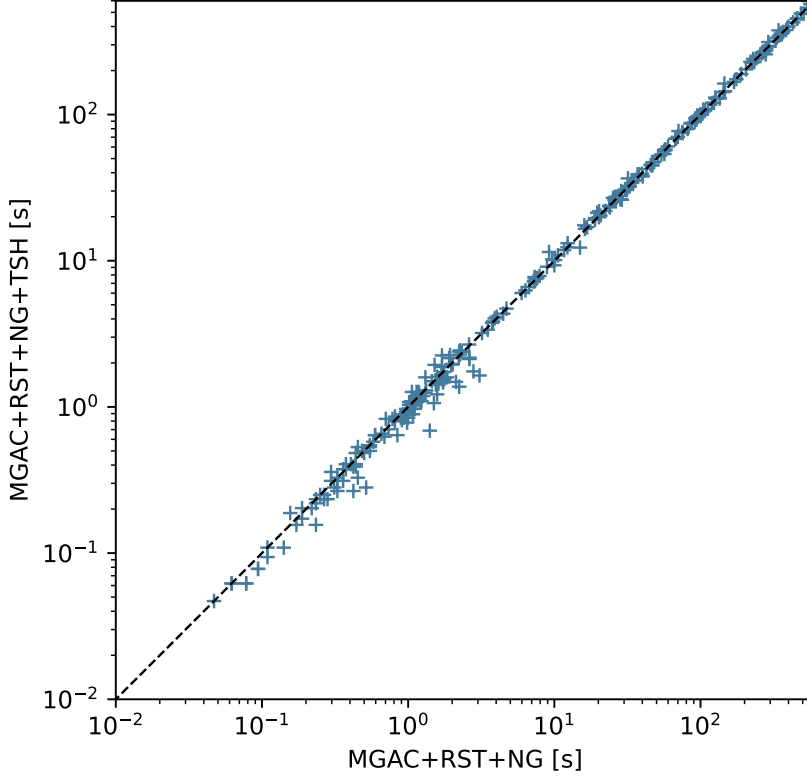


Figure 4.3: Pairwise comparison of CPU time between MGAC+RST+NG and MGAC+RST+NG+TSH on 31 JobShop Problem instances with 10 runs each, binary labeling branching, restarts' failure limit of 50 and a timeout of 600 seconds.

Finally, Figure 4.4 shows the performance profile[25] of the three discussed strategies. A performance profile of a strategy s is the cumulative distribution function ρ_s of a performance metric, which are the $r_{i,s}$ here. Therefore, $\rho_s(\tau) = \frac{1}{n}|\{i \in I \mid r_{i,s} \leq \tau\}|$ indicates the probability for a strategy s that a performance ratio $r_{i,s}$ for instance i is within a factor $\tau \in \mathbb{R}$ of the best possible ratio. In other words, it denotes the fraction of instances of I that the strategy s can solve within a factor τ of the best strategy. However, the performance profile from Figure 4.4 is not made on instances but on the 10 runs made for each of the

31 instances. That is done in order to reward strategies that complete a run that other searches may have failed due to a timeout. Therefore, neither these measures nor the instances in question are deleted from the pool.

Figure 4.4 shows that MGAC+RST has 38.71 % of probability of being the optimal search. In addition, it solves 42.89 % of the runs the most efficiently within a factor $\tau \leq 1.01$ of the best strategy. If τ ranges between $[1.01, 1.03]$ it is not clear which one among MGAC+RST and MGAC+RST+NG+TSH is better since they keep one upping each other. However, MGAC+RST+NG+TSH surpasses MGAC+RST for a factor $\tau \geq 1.03$ of best the strategy. Indeed, it can solve 87.75 % of runs most efficiently for that factor. It is also the most robust for such a factor because it can solve more runs than any other search. In comparison, MGAC+RST+NG ultimately reached 87.42 % and MGAC+RST 86.77 %. The slight differences between these percentages are due to the fact that MGAC+RST+NG has 1 more timeout than MGAC+RST+NG+TSH whereas MGAC+RST has 3 more. A last observation is that MGAC+RST+NG becomes more efficient than MGAC+RST to solve 63.87 % up to 87.42 % of problems within a factor $\tau \geq 1.09$ of the best strategy.

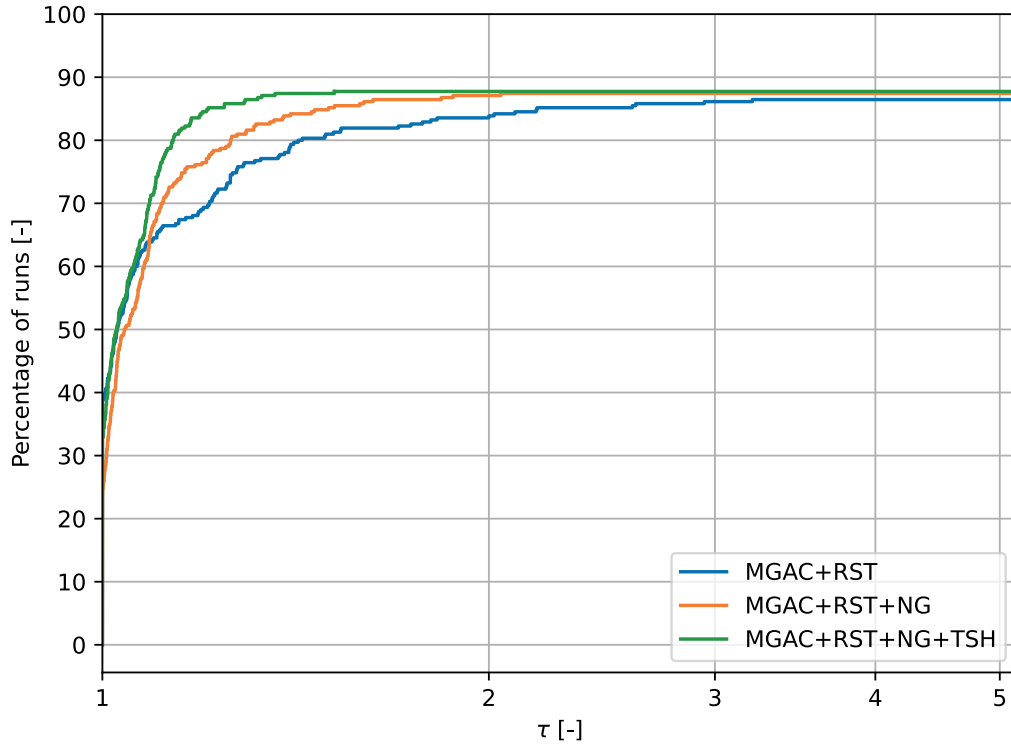


Figure 4.4: Performance profile of CPU time between MGAC+RST, MGAC+RST+NG and MGAC+RST+NG+TSH on 31 JobShop Problem instances with 10 runs each, binary labeling branching, restarts' failure limit of 50 and a timeout of 600 seconds.

4.1.3 Restarts with Binary Split

This section shows the influence of nogoods when solving the JobShop Problem with a binary split branching this time. The heuristic used is a random variable and a random value selection. Note that the local maximal value cannot be chosen since in that case, the left branch would have the same state outside randomness and the right branch would directly lead to a domain wipeout before any propagation. Also, the nogoods referred by NG here are the minimized ones described in Section 3.5.2. Finally, the failure limit of restarts was fixed to 50 for this whole section.

Table 4.2 shows different performances of restart variations in the same format as in Table 4.1. The data is also an average of 10 runs but with 6 instances instead of 5. According to the data, apart from instance **1a18**, MGAC+RST is faster at solving the instances than MGAC+RST+NG. In terms of nodes traveled, the most interesting strategy varies since it is often different for each instance. Between MGAC+RST+NG and MGAC+RST+NG+TSH, there is a slight average improvement in time when adding the threshold, and the same goes for the nodes visited.

Instances	Size	Statistics	MGAC+RST		
				+NG	+TSH
1a15	20 x 5	Time [s]	5.05	5.53	5.51
		Nodes	12 537	12 537	12 537
abz6	10 x 10	Time [s]	10.11	13.9	16.5
		Nodes	17 815	20 932	24 137
1a18	10 x 10	Time [s]	17.25	12.69	11.93
		Nodes	26 444	19 913	18 934
abz5	10 x 10	Time [s]	55.82	70.44	59.55
		Nodes	75 641	84 343	74 035
1a23	15 x 10	Time [s]	62.0	67.57	66.96
		Nodes	63 984	63 700	63 700
1a30	20 x 10	Time [s]	278.61	307.96	292.86
		Nodes	191 k	190 k	183 k

Table 4.2: Results obtained with MGAC+RST, MGAC+RST+NG and MGAC+RST+NG+TSH on 6 JobShop Problem instances with 10 runs each, binary split branching, restarts' failure limit of 50 and a timeout of 600 seconds.

The nogood performance on the branching type analyzed here compared to the binary labeling branching shown in the previous section is noticeably worse. A

significant factor contributing to this is that the binary split branching used here is capable of performing significantly fewer inferences compared to the previous one. To illustrate, here are the results when testing nogoods on the 1a23 problem instance. The binary labeling branching managed to solve it within 212s while recording 19k nogoods and making 4.3k inferences. On the other hand, the binary split branching solved the same instance in 69s while recording 5k nogoods and only making 59 inferences. Even when accounting for the time difference, the second branching is clearly outmatched by the first in terms of nogood performance.

Figure 4.5 displays the pairwise comparison of MGAC+RST and MGAC+RST+NG to solve 31 instances, 10 times each, in CPU time. Out of the 310 measures, 193 of them are solved faster with MGAC+RST. It also follows the same averaging performance as seen in Table 4.2 since MGAC+RST is 4.99 % faster than MGAC+RST+NG. There is a clear difference between the smaller and larger measures however. When analyzing the data points under the 3 second mark, MGAC+RST is 9.39 % faster than MGAC+RST+NG. On the contrary, when keeping the ones above that time, MGAC+RST is 2.95 % slower than MGAC+RST+NG.

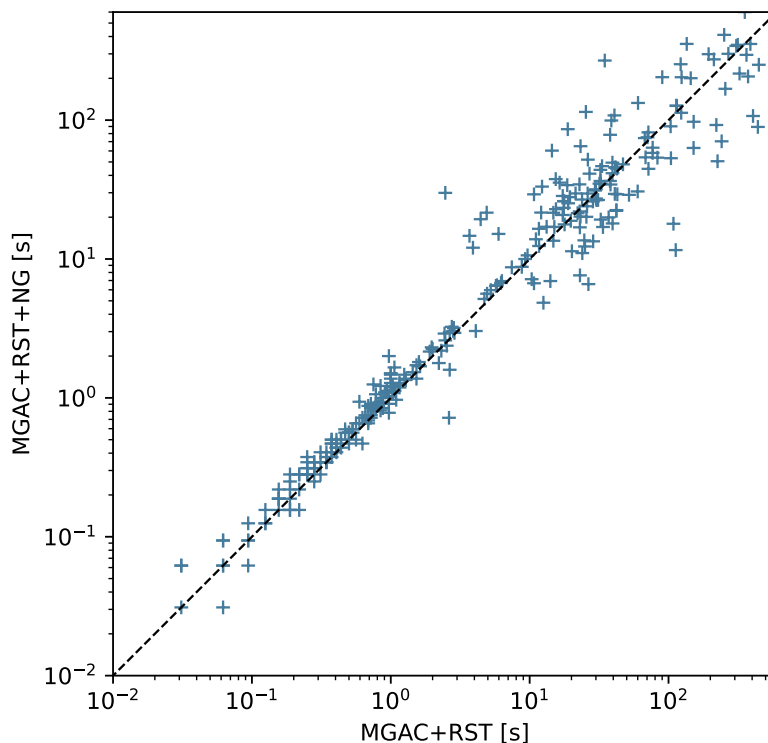


Figure 4.5: Pairwise comparison of CPU time between MGAC+RST and MGAC+RST+NG on 31 JobShop Problem instances with 10 runs each, binary split branching, restarts' failure limit of 50 and a timeout of 600 seconds.

Figure 4.6 shows the same comparison as the previous figure except that it is with MGAC+RST+NG and MGAC+RST+NG+TSH. Out of the 310 measures, 194 of these are better with a threshold and the corresponding strategy is also 0.71 % faster than MGAC+RST+NG.

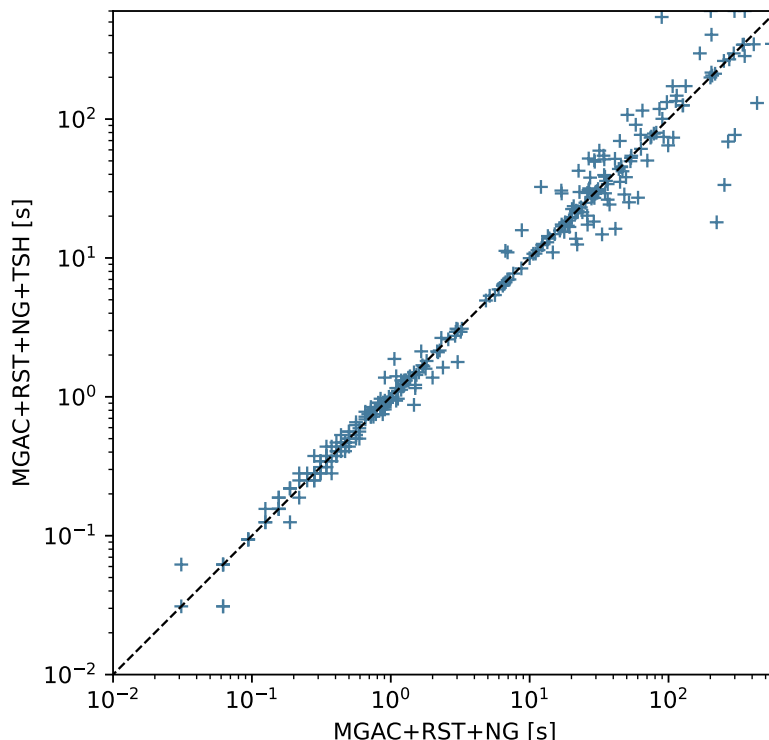


Figure 4.6: Pairwise comparison of CPU time between MGAC+RST+NG and MGAC+RST+NG+TSH on 31 JobShop Problem instances with 10 runs each, binary split branching, restarts' failure limit of 50 and a timeout of 600 seconds.

Finally, Figure 4.7 shows the performance profile of the three discussed strategies which, like Figure 4.4, is made on the 10 runs performed for each of the 31 instances. It accentuates again the fact that MGAC+RST performs better than other strategies on this branching. Indeed, it shows that MGAC+RST has 63.23 % of probability of being the optimal search. In addition, it solves 80.32 % of the runs the most efficiently within a factor $\tau \leq 1.52$ of the best strategy. If τ ranges between $[1.52, 1.80]$ or $[4.04, 6.08]$, it is not clear which one among the strategies is the best since they keep one upping each other. However, MGAC+RST+NG+TSH surpasses the others for a factor τ between $[1.80, 4.04]$ of best strategy by solving 85.80 % to 93.22 % of runs most efficiently. It is also the best for a τ ranging between $[6.08, 12.31[$, where it is finally equaled by MGAC+RST+NG and where they solve 96.13 % of runs the most efficiently for that factor. In comparison, MGAC+RST ultimately reached 95.16 %.

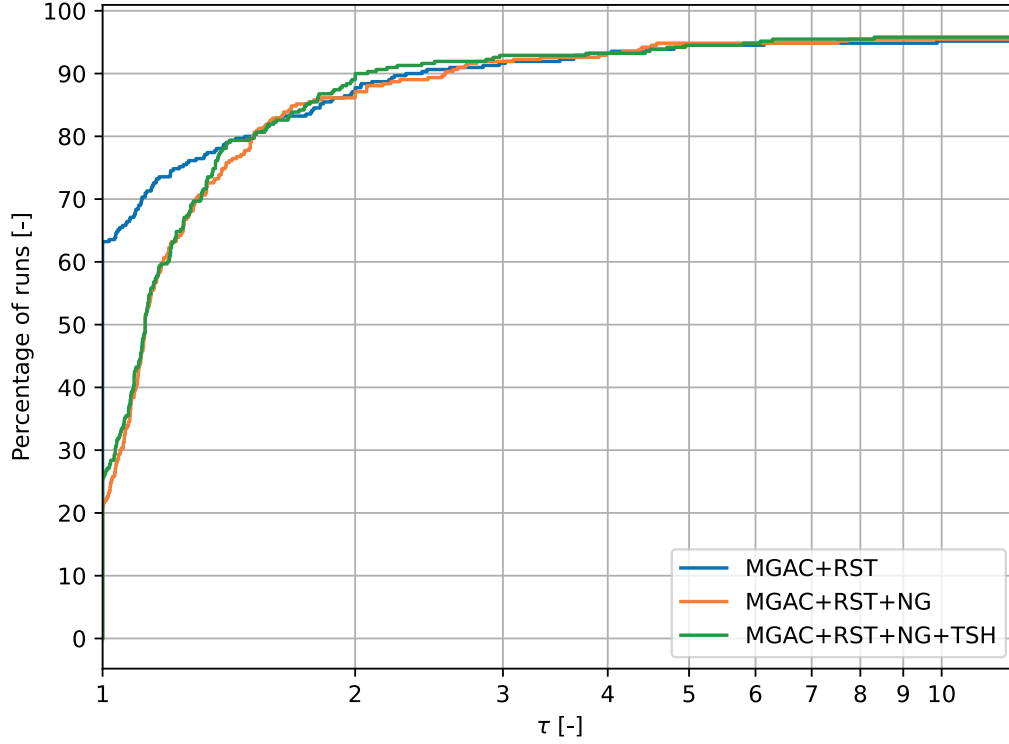


Figure 4.7: Performance profile of CPU time between MGAC+RST, MGAC+RST+NG and MGAC+RST+NG+TSH on 31 JobShop Problem instances with 10 runs each, binary split branching, restarts' failure limit of 50 and a timeout of 600 seconds.

4.2 Traveling Salesman Problem

TSP, detailed in Section 2.1.2, is used to evaluate nogoods with a LNS restart strategy. The instances used in this section originate from the TSPLIB[26, 27] and have all been converted to distance matrix format for uniformity. All strategies use the same random variable and random value ordering heuristic. Finally, as explained in Section 3.6, only the binary labeling branching type will be used with LNS in this section.

4.2.1 Large Neighborhood Search

Making a thorough performance test of different methods using LNS is a difficult task. LNS tends to fall in local optima as it follows a portion of the previous best solution, meaning that it is not the best strategy for finding the optimal solution of a problem[28]. For this reason, instead of comparing the time taken

to reach the optimal solution of an instance, comparing the best solution found within a restricted amount of time is better suited. There are also many distinct parameters that affect the search: the failure limit, the relaxation phase strategy and the percentage of solution kept. To limit these parameters, the relaxation phase strategy is fixed to a simple random variable selection.

Table 4.4 displays the statistics, averaged from 10 runs, of MGAC+LNS and MGAC+LNS+NG on 7 different instances with varying search parameters (sp) defined in Table 4.3. The size of each instance can be seen in its name as the number corresponds to the number of locations in the problem. The statistics shown are the best objective value found and the number of nodes visited to reach that objective value.

Search parameter	Percentage kept	Failure limit
sp_0	25	50
sp_1	50	10
sp_2	50	50
sp_3	50	100
sp_4	75	50
sp_5	90	50

Table 4.3: Search parameters used in Table 4.4.

As the purpose of this thesis is not to find the best search parameters for a LNS strategy given a specific problem, only the differences between MGAC+LNS and MGAC+LNS+NG on specific search parameters will be analyzed. Note that comparing the number of visited nodes is meaningful only when the achieved objective values are equal, as it corresponds to the number to reach that value³⁸. By using the same performance percentage as in Section 4.1.2 but with the objective value reached instead of the time taken, the different options can be analyzed.

The search parameters sp_0 , sp_2 , sp_4 and sp_5 share the same failure limit and differ in the percentage of solution kept. Among them, sp_0 , sp_2 , sp_4 respectively exhibit a 2.46 %, 0.3 % and 0.78 % decrease in performance when using MGAC+LNS+NG compared to MGAC+LNS. Lastly, sp_5 shows a minimal 0.01 % performance increase when using MGAC+LNS+NG instead of MGAC+LNS.

³⁸Appendix C.3.1 deals with this analysis, but the results seem irrelevant.

Instances	Statistics	MGAC+LNS					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2085	2089	2087	2087	2089	2091
	Nodes	687 k	97 574	777 k	1449 k	1013 k	51 953
tsp_26	Objective	937	937	937	937	937	945
	Nodes	1329 k	2304 k	227 k	70 544	29 581	3228 k
tsp_38	Objective	11 109	6823	6656	6656	6767	6768
	Nodes	2688 k	537 k	174 k	468 k	207 k	718 k
tsp_42	Objective	1160	709	707	706	707	721
	Nodes	11 861 k	1751 k	1925 k	2555 k	2194 k	2225 k
tsp_48	Objective	78 039	47 366	33 961	34 123	33 934	34 139
	Nodes	797 k	687 k	676 k	677 k	284 k	281 k
tsp_131	Objective	3463	3074	2972	2981	1666	596
	Nodes	6871 k	7510 k	8414 k	8345 k	10 753 k	3156 k
tsp_194	Objective	75 023	70 381	69 390	69 387	55 750	9864
	Nodes	2127 k	2585 k	2445 k	2733 k	3514 k	2573 k
Instances	Statistics	MGAC+LNS+NG					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2086	2088	2087	2088	2090	2091
	Nodes	282 k	108 k	106 k	136 k	461 k	56 692
tsp_26	Objective	942	946	937	937	937	945
	Nodes	489 k	143 k	32 567	58 946	29 334	3226 k
tsp_38	Objective	11 403	6830	6656	6712	6767	6768
	Nodes	2139 k	377 k	162 k	236 k	98 504	718 k
tsp_42	Objective	1309	708	710	706	709	721
	Nodes	3815 k	782 k	653 k	486 k	196 k	2221 k
tsp_48	Objective	78 184	47 750	34 223	34 430	33 934	34 139
	Nodes	761 k	682 k	633 k	600 k	281 k	281 k
tsp_131	Objective	3478	3081	2996	2992	1749	596
	Nodes	4660 k	6717 k	6510 k	6681 k	7956 k	3247 k
tsp_194	Objective	75 427	70 394	69 391	69 387	55 843	9859
	Nodes	1743 k	2544 k	2363 k	2733 k	3236 k	2437 k

Table 4.4: Results obtained with MGAC+LNS and MGAC+LNS+NG on 7 TSP instances with 10 runs each, 5 different search parameters, and a timeout of 300 seconds.

The search parameters sp_1 , sp_2 , and sp_3 all maintain the same percentage of solution kept while varying failure limits. Respectively, sp_1 , sp_2 , and sp_3 exhibit a decrease in performance of 0.28 %, 0.3 %, and 0.3 % when using MGAC+LNS+NG compared to MGAC+LNS. Thus the comparison in performance tends to stay the same and no substantial differences are observed in terms of the average objective value when changing the failure limit.

As the heuristic used in this analysis is fully random, a second analysis on a more sophisticated heuristic has been done in Appendix C.3.2. This heuristic is the random variable and minimal distance heuristic and shows that even though being more performant, nogoods still do not have a positive impact on it. Similarly, another test was done with a different relaxation phase but the performances of nogoods still exhibit the same pattern. The new relaxation phase consists of keeping a percentage of variables all linked together from the best solution. These values can be seen in Appendix C.3.3.

Chapter 5

Discussion and Perspectives

This chapter discusses the results of experiments carried out in Chapter 4, the improvements as well as the perspectives on what could be done on the basis of this thesis. These topics are covered in Sections 5.1, 5.2 and 5.3, respectively.

5.1 Discussion

The objective of Chapter 4 is to evaluate how the usage of nogoods, limited or not in size, influences various restart strategies, namely search with restarts and large neighborhood search, within MiniCP. Therefore the goal is neither to compare these strategies together, nor to use the best ordering heuristics, but to highlight the impact of nogoods on the different strategies.

Note that despite some efforts, the results may still be influenced in some way. Indeed, the tested instances may not be entirely representative of the problems evaluated. Also, among the measurements of a few instances, the resolution time is sometimes less than a second, and can therefore be affected by the state of the CPU.

5.1.1 Restarts

The statistics shown in Section 4.1.2 demonstrate that nogoods can have a positive impact on a basic restart strategy for some problems. A restart strategy with learning is only better if the overhead cost associated with storing and utilizing the nogoods is outweighed by the time gained through inferences³⁹. However, many different factors affect this result such as different problems, heuristics, stopping

³⁹Applicable on a large enough sample size since randomness still comes into play when inferences generate divergences in the path.

criteria within restarts, the nogoods generated, and the branching type. Even though they are all linked together, they will individually be discussed.

The first factor is the problem instance itself as it is directly correlated to the resolution time. As seen in Figure 4.2 or also C.1 and C.2, the usefulness of nogoods varies based on resolution time, which is related to the number of restarts, within a certain factor of error, since the stopping criterion is fixed throughout the restarts. From these Figures, three interesting points can be deduced. When the instance is solved within too few restarts, there are not enough nogoods to make many inferences which leads to a bad performance, since the overhead of the nogood base is present. When the instance takes a large number of restarts to find the best objective value, it means a large search tree and thus fewer possibilities for repetition. So even though many nogoods are stored in the base from the previous restarts, they do not make enough inferences to have a significant impact. When the instances are within these two areas, it is kind of a sweet spot where the nogoods stored are able to make enough inferences to have an increase in performance.

Different heuristics have the potential to guide the search in different ways but remain the same for basic restarts with and without nogoods. They have different complexities which affect the search as each non-leaf node has to use it to generate the next branches. In addition to that, when nogoods are used in the search, it adds a bit more computation time to each node because of watched decision updates or inferences being made. Therefore, when adding the two together, the time gained by not reexploring subtrees has to be greater than the extra computation time of nogoods for nogoods to have a chance at being beneficial. However, the heuristic used for the tests in Section 4.1.2 is a simple and quick random variable and minimal value selection and still showed improvements when adding the learning overhead to the search. Thus, complex heuristics may guide the search better and also show a more significant improvement.

The stopping criterion is another important factor since this parameter drastically changes the average performance of nogoods on a problem instance. When the restarts are executed too quickly, there will be more nogoods generated which will be able to make inferences more often. However, these inferences will not have a big impact since the search would not have stayed in that search space for long due to the quick restart. On the contrary, when the restarts do not happen often enough, even though the search space skipped through inferences would be impactful, there is not a sufficient number of nogoods to make such inferences often. Finally, when falling in between these two cases, the performance of nogoods can be most noticeable. Section 4.1.2 demonstrates this phenomenon with various

stopping criteria in the form of failure limits and shows that on average, a stopping criterion of 50 performs best with nogoods.

The penultimate factor that can have an impact is the determination of which nogoods are generated and kept. As seen in Section 4.1.1 different nogood sizes have different impact on the search. Thus, adding a threshold in size, such as keeping around 99 % of inferences on average, removes the useless computation time of nogoods too big to use. Although the determination of this threshold was made on a single profiling run for each instance, and so not totally accurate for all runs, the increase in performance is noticeable as seen in Figure 4.4.

Lastly, the type of branching significantly influences the effectiveness of nogoods, as the search tree is completely different and the decisions impact the nogood base differently. When comparing the binary labeling discussed in Section 4.1.2 with the binary split outlined in Section 4.1.3, there were substantial differences in the inferences made. These differences highlighted the reasons behind the poor performance of the second branching strategy. The same conclusion on nogood performance can also be drawn from the performance profiles of each branching method, as seen in Figures 4.4 and 4.7.

One final note on the binary split branching’s low inference number can be made. It happens mainly due to the fact that, in this case, the decisions composing a nogood constraint do not take into account the restriction on the right bound provided by the positive decisions sharing the same variable on the according nld-subsequence. Therefore, this missing restriction prevents a decision on a given variable to be falsified if the upper nogoods are not already entailed thanks to this variable. For the same reason, inferences have less pruning capability since no restriction is performed on the right bound.

5.1.2 Large Neighborhood Search

The results from Section 4.2 do not demonstrate a positive impact of nogoods on the average performance of the TSP instances tested with a LNS approach. Since this potential impact of nogoods on LNS might be influenced by various parameters, such as the percentage of retained solutions or the failure limit, the analysis was conducted using different pairs of search parameters *sp* outlined in Table 4.3. Additionally, the relaxation phase of LNS, which has not changed the negative impact of nogoods, was arbitrarily fixed to a random selection of variables, with the same randomness whether nogoods were used or not. However, maintaining the same randomness does not imply that the relaxation phase will yield identical outcomes, as the solutions upon which it operates could differ across restarts.

Whether it is the kept percentage of solution found which varies among 25 %, 50 %, 75 % or the failure limit which changes, LNS with nogoods tends to find less interesting optimal values. A percentage of 90 % is the only one showing positive results for nogoods. It is important to bear in mind that either the gain or the loss in performance is mostly within the range of tenths, which is not that significant.

The inferior average results when compared to those in Section 4.1 can be attributed to the fundamental distinction between a basic restart strategy and a LNS approach. Indeed, as long as the best solution found during the search gets better, the nogoods generated during previous neighborhood searches will become less relevant. Furthermore, in the case of LNS, since a certain percentage of the best solution found so far is kept, the minimal size of the nogoods generated after the initial solution discovery is directly proportional to the instance dimension. However, lengthy nogoods prove to be ineffective since they will at most make inferences in rare cases to avoid small subtrees. Unfortunately, unlike a regular restart strategy, LNS cannot use nogoods with a threshold size to suppress this phenomenon. This is due to the fact that the threshold size would need to be at least as large as the smallest nogood generated during the neighborhood search phase. Thus having smaller nogoods to improve their impact can only be achieved by using a small percentage of solution kept, which tends to act like regular restarts.

5.2 Improvements

The implementation of nogoods in MiniCP has already been performed in 2022[2]. However, it did not use the base $\mathcal{B}$ to store nogoods as described by Lecoutre in [3]. Instead, each generated nogood was either reduced or minimized, with their corresponding nogood constraint stored in a list, as depicted in Section 2.4.3, and posted as an `Or` constraint. Nonetheless, the objective of the paper is to stick to Lecoutre’s modeling and implement its base $\mathcal{B}$ in MiniCP. Compared with the implementation done in 2022, there are three main differences. The first is the `Or` constraint which uses two `StateInt` objects to indicate the indices of watched decisions. These act as bounds, moving towards the center and ensure that only the decisions within these are watchable. As they are stateful objects, they are restored upon backtracks or restarts, unlike Lecoutre’s approach. The second difference is that creating an `Or` constraint is costly. For every decision in a nogood constraint, a `BoolVar` is instantiated. This `BoolVar` is an extension of an `IntVar` meaning it has most of its attributes, including a domain and stacks of constraints that are stateful objects themselves. The final difference follows the previous one. With `BoolVar`, a domain listener is added for each watched decision, even if two `BoolVar`

represent the same underlying decision. This greatly contrasts with the behavior of the base $\mathcal{B}$ as there is only one listener per variable. Overall, while the impact of stateful objects should be lessened with a trailing state management strategy and that only watched decisions have the constraint in one of their stacks, the process still involves a substantial number of object instantiations to only add one nogood.

Additionally, the thesis [2] evaluated the impact of nogoods over a regular restart strategy, by measuring both the time taken and the number of nodes visited by the strategies in a controlled environment, where the search tree remains the same between the strategies for every restart. The idea is to execute the regular restart strategy and save all branches of the explored search trees and then replay both strategies on these search trees already fixed. If the nogood approach reached an unexplored section of the search tree during a restart, it simply stops there and restarts. This method enables for an *educational analysis* by eliminating differences in the explored search spaces. However, it does suffer from the limitation that the time spent by the heuristic is not reproduced during the replay. Indeed, in addition to avoiding a subtree already explored, nogoods make it possible to avoid the cost of the heuristics that will be applied to each non-leaf node of this subtree.

In contrast, the method outlined in Chapter 4 evaluates the influence of nogoods by comparing strategies with and without nogoods under the same circumstances. This evaluation considers the time required to achieve an optimal solution or the best objective value found within a specified time frame, while also recording the number of nodes visited to reach the optimal solution. Although they are subject to the same circumstances in terms of parameters or randomness seeds, they may differ in the search spaces explored, since randomness can affect the results upon inferences, potentially leading to inaccurate comparisons. To smooth the impact of randomness, multiple runs were performed for each instance. This evaluation method aligns more closely with a *real case analysis* since it compares the strategies as a user would compare them, with identical parameters and seeds, but not by forcing them to act the same after inferences.

Nevertheless, it would be interesting to conduct the first evaluation method while also considering heuristic execution time to truly and fairly compare the strategies.

In Section 3.1.1, it was explained that the `Decision` object does not explicitly specify the action taken on the variable and value. Instead, this information is embedded within the implementation, as each branching type is handled by a distinct class to enhance performance. With hindsight, given that MiniCP is primarily intended for educational purposes rather than competitive performance,

a more interesting approach might involve designing the `Decision` object to incorporate the type of action performed. This adjustment would enable the use of a single class to manage the base structure with generalized methods. Moreover, a generalized version of `Decision` would allow nogoods to have decisions of different types of actions that could be useful for conducting LNS with non-binary labeling branchings.

5.3 Perspectives

First of all, one could evaluate the impact of modeling watched decisions as `StateInt` as done by the `Or` constraint compared to Lecoutre’s approach. The goal is to see if the cost of stateful objects is outweighed by the reduction in complexity when updating watched decisions.

Remaining on the implementation aspect, further filtering can be done on nogoods at the beginning of a run following a single nogood inference, as seen in Section 2.4.4. At the start of the search, the entailed nogood constraints can be removed from the base and the falsified decisions can be removed from nogoods, since none of these will ever be useful again. Despite this possibility, the choice was made to not waste time removing these elements. This was motivated by the fact that these deletions need to browse all nogoods looking for satisfied and falsified decisions which is costly. Moreover entailed nogoods with satisfied decisions will tend to be detected quickly during the upcoming restarts since the watched decisions will cluster around these. The hypothesis that the deletions will have a negative impact has not been evaluated and remains unproven. Thus, assessing it would be interesting.

Another avenue of exploration would consist in studying the compatibility of other heuristics with nogoods and examining the changes in performances in MiniCP. Additionally, the relaxation strategy of LNS has not been proven to alter the overall impact of nogoods, despite the two relaxations tested. Thus more relaxation strategies could be tested to see if one could perform better when used with nogoods or prove that they do not influence the impact of nogoods.

Finally, while functional in the implementation, a binary split used for example in a binary search manner, will need to branch on the same variables multiple times before reaching the first leaf node. Because of this effect, the search tree becomes deep and nogoods are less interesting in general. However it might be interesting to explore a mix of the different branchings for the JobShop Problem as this could give different results.

Chapter 6

Conclusion

The two objectives of this master’s thesis were to integrate nogoods in MiniCP following the framework of Lecoutre and to evaluate their potential over different restart strategies.

While the first is successfully achieved, the implementation differs slightly from the approach of Lecoutre on one aspect: watched decisions becoming falsified are not relocated if the nogood is already entailed by an unwatched decision. This avoids unnecessary swaps of watched decisions. In addition, the base of nogoods has been designed to handle binary split branching, which has necessitated adjustments in the way falsified decisions are detected and in the way nogoods are generated. The implementation has also been designed to facilitate future branching implementations and allow users to easily make use of nogoods with different options, such as their threshold size or the branching type applied.

The second objective has also been achieved with a focus on evaluating reduced nld-nogoods, limited or not in size by a threshold, on various search strategies. Since these variations behaved differently with nogoods, they were analyzed independently.

The experiments have shown that when the JobShop Problem is solved by a regular restart strategy with a binary labeling branching, the use of reduced nld-nogoods outperforms the regular strategy on average. Moreover, reduced nld-nogoods limited in length with a suitable threshold perform the best. However, nogoods can still have a negative contribution on particular instances. These include small and large instances where nogoods cannot make enough inferences. This is either because of a low number of nogoods or because the search tree is too large for frequent repetitions to occur.

When a binary split branching is used instead, minimized reduced nld-nogoods have not been shown to contribute positively on average to the regular restart strategy. While this branching performs better than the binary labeling one on the JobShop Problem, the number of inferences is very low in comparison and they also have less pruning capability than expected since no restriction is performed on the right bound of the variable domains.

The results show that the contribution of nogoods on a LNS, with binary labeling branching, to solve the TSP is not positive on average. Those results can be attributed to the fundamental nature of LNS. Indeed, as long as the best solution found during the search gets better, the nogoods generated during previous neighborhood searches will become less relevant. Also, since a certain percentage of the best solution found so far is kept, the minimal size of the nogoods generated after the initial solution discovery is directly proportional to the instance dimension. This generally results in lengthy nogoods which are ineffective. In addition, nogoods cannot be limited in size with a threshold inferior to the number of variables that remain assigned during relaxation.

In the end, these results depend on the parameters tested. Randomness directly influences the performance of different strategies, both with and without the incorporation of nogoods, which is why measures were multiplied. Moreover, research in this field remains endless since the effectiveness of nogoods may vary depending on the heuristic used. We hope that this work will encourage further exploration in the usage of nogoods in MiniCP.

Appendix A

Alternative Version of Nogood Generation

In the case where the stopping criterion to restart is a failure limit, the last state visited has no possible solution and thus, Algorithm 3.1 can be adapted in order to avoid re-exploring the last states that were not outcast. But there is more since, in this case, if the last decision is negative, it means that it's also the case for the opposite decision performed previously in the search's branching order. So negative decisions ending the list are therefore useless since shorter nogoods generated before will include them. Ignoring them therefore avoids creating longer and unnecessary nogoods. That is what the first for loop does in Algorithm A.1 by considering the last index as the one of the last positive decision in the list.

The second loop being exactly the same as in Algorithm 3.1, its explanation can be found in Section 3.2. However, for the last decision, which is positive, the based nogood is also stored once increased since it is the end of the loop. This is only allowed since the possible negative decisions that were removed from sight have necessarily no remaining unexplored solutions. Hence, it is safe to prohibit all deeper explored states.

Algorithm A.1: Generate nogoods from Decisions with a failure limit
as stopping criterion of restart

Input:

decisions // list of Decision of last branch
depth // level before restart; root is zero

begin

nogood $\leftarrow$ *List*()
 last $\leftarrow$ *depth* - 1 // index of last positive decision
 for *last* **to** 0 **do**
 if *decisions*[*last*].*assign* = *True* **then**
 break
 end
 end
 for *i* $\leftarrow$ 0 **to** *last* **do**
 if *decisions*[*i*].*assign* = *True* **then**
 nogood.add(decisions[i])
 else
 new $\leftarrow$ *Copy*(*nogood*)
 new.add(decisions[i])
 StoreNogood(new)
 end
 end
 StoreNogood(nogood)

end

Appendix B

Minimizing Randomness for Nogood Comparison

Listing B.1 shows a small example code that uses different `Random` objects and seeds to maximize the path similarity when using nogoods or not. The problem solved in this case is TSP with restarts, using a random variable and minimal value heuristic, with a failure limit of 50 and no time limit.

Line 8 creates a `Solver` which uses nogoods and can be downgraded to a normal restart search by removing the second parameter. Lines 19 and 20 initialise the two `Random` objects that will be used. The `makeDFS` on line 27 makes use of the `randomBranching` object only for its variable and value selection. Then for each restart in the loop of line 55 the `randomBranching` seed is renewed via the `randomRestart` object. This way all restarts will be different but in the same order, so reproducibility is possible even when inferences are made in a restart or not.

```
1 // -----Reading the data-----
2 InputReader reader = new InputReader(file);
3 int n = reader.getInt();
4 int opti = reader.getInt();
5 int [][] distanceMatrix = reader.getMatrix(n, n);
6
7 // -----Starting solver and posting constraints -----
8 Solver cp = makeSolver(false, "equal");
9 IntVar[] succ = makeIntVarArray(cp, n, n);
10 IntVar[] distSucc = makeIntVarArray(cp, n, 10000);
11 cp.post(new Circuit(succ));
12 for (int i = 0; i < n; i++) {
13     cp.post(new Element1D(distanceMatrix[i], succ[i], distSucc[i]));
14 }
15 IntVar totalDist = sum(distSucc);
```

```

16 Objective obj = cp.minimize(totalDist);
17
18 // ----- Initializing randoms -----
19 Random randomRestart = new Random(42);
20 Random randomBranching = new Random();
21
22 // ----- Adding variables to nogoods if applicable -----
23 if(cp.getNogoodConstraint() != null)
24     cp.getNogoodConstraint().addVariable(succ);
25
26 // -----makeDFS unsing RandomBranching -----
27 DFSearch dfs = makeDfs(cp, () -> {
28
29     // Random variable selection
30     IntVar xs = null;
31     int index = randomBranching.nextInt(n);
32     for (int i = (index + 1 < n) ? index + 1 : 0; i != index; i =
33         (i + 1 < n) ? i + 1 : 0) {
34         if (succ[i].size() > 1) {
35             xs = succ[i];
36             break;
37         }
38     }
39     if (xs == null) return new Procedure[0];
40
41     // min value selection
42     int v = xs.min();
43
44     IntVar finalXs = xs;
45     return branch(() -> cp.post(equal(finalXs, v)),
46         () -> cp.post(notEqual(finalXs, v)));
47 });
48
49 // ----- Set on solution breaking point -----
50 AtomicInteger best = new AtomicInteger(Integer.MAX_VALUE);
51 dfs.onSolution(() -> {
52     best.set(totalDist.min());
53 });
54
55 // ----- Initiate restart -----
56 while (best.get() > opti) {
57     randomBranching.setSeed(randomRestart.nextInt());
58     dfs.optimize(obj, statistics -> statistics.numberOfFailures()
59         >= 50);
60 }

```

Listing B.1: Example code to demonstrate the usage for Random objects to maximise path similarity between restart strategies using nogoods or not.

Appendix C

Additional Benchmarks

C.1 Graph Coloring Problem

Table C.1 shows the computation time and number of nodes visited of search variations to reach a solution on 5 Graph Coloring Problem instances. A solution is reached once the search has managed to color the nodes with the minimum known coloring number possible for that instance. Each value in the table is an average of 5 runs to limit randomness factors. The failure limit of restarts is fixed to 50 and the search is limited to 600 seconds. If all runs timeout for a specific instance and search, TO is used. The instances come from GCPLIB[29] and the size respectively show the vertices and edges.

Instances	Size	Statistics	MGAC		
				+RST	
					+NG
3-Insertions_3	56 x 110	Time [s]	0.0	0.0	0.0
		Nodes	54	54	54
huck	74 x 602	Time [s]	0.0	0.0	0.0
		Nodes	71	71	71
myciel7	191 x 2360	Time [s]	0.0	0.0	0.0
		Nodes	172	169	169
2-Insertions_4	194 x 541	Time [s]	TO	TO	TO
		Nodes	355 M	391 M	4924 k
queen11_11	121 x 3960	Time [s]	TO	TO	TO
		Nodes	32 688 k	41 664 k	8494 k

Table C.1: Results obtained with MGAC, MGAC+RST and MGAC+RST+NG on 5 Graph Coloring Problem instances with 5 runs each, binary labeling branching, restarts' failure limit of 50 and a timeout of 600 seconds.

C.2 JobShop Problem

Figures C.1 and C.2 are pairwise comparisons on CPU time between MGAC+RST and MGAC+RST+NG⁴⁰ on 31 JobShop instances from JSPLIB[21] with 10 runs each. Unlike in Section 4.1.2, the failure limits here are of 10 and 100.

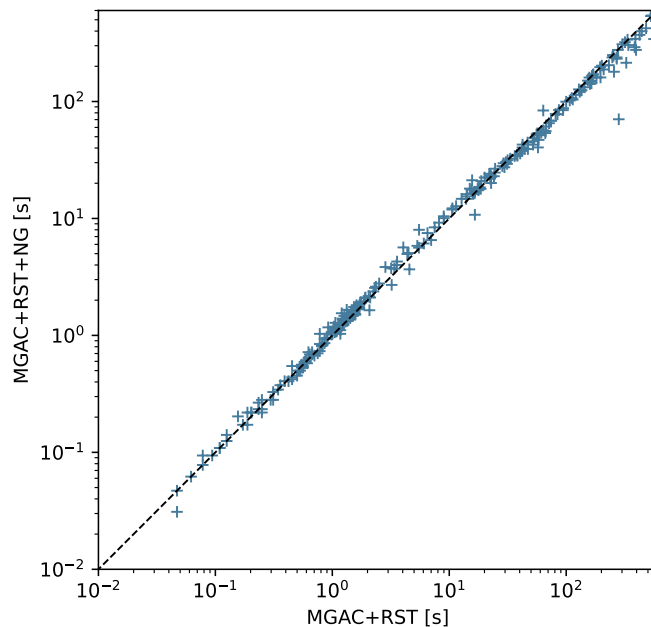


Figure C.1: Pairwise comparison of CPU time between MGAC+RST and MGAC+RST+NG on 31 JobShop instances with 10 runs each, binary labeling branching, restarts' failure limit of 10 and a timeout of 600 seconds.

⁴⁰MGAC+RST and MGAC+RST+NG have been defined in Chapter 4.

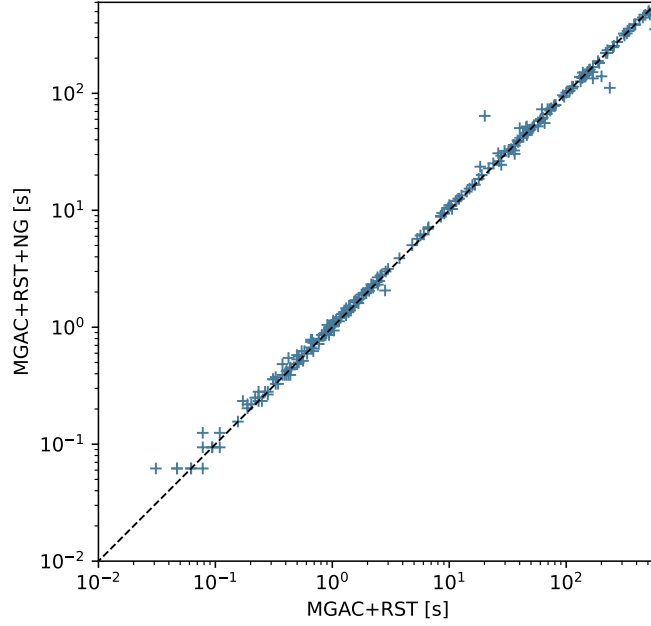


Figure C.2: Pairwise comparison of CPU time between MGAC+RST and MGAC+RST+NG on 31 JobShop instances with 1 runs each, binary labeling branching, a restarts' failure limit of 100 and a timeout of 600 seconds.

C.3 Traveling Salesman Problem

C.3.1 Node Analysis

As discussed in Section 4.2.1 with the results of Table 4.4, comparing the number of nodes visited is only meaningful when the achieved objective values are equal, as it corresponds to the number to reach that value. By adapting the performance percentage described in Section 4.1.2 to take the number of nodes explored up to a same final objective value instead of time and by doing a geometric mean instead of an arithmetic mean, as they are exponential in nature, such an analysis can be handled.

It appears that, MGAC+LNS performs better than MGAC+LNS+NG for sp_1 , sp_4 and sp_5 by 33.48 %, 4.09 % and 0.37 % respectively. However, MGAC+LNS is worse than MGAC+LNS+NG for sp_0 , sp_2 and sp_3 by 7.95 %, 18.93 % and 26.53 % respectively. Although the average is in favor of MGAC+LNS+NG, the results are so disparate as to be irrelevant.

C.3.2 Alternative Heuristic

Table C.2 shows the statistics, averaged from 10 runs, of MGAC+LNS+NG and MGAC+LNS on 7 different instances with varying search parameters (sp) defined in Table 4.3. The statistics shown are the best objective value found within a time limit of 300 seconds and the number of nodes visited to reach that objective value. All strategies use the same random variable and random distance heuristics and the relaxation phase strategy is a simple random variable selection. The size of each instance can be seen in its name as the number corresponds to the number of locations in the problem.

C.3.3 Alternative Relaxation Phase

Table C.3 shows the statistics, averaged from 5 runs, of MGAC+LNS+NG and MGAC+LNS on 7 different instances with varying search parameters (sp) defined in Table 4.3. The statistics shown are the best objective value found within a time limit of 300 seconds and the number of nodes visited to reach that objective value. All strategies use the same random variable and random value heuristics. The relaxation phase varies from the other tests though. For these tests, the variables kept remain linked together to form a continues path. The size of each instance can be seen in its name as the number corresponds to the number of locations in the problem.

Instances	Statistics	MGAC+LNS					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2085	2085	2085	2085	2087	2094
	Nodes	5156	151 k	2508	7398	5357	59 129
tsp_26	Objective	937	937	937	937	937	949
	Nodes	14 345	7646	111 k	23 035	9414	1856 k
tsp_38	Objective	6656	6656	6656	6656	6878	6878
	Nodes	49 008	361 k	219 k	104 k	26 678	726 k
tsp_42	Objective	699	705	705	702	705	710
	Nodes	1033 k	15 623	180 k	3097 k	72 874	2564 k
tsp_48	Objective	33 812	33 722	33 722	33 646	34 183	34 267
	Nodes	792 k	349 k	324 k	435 k	80 833	136 k
tsp_131	Objective	658	589	590	581	584	594
	Nodes	5438 k	5725 k	6474 k	6080 k	3385 k	2063 k
tsp_194	Objective	11 747	11 845	11 627	11 879	9753	9763
	Nodes	1369 k	3052 k	2958 k	3073 k	1882 k	1865 k
Instances	Statistics	MGAC+LNS+NG					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2085	2085	2085	2085	2087	2094
	Nodes	5145	99 567	60 824	7073	5301	59 128
tsp_26	Objective	937	937	937	937	937	949
	Nodes	14 023	7604	100 k	22 347	9316	1853 k
tsp_38	Objective	6656	6767	6656	6656	6878	6878
	Nodes	49 002	112 k	429 k	102 k	26 519	725 k
tsp_42	Objective	700	705	705	707	705	710
	Nodes	570 k	15 599	178 k	78 142	72 186	2563 k
tsp_48	Objective	33 851	33 744	33 722	33 646	34 183	34 267
	Nodes	761 k	172 k	272 k	434 k	80 437	136 k
tsp_131	Objective	660	589	592	583	585	594
	Nodes	3943 k	5725 k	5483 k	5322 k	1747 k	2062 k
tsp_194	Objective	11 747	11 924	11 720	11 886	9759	9763
	Nodes	1369 k	2739 k	2820 k	2965 k	1561 k	1865 k

Table C.2: Results obtained with MGAC+LNS and MGAC+LNS+NG on 7 TSP instances with 10 runs each, 5 different search parameters, a timeout of 300 seconds, and a random variable and minimal distance heuristics.

Instances	Statistics	MGAC+LNS					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2087	2087	2089	2086	2637	4613
	Nodes	95 419	12 094	12 790	9984	767	0
tsp_26	Objective	955	941	944	951	1193	2138
	Nodes	10 430 k	439 k	65 438	236 k	6094	2345
tsp_38	Objective	12 318	7675	6708	6669	9796	18 340
	Nodes	3474 k	3567 k	3036 k	4584 k	115 k	4416
tsp_42	Objective	1310	866	737	736	1026	1888
	Nodes	11 569 k	13 189 k	12 580 k	14 623 k	115 k	5364
tsp_48	Objective	73 590	56 428	48 198	46 021	46 921	86 054
	Nodes	1094 k	903 k	1228 k	1404 k	226 k	6851
tsp_131	Objective	3484	3073	3078	3069	2089	1209
	Nodes	6025 k	5332 k	7453 k	9015 k	10 784 k	8127 k
tsp_194	Objective	76 239	70 704	70 188	69 872	59 142	29 029
	Nodes	1264 k	2416 k	3212 k	3049 k	3549 k	4350 k
Instances	Statistics	MGAC+LNS+NG					
		sp_0	sp_1	sp_2	sp_3	sp_4	sp_5
tsp_17	Objective	2087	2090	2089	2087	2637	4613
	Nodes	81 551	15 492	10 506	5916	767	0
tsp_26	Objective	1057	951	941	947	1193	2138
	Nodes	1946 k	345 k	104 k	51 519	5973	2345
tsp_38	Objective	12 683	8284	6864	6863	10 346	18 340
	Nodes	2131 k	2282 k	2423 k	2261 k	19 803	4416
tsp_42	Objective	1411	940	793	756	1051	1888
	Nodes	3374 k	3694 k	3558 k	4409 k	53 117	5364
tsp_48	Objective	73 628	55 824	49 452	47 223	46 414	86 054
	Nodes	1069 k	911 k	1220 k	1314 k	219 k	6851
tsp_131	Objective	3515	3076	3099	3098	2135	1217
	Nodes	4292 k	5226 k	5662 k	6143 k	7728 k	4817 k
tsp_194	Objective	76 239	70 774	70 580	69 947	59 324	28 800
	Nodes	1264 k	2307 k	2252 k	2959 k	3355 k	4088 k

Table C.3: Results obtained with MGAC+LNS and MGAC+LNS+NG on 7 TSP instances with 5 runs each, 5 different search parameters, a timeout of 300 seconds, and a relaxation phase strategy that keeps variables linked together.

Bibliography

- [1] L. Michel, P. Schaus, and P. Van Hentenryck. “MiniCP: a lightweight solver for constraint programming”. In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184. DOI: 10.1007/s12532-020-00190-7.
- [2] M. Skjølnes. “Recording and Minimising Nogoods from Restarts in the Solver MiniCP”. PhD thesis. 2022. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-485982>.
- [3] C. Lecoutre. *Constraint Networks: Techniques and Algorithms*. Jan. 2009, pp. 137–150, 431–457. DOI: 10.1002/9780470611821.
- [4] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. USA: Elsevier Science Inc., 2006, pp. 1–83. ISBN: 9780080463803.
- [5] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010, pp. 203–220.
- [6] Google. *The Job Shop Problem*. Apr. 2023. URL: https://developers.google.com/optimization/scheduling/job_shop.
- [7] J. D. C. Little et al. “An Algorithm for the Traveling Salesman Problem”. In: *Operations Research* 11.6 (1963), pp. 972–989. DOI: 10.1287/opre.11.6.972.
- [8] C. Dejemeppe. “Constraint programming algorithms and models for scheduling applications.” PhD thesis. Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2016.
- [9] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003, pp. 54–80. DOI: 10.1017/CB09780511615320.
- [10] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011, pp. 518–536. ISBN: 978-0-321-57351-3.
- [11] Carla Gomes et al. “Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems”. In: *Journal of Automated Reasoning* 24 (Jan. 2000). DOI: 10.1023/A:1006314320276.
- [12] C. Lecoutre et al. “Recording and Minimizing Nogoods from Restarts”. In: *JSAT* 1 (May 2007), pp. 147–167. DOI: 10.3233/SAT190009.

- [13] A. Fukunaga. “Complete restart strategies using a compact representation of the explored search space”. In: (Jan. 2003).
- [14] R. Stallman and G. Sussman. “Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis”. In: *Artificial Intelligence* 9.2 (Aug. 1977), pp. 135–196. DOI: 10.1016/0004-3702(77)90029-7.
- [15] G. L. jr. Steele. “The Definition and Implementation of a Computer Programming Language Based Constraints”. In: *MIT - AITR-595* (Oct. 2004).
- [16] R. Dechter. “Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition”. In: *Artificial Intelligence* 41 (Jan. 1990), pp. 273–312. DOI: 10.1016/0004-3702(90)90046-3.
- [17] L. Baptista, I. Lynce, and J. Marques-silva. “Complete Search Restart Strategies for Satisfiability”. In: (Nov. 2001).
- [18] C. Lecoutre et al. “Nogood Recording from Restarts”. In: *IJCAI*. Jan. 2007, pp. 131–136.
- [19] M. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: (May 2001), pp. 530–535. DOI: 10.1145/378239.379017.
- [20] J. Gosling et al. *The Java Language Specification: Java SE 8 Edition*. 5th ed. Java Series. Addison-Wesley, 2014. ISBN: 978-0-13-390069-9.
- [21] Y. Tamura. *JSPLIB*. Feb. 2023. URL: <https://github.com/tamy0612/JSPLIB>.
- [22] J. Adams, E. Balas, and D. Zawack. “The Shifting Bottleneck Procedure for Job Shop Scheduling”. In: *Management Science* 34.3 (1988), pp. 391–401. DOI: 10.1287/mnsc.34.3.391.
- [23] S. Lawrence. “Resouce constrained project scheduling : an experimental investigation of heuristic scheduling techniques (Supplement)”. In: *Graduate School of Industrial Administration, Carnegie-Mellon University* (1984). URL: <https://cir.nii.ac.jp/crid/1571980073974705920>.
- [24] D. Applegate and W. Cook. “A Computational Study of the Job-Shop Scheduling Problem”. In: *INFORMS Journal on Computing* 3 (May 1991), pp. 149–156. DOI: 10.1287/ijoc.3.2.149.
- [25] E.. Dolan and J. Moré. “Benchmarking Optimization Software with Performance Profiles”. In: *Mathematical Programming* 91 (Mar. 2001), pp. 201–213. DOI: 10.1007/s101070100263.
- [26] J. Burkardt. *TSP - Data for the Traveling Salesperson Problem*. Apr. 2023. URL: <https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html>.

- [27] W. Cook et al. *TSP Test Data*. Apr. 2023. URL: <https://www.math.uwaterloo.ca/tsp/data/index.html>.
- [28] T. Friedrich et al. “Escaping Local Optima with Local Search: A Theory-Driven Discussion”. In: *Parallel Problem Solving from Nature – PPSN XVII*. Ed. by G. Rudolph et al. Cham: Springer International Publishing, 2022, pp. 442–455. ISBN: 978-3-031-14721-0.
- [29] M. Trick. *GCPLIB*. Mar. 2023. URL: <https://mat.gsia.cmu.edu/COLOR04/>.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl