UNIVERSITY OF GRONINGEN

BACHELOR THESIS

Multi-Core Solver for Discrete Finite-Domain Constraint Satisfaction Problems

Author:
Erblin IBRAHIMI

Supervisor:
dr. Arnold MEIJSTER
dr. Malvin GATTINGER

A thesis submitted in fulfilment of the requirements for the Bachelor degree in Computing Science

UNIVERSITY OF GRONINGEN

Abstract

Faculty of Science and Engineering
Computing Science

Bachelor of Science

Multi-Core Solver for Discrete Finite-Domain Constraint Satisfaction Problems

by Erblin IBRAHIMI

This thesis presents an implementation of a general-purpose, multi-core solver for finite-domain Constraint Satisfaction Problems (CSPs) that runs efficiently on all multi-core machines, including personal computers. This solver reads input CSPs from a small, custom-designed specification language that provides the user a flexible way of specifying CSPs. The first part of the thesis covers some of the most popular example CSPs and how the backtracking search algorithm, including heuristics and inferences, can be used to solve CSPs. Next, we look at how this algorithm can be parallelized by starting the searching process on multiple threads, each starting at a different variable. This approach leads to duplicate searches between threads, however. We discuss how custom search trees can be used to store partial states that have already been covered to avoid these duplicate searches, and what the drawbacks are of using these trees. We then look at the specification language itself and what features are implemented. We will show how the aforementioned example CSPs can be represented in this language, followed by a section on how the CSP solver works under the hood. Lastly, we analyse the performance of the solver, using different settings for each example CSP, and conclude that there is a speed-up for the majority of these CSPs.

Contents

Chapter 1

Introduction

We implemented a multi-core solver for finite-domain Constraint Satisfaction Problems. We designed a small specification language that allows the user to specify these problems in a flexible way. This research results in a general-purpose solver that will run efficiently on all multi-core machines, including personal computers.

1.1 Definition of a CSP

A *discrete finite domain Constraint Satisfaction Problem* is a tuple (X, D, C) where:

- X is a finite set of variables, $\{X_1, \ldots, X_n\}$.
- D is a set of discrete finite domains, $\{D_1, \ldots, D_n\}$, one for each variable. The set D_i consists of a set of permissible values for variable X_i.
- C is a set of constraints on the variables of X, where each constraint is a logical expression, or a built-in constraint.

A *solution* of a CSP is an assignment of a value taken from D_i for each variable $X_i \in X$ such that all the constraints in C are satisfied.

We define a *state* as a set of assignments of values to some or all of the variables of the CSP. A *partial state* is a state in which some variables have not been assigned a value (yet), while a *complete state* is a state in which all variables have been assigned a value. A *state* S is a *substate* of a state T if the set of variables of S is a subset of the variables of T, and each of the overlapping variables in S and T has been assigned the same value. A state, either partial or complete, that does not violate any of the constraints is called a *consistent* state, otherwise, it is called an *inconsistent state*. Therefore, we can also define a *solution* as a complete state that is consistent.

1.2 Examples CSPs

In this section, we give a few concrete examples of CSPs. They are given to make the reader familiar with the concept of a CSP, but they are also used as reference examples later in this document.

1.2.1 N-queens problem

A famous CSP is the eight-queens puzzle, which was first published in 1848 by chess composer Max Bezzel [2].

The objective is to place eight queens on a chessboard such that no queen attacks any other queen. Later this puzzle was generalised to $n \times n$ chessboards, and is now known as the n-queens problem. It can be formulated as a CSP as follows:

- $X = \{X_1, \ldots, X_n\}$, where each X_i is the column of the queen that is placed on row i of the chessboard.
- For each X_i we have $D_i = \{1, \ldots, n\}$ (i.e. column numbers).
- $C = \textit{AllDiff}(X_1, \ldots, X_n) \cup \{j - i \neq abs(X_i - X_j) \mid 1 \leq i < j \leq n\}$.

Here $abs(x)$ denotes the absolute value of x. Moreover, we also made use of the *AllDiff* constraint here, which is a constraint that takes an arbitrary number of variables. This constraint says that all variables involved must have a different value, and is helpful in specifying multiple constraints as a single expression. For example, $\textit{AllDiff}(X_1, X_2, X_3, X_4)$ is a compact notation for the set of constraints $\{X_1 \neq X_2, X_1 \neq X_3, X_1 \neq X_4, X_2 \neq X_3, X_2 \neq X_4, X_3 \neq X_4\}$.

A solution of the eight-queens problem is $[X_1, .., X_8] = [5, 7, 2, 6, 3, 1, 4, 8]$, which is shown in Figure 1.1.

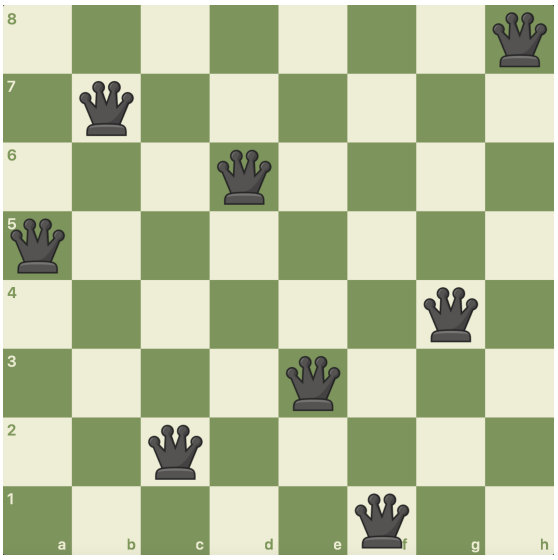
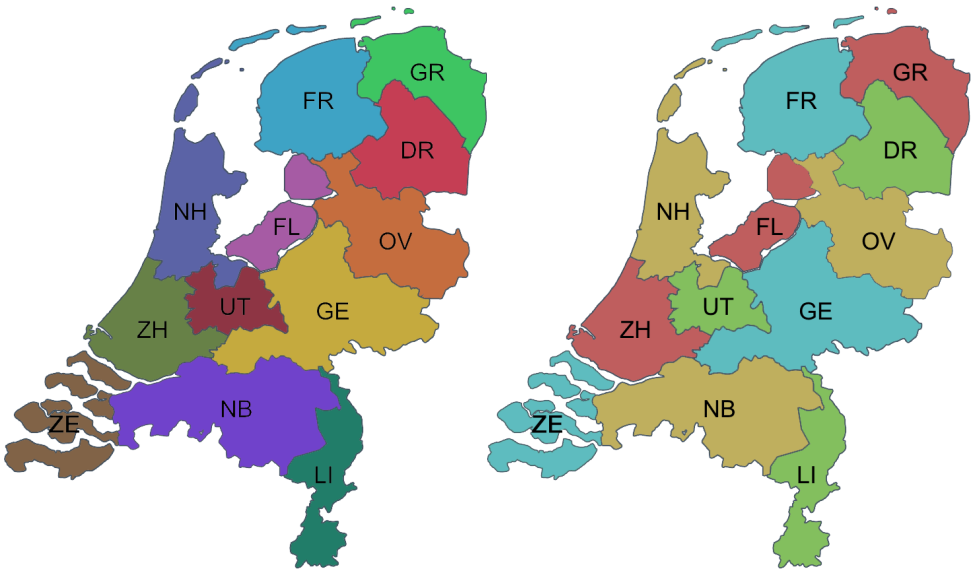


FIGURE 1.1: An example solution of the eight-queens puzzle.

1.2.2 Map-colouring problem

The *map-colouring problem* (also known as the *four-colour map problem*) is the task of trying to assign one of four given colours to each area in a map such that no neighbouring areas have the same colour. As an example, in Figure 1.2a we see the provinces of the Netherlands. Using the four colours `red`, `green`, `blue`, and `yellow`, each province can be assigned a colour such that no neighbouring provinces have the same colour, as shown in Figure 1.2b.



(a) (b)

FIGURE 1.2: (a) A map of the Netherlands, showing the provinces. (b) A solution to the map-colouring problem applied to the map of (A).

Formally, we can define this as the following CSP:

- $X = \{GR, FR, DR, OV, GE, FL, NH, ZH, UT, ZE, NB, LI\}$
- For each X_i we have $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}, \texttt{yellow}\}$
- $C = C_{GR} \cup C_{FR} \cup C_{DR} \cup C_{OV} \cup C_{GE} \cup C_{FL} \cup C_{NH} \cup C_{ZH} \cup C_{UT} \cup C_{ZE} \cup C_{NB} \cup C_{LI}$, where

$$
\begin{aligned}
C_{GR} &= \{GR \neq FR, GR \neq DR\} \\
C_{FR} &= \{FR \neq GR, FR \neq DR, FR \neq OV, FR \neq FL, FR \neq NH\} \\
C_{DR} &= \{DR \neq GR, DR \neq FR, DR \neq OV\} \\
C_{OV} &= \{OV \neq DR, OV \neq FL, OV \neq GE\} \\
C_{FL} &= \{FL \neq FR, FL \neq OV, FL \neq GE, FL \neq UT, FL \neq NH\} \\
C_{NH} &= \{NH \neq FR, NH \neq FL, NH \neq UT, NH \neq ZH\} \\
C_{GE} &= \{GE \neq OV, GE \neq FL, GE \neq UT, GE \neq NB, GE \neq LI\} \\
C_{UT} &= \{UT \neq FL, UT \neq GE, UT \neq NB, UT \neq ZH, UT \neq NH\} \\
C_{ZH} &= \{ZH \neq NH, ZH \neq UT, ZH \neq GE, ZH \neq NB, ZH \neq ZE\} \\
C_{ZE} &= \{ZE \neq ZH, ZE \neq NB\} \\
C_{NB} &= \{NB \neq ZE, NB \neq ZH, NB \neq GE, NB \neq LI\} \\
C_{LI} &= \{LI \neq NB, LI \neq GE\}
\end{aligned}
$$

1.2.3 Sudoku

Another well-known CSP is the 9x9 Sudoku puzzle, such as the one shown in Figure 1.3. The objective is to fill all the empty cells in the grid with digits such that

there are no duplicates in each row, column, or 3×3 block. More formally, the rules of the puzzle can be specified as constraints as follows (where $a_{i,j}$ denotes the value in the ith row of the jth column of the Sudoku puzzle):

- for $1 \leq i \leq 9$: *AllDiff*$(a_{i,1},\ a_{i,2},\ a_{i,3},\ a_{i,4},\ a_{i,5},\ a_{i,6},\ a_{i,7},\ a_{i,8},\ a_{i,9})$
- for $1 \leq j \leq 9$: *AllDiff*$(a_{1,j},\ a_{2,j},\ a_{3,j},\ a_{4,j},\ a_{5,j},\ a_{6,j},\ a_{7,j},\ a_{8,j},\ a_{9,j})$
- for $0 \leq i \leq 2,\ 0 \leq j \leq 2$:

$$\begin{aligned} \textit{AllDiff}(\ & a_{3i+1,3j+1},\ a_{3i+1,3j+2},\ a_{3i+1,3j+3}, \\ & a_{3i+1,3j+1}, a_{3i+1,3j+2}, a_{3i+1,3j+3}, \\ & a_{3i+1,3j+1}, a_{3i+1,3j+2}, a_{3i+1,3j+3}\ ) \end{aligned}$$

Now, for a specific sudoku puzzle, all that is needed are constraints that specify the given grid cells. For example, the sudoku in 1.3(a) is defined by the following extra constraints:

$$\begin{aligned} 1 &= a_{1,4} = a_{2,9} = a_{3,1} = a_{4,5} = a_{8,6} = a_{9,3} \\ 2 &= a_{2,6} = a_{3,8} \\ 3 &= a_{1,2} = a_{2,5} = a_{3,9} = a_{5,3} = a_{6,4} \\ 4 &= a_{1,3} = a_{3,4} = a_{8,2} \\ 5 &= a_{1,6} = a_{3,3} = a_{4,4} = a_{5,9} = a_{6,2} = a_{9,5} \\ 6 &= a_{1,7} = a_{2,4} = a_{3,2} = a_{4,6} = a_{5,8} = a_{7,9} \\ 7 &= a_{6,6} \\ 8 &= a_{4,2} = a_{5,4} = a_{9,6} \\ 9 &= a_{1,9} = a_{2,2} = a_{3,6} = a_{5,5} = a_{6,1} \end{aligned}$$

	3	4	1		5	6		9
	9		6	3	2			1
1	6	5	4		9		2	3
	8		5	1	6			
		3	8	9			6	5
9	5		3		7			
								6
	4				1			
		1		5	8			

(a)

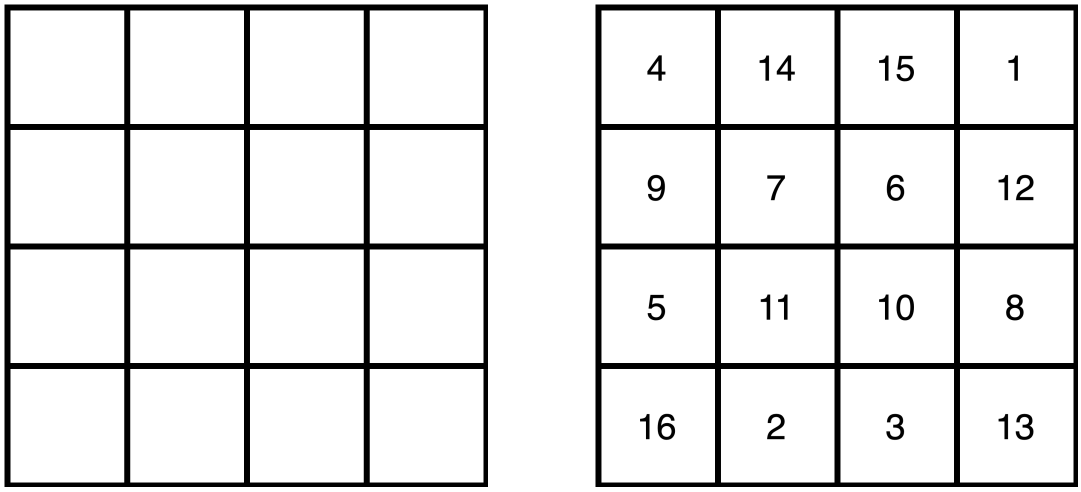
2	3	4	1	7	5	6	8	9
8	9	7	6	3	2	4	5	1
1	6	5	4	8	9	7	2	3
4	8	2	5	1	6	9	3	7
7	1	3	8	9	4	2	6	5
9	5	6	3	2	7	1	4	8
5	2	9	7	4	3	8	1	6
3	4	8	9	6	1	5	7	2
6	7	1	2	5	8	3	9	4

(b)

FIGURE 1.3: (a) A 9x9 Sudoku puzzle. (b) The solution to (A).

1.2.4 Magic square

A *magic square* is an $n \times n$ grid where the sum of each row, column, and both diagonals must be the same. Unlike the Sudoku puzzle, the grid is initially empty and the player is tasked to fill in the grid such that these requirements are met. In the context of this project, we restrict the domains of the grid cells to a predefined finite domain. For example, for a 4×4 magic square it suffices to restrict the domains to the range $[0, \ldots, m)$, where $m \geq 26$ (as can be seen from Figure 1.4).



(a) (b)

FIGURE 1.4: (a) An empty 4x4 grid. (b) A 4x4 magic square.

Solving a magic square puzzle as shown in 1.4a can be done by solving CSP with the following constraints (where $a_{i,j}$ denotes the value in the ith row of the jth column of the magic square puzzle):

- for $1 \leq i \leq 4$: $\mathit{sum}(a_{i,1},\ a_{i,2},\ a_{i,3},\ a_{i,4}) = S$
- for $1 \leq j \leq 4$: $\mathit{sum}(a_{1,j},\ a_{2,j},\ a_{3,j},\ a_{4,j}) = S$
- $\mathit{sum}(a_{1,1},\ a_{2,2},\ a_{3,3},\ a_{4,4}) = S$
- $\mathit{sum}(a_{1,4},\ a_{2,3},\ a_{3,2},\ a_{4,1}) = S$

Here, $\mathit{sum}(x_0, \ldots, x_n)$ denotes $\sum_{i=0}^{n} x_i$. A solution of the above CSP yields the magic square shown in Figure 1.4b.

1.2.5 Crypt-arithmetic puzzle

A *crypt-arithmetic puzzle* is a mathematical puzzle where individual digits of numbers in a mathematical expression are replaced by letters or symbols. Each letter in these puzzles represents a unique digit and there are no leading zeros in the numbers. The task is to assign to each letter a unique digit such that the mathematical expression is valid. While there are many different kinds of crypt-arithmetic puzzles available, the most common ones are only concerned with addition. An example of such a crypt-arithmetic puzzle can be seen in Fig. 1.5.

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

FIGURE 1.5: A simple crypt-arithmetic puzzle.

This puzzle can be formulated as a CSP as follows:

- $X = \{F, T, O, W, U, R, c_{10}, c_{100}, c_{1000}\}$.
- The domain of c_{10}, c_{100}, and c_{1000} is $\{0, 1\}$. For all other variables the domain is $D = \{0, \ldots, 9\}$.
- $C = \textit{AllDiff}(F, T, O, W, U, R) \cup \{O + O = R + 10 * c_{10},\ c_{10} + W + W = U + 10 * C_{100},\ c_{100} + T + T = O + 10 * c_{1000},\ c_{1000} = F,\ F \neq 0, T \neq 0\}$.

Note that the augmented variables c_{10}, c_{100}, and c_{1000} are introduced. They play the role of the carry (either 0 or 1) in the tens, hundreds, and thousands column respectively. Several solutions for this puzzle exist. An example solution is $F = 1,\ T = 8,\ O = 6,\ W = 4,\ U = 9,\ R = 2$, which corresponds to $846 + 846 = 1692$. Moreover, $c10 = 1,\ c100 = 0,\ c1000 = 1$, but these are helper variables, so their values in the solutions are not relevant.

1.2.6 Boolean satisfiability problem

A *Boolean satisfiability problem*, or *SAT*, is a problem in which you are given a Boolean formula and are tasked with determining whether there exists an interpretation that satisfies the given formula by incrementally replacing the variables in the formula with a Boolean value, `true` or `false`. A simple example of a Boolean formula is $(x \wedge y) \vee (\neg x \wedge z)$. The interpretation $x = \texttt{true},\ y = \texttt{true},\ z = \texttt{true}$ satisfies the aforementioned Boolean formula.

This SAT can be formulated as a CSP as follows:

- $X = \{x, y, z\}$
- For each X_i we have $D_i = \{\texttt{true}, \texttt{false}\}$
- $C = (x \wedge y) \vee (\neg x \wedge z)$

1.3 Solving CSPs using a backtracking search

Backtracking search is a depth-first search that incrementally assigns values to the given variables and backtracks when there are no more legal values left (in other words, when the state becomes inconsistent). Algorithm 1 shows a pseudo-code that implements this technique [20, p. 215].

Algorithm 1 Pseudocode for the backtracking search algorithm

```
1:  function BACKTRACKING-SEARCH(csp) returns a solution, or failure
2:      return Backtrack({ }, csp)
3:  end function
4:
5:  function BACKTRACK(state, csp) returns a solution, or failure
6:      if state is complete then return state
7:      end if
8:      var ← SELECT-UNASSIGNED-VARIABLE(csp)
9:      for all value in ORDER-DOMAIN-VALUES(var, state, csp) do
10:         if value is consistent with state then
11:             add {var = value} to state
12:             inferences ← INFERENCE(csp, var, value)
13:             if inferences ≠ failure then
14:                 add inferences to state
15:                 result ← BACKTRACK(state, csp)
16:                 if result ≠ failure then return result
17:                 end if
18:             end if
19:         end if
20:         remove {var = value} and inferences from state
21:     end for
22:     return failure
23: end function
```

The introduction of states allows us to look upon solving a given CSP as a backtracking state search problem. A solution to a CSP is a leaf node of the corresponding search tree in which a consistent value has been assigned to each variable of the CSP. An example of a partial search tree that is traversed by the backtracking search algorithm for the map-colouring problem is shown in Figure 1.6.

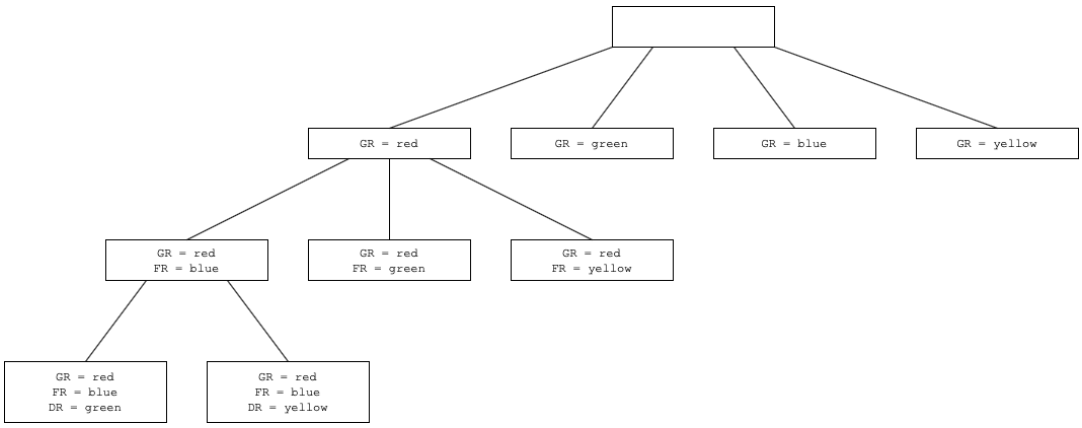


FIGURE 1.6: A (partial) search tree for the map-colouring problem described in Chapter 1.2.2

1.3.1 Heuristics

In each step of the backtracking search algorithm, we must select an unassigned variable in the CSP and assign a legal value to it such that no constraints are violated. The straightforward approach of selecting the next variable is to simply select the next unassigned variable in the list, although this can lead to nodes with a very large branching factor. In order to optimise the search, we can make use of *heuristics* to intelligently select the next variable to be assigned. Many different types of heuristics have been published, but we will resort to the most common ones.

Minimum-remaining-values (MRV) heuristic

In Figure 1.2b, we see that a partial state that the solving algorithm may reach is:

$$\begin{aligned} NH &= \texttt{yellow} \\ UT &= \texttt{green} \\ GE &= \texttt{blue} \end{aligned}$$

Since ZH is a neighbour to NH, UT and GE, we know that ZH must be `red` in order for $C_{ZH} = \textit{AllDiff}(ZH, ZE, NH, NH, UT, GE, NB)$ to hold.

The process of choosing the variable with the fewest legal values is called the *minimum-remaining-values* heuristic. It selects the variable that has the highest probability of leading to an inconsistent state soon. If a partial state is inconsistent, then the entire sub-tree of which this state is the root node can be pruned. Hence, this technique is likely to yield a smaller search tree than random variable selection. In the literature, this heuristic is also called the *most constrained variable heuristic* or the *fail-first* heuristic.

Degree heuristic

The MRV heuristic does not help at all in the case that all variables have an equal number of available values. In that case, the *degree* heuristic can be used. This heuristic attempts to reduce the branching factor of a search tree by looking at which variable is involved in the most constraints and assigning a value to this variable. If we take Figure 1.2b again as an example, we see that GE has the most neighbours (and therefore constraints) in the map of the Netherlands. If we decide to assign `blue` to GE, the branching factor will be reduced for all of GE's neighbouring nodes, as $LI, NB, ZH, UT, FL, OV \neq \texttt{blue}$.

Least-constraining-value heuristic

Unlike the aforementioned heuristics, the *least-constraining-value* (LCV) heuristic is used to improve the performance of the backtracking search by examining in which order the *values* should be assigned to the selected, unassigned variable. Unlike MRV, LCV chooses values for a variable x_i that occur in the *lowest* number of domains of all variables that occur in constraints that x_i also occurs in. In other words, LCV selects values that eliminate the fewest choices for the neighbouring variables in the constraint graph, which allows for flexibility for subsequent assignments [20, p. 217].

1.3.2 Inference

In Algorithm 1 we saw the following line:

1: $inferences \leftarrow$ INFERENCE$(csp, var, value)$

This type of *inference* is called *constraint propagation*. Constraint propagation uses the constraints to reduce the domains of the variables in the CSP. Once the domain of a variable x_i is reduced, this could, in turn, also reduce the domain for another variable x_j, et cetera. We will look at two kinds of constraint propagation; *node consistency*, and *arc consistency*.

Node consistency

A CSP is said to be *node consistent* when for each *unary* constraint C_x on a variable x, all values in its domain respect C_x.

For example, if x has a domain of $\{1, 2, 3, 4, 5\}$ and $C_x = \{x \mid x < 3\}$, we can remove the values $\{3, 4, 5\}$ from the domain of x and reduce it to $\{1, 2\}$. Making a CSP node consistent may result in great reduction of the execution time of the backtracking algorithm.

Arc consistency

A *binary* constraint C_{xy} on variables x, y is said to be *arc consistent* if for each value for x (taken from the domain of x) there exists a value for y (in the domain of y) such that C_{xy} holds. A CSP is said to be *arc consistent* if all its binary constraints are arc consistent.

As an example, let us take a look at the following CSP:

- $X = \{a, b, c\}$
- For each X_i, we have $D_i = \{1, 2, 3\}$
- $C = \{a > b,\ b = c\}$

This CSP is clearly not arc consistent, as there is no value for b if $a = 1$. Therefore, we remove the value 1 from the domain of a:

- $X = \{a, b, c\}$
- $D_a = \{2, 3\}$, $D_b = \{1, 2, 3\}$, $D_c = \{1, 2, 3\}$
- $C = \{a > b,\ b = c\}$

This CSP is still not arc consistent, as there is no value for a if $b = 3$ (because $a > 3$ fails). Therefore, we remove the value 3 from the domain of b:

- $X = \{a, b, c\}$
- $D_a = \{2, 3\}$, $D_b = \{1, 2\}$, $D_c = \{1, 2, 3\}$

- $C = \{a > b,\ b = c\}$

This still leaves us with a CSP that is not arc consistent. If $c = 3$, there is no matching value for b. Hence, we remove 3 from the domain of c, which (finally) leaves us with an arc consistent CSP:

- $X = \{a, b, c\}$
- $D_a = \{2, 3\}$, $D_b = \{1, 2\}$, $D_c = \{1, 2\}$
- $C = \{a > b,\ b = c\}$

In this example, we have reduced the domain of each variable by one-third! Making a CSP arc consistent may speed up the backtracking solving process significantly, since the number of states visited by this brute force process may be as large as the product of the sizes of the domains.

1.4 Literature review

1.4.1 History

Prolog [14] was the first logic programming language. A Prolog program essentially consists of a set of Boolean equations (constraints) that declaratively specify a problem. Hence, Prolog can be seen as the first Constraint Logic Programming (CLP) programming language [3], and a Prolog implementation can be seen as a solver for Boolean CSPs.

Soon after Prolog's release in 1972, research started to appear on the theory behind efficiently solving CSPs. This started with the introduction of minimal graphs for the representation and handling of constraints by Montari in 1974 [18]. Soon after, a paper was published in 1977 [16] that extended Montari's work and introduced a number of problem reduction strategies and algorithms [22].

Researchers began experimenting with the idea of using backtrack-free search algorithms [7] and backtrack-bound searches [10]. The focus, however, shifted towards standard backtracking search algorithms again [15, 17], which are used to this day.

Following the research on search algorithms, heuristics started to get developed to optimise the search process for solving CSPs [11]. This ranged from using network-based heuristics [7] to using evolutionary algorithms [5, 9], which greatly improved the performance of solvers on a subset of CSPs.

Research on concurrent CLP started to appear [13] in the 1980s, followed by research on general Concurrent Constraint Programming [21]. This research on general Concurrent Constraint Programming inspired the addition of constrained reasoning into multi-paradigm languages, notably Mozart [6] in 1991. As an example, Mozart has been used to create a program verification system [8]. Mozart eventually dropped the support for constraints in the first release of Mozart 2 due to the complexity of the ongoing developments in efficient constraint solving; it is still possible to do constraint solving in Mozart, but this will have to go through the library Gecode [12].

Over the years, many CSP solvers have been made and released, but there are still very few (efficient) general-purpose CSP solvers [1, 19].

1.4.2 Multi-core CSP solvers

Multi-core processing has become the new standard in modern computing [4]. Many computing science (related) fields have benefited from using multi-core processing by adapting algorithms such that they can run concurrently using the shared memory threading model. The field concerned with solving CSPs, however, appears to lag behind in this respect. While there are currently many CSP solvers that are highly optimised for specific CSPs, there appears to be very little focus on developing efficient multi-core general-purpose CSP solvers.

1.4.3 Parallelization

We will implement a concurrent CSP solver based on a parallelization of the backtracking search algorithm which is commonly used for solving CSPs [20]. Additionally, we need to implement some concurrent data structures for performing bookkeeping.

We will use a *search tree* data structure to keep track of the history of all partial states that have been covered by the search algorithm to avoid duplicate searches by multiple threads. Since we only need to know whether a substate of the state that is currently being inspected has been covered in the past, it does not matter whether the (partial) states that are stored in the tree previously yielded a solution or not. Therefore, we can safely store both inconsistent as well as consistent partial states in the same search tree. Note that because of this property, we only need to store partial states in the tree and the solutions (which are complete states). We do not need to store complete states that are not a solution, as these states must have been covered earlier on by other threads in the program.

Chapter 2

Parallelization

2.1 Parallelizing the Search Algorithm

In Algorithm 1, we saw that a CSP is solved by first selecting an unassigned variable, followed by iterating over the values in that variable its domain. This was shown in the following lines:

1: $var \leftarrow$ SELECT-UNASSIGNED-VARIABLE(csp)
2: **for all** $value$ in ORDER-DOMAIN-VALUES($var, state, csp$) **do**

This algorithm, therefore, tries to solve for one variable at a time, while the other (unassigned) variables are simply waiting until it is their turn to be processed. This approach of consecutively solving the unassigned variables is therefore quite inefficient on a multi-core computer, as other variables could be processed in the mean time.

2.1.1 Processing Variables Independently

Since Algorithm 1 does not depend on the order of the unassigned variables that are picked (it does not matter which variable SELECT-UNASSIGNED-VARIABLE(csp) returns), we could start the searching process from any unassigned variable. This observation naturally allows for this algorithm to be parallelized.

Idea A better approach to solving a CSP using the backtracking search algorithm shown in Algorithm 1 is to assign a thread to each variable and start solving the CSP from that variable. This way, all variables will be processed concurrently, rather than consecutively, which will save time as the variables are no longer 'waiting' on other variables.

2.2 Master-slave Parallelization

There are different approaches that one could take to implement this parallelization idea, but the most natural approach that comes to mind is making use of the *master-slave parallelization* (or MSP) paradigm. In MSP, one thread (the so-called *master thread*) is assigned a problem to solve. It does this by splitting the problem into smaller *tasks*, which are distributed and assigned to (multiple) *slave threads*. These

slave threads are each responsible for their own tasks. Once they receive a task, they will perform the task and request a new one once they are done with their current task. This process continues until there are no more tasks left.

2.2.1 Tasks

One idea that was briefly mentioned in Chapter 2.1.1 is to assign one slave thread per variable. Although this solution might speed up the solving process, there are some complications with this approach;

- What happens if some variable X_i has a large domain, whereas the other variables have relatively small domains? Processing variable X_i will take much longer than the other variables, because trying all elements of its domain takes longer than trying all elements of the other smaller domains. This means that processing this variable will lead to a load-balancing bottleneck.
- Perhaps more importantly; what happens if there are more slave threads available than variables? If there are two unassigned variables and we have eight slave threads available, six slave threads will remain idle. This leads to an unnecessary waste of resources.

Idea Rather than assigning one slave thread per variable, we could split the variables and their domains into smaller *tasks*. In this case, a task consists of a variable and a subset of its original domain. If the original domain is sufficiently small, we may use the entire domain in a task as well.

The two aforementioned issues are now no longer present. It does not matter whether there are variables with large domains anymore, as they are split into smaller tasks anyway. Moreover, since a variable can now be split over multiple tasks, we can now assign multiple slave threads to different subsets of a domain of a variable. This means that all slave threads will be active in solving a CSP and will have an (approximately) equally sized task. Consequently, this approach leads to *load balancing* amongst the slave threads which can be seen in Figure 2.1.

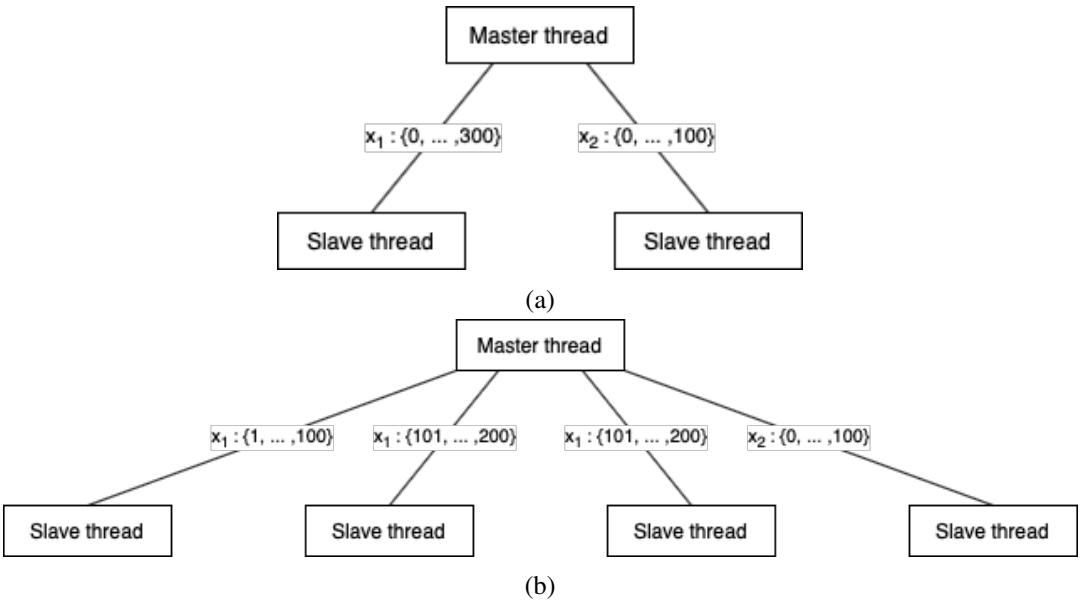


FIGURE 2.1: Master-slave parallelization without (a) and with (b) load balancing.

The implementation of the master-slave parallelization is discussed in Chapter 5.3.

Chapter 3

Search tree

Let us consider the following CSP:

- $X = \{w, x, y, z\}$
- For each $X_i,\ D_i = \{1, \ldots, 100\}$

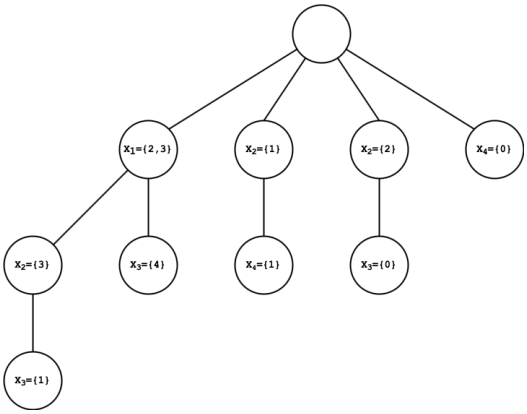
where C can be any set of constraints. Suppose that we would like to solve such a CSP with two threads T_1 and T_2. Thread T_1 starts the solving process with variable x and T_2 starts with variable y. Now, at some point, T_1 may process the partial state $\{x = 1, y = 2\}$ whereas T_2 has processed similar partial state in the past, namely $\{y = 2, x = 1\}$. From this point onward, the search for assignments for the remaining variables (w and z) in T_1 will yield an identical search tree as the one that was already generated by T_2 because the partial states are identical (albeit they were reached with a different order of variables)! This means that T_1 will be searching for solutions in a subspace that has already been searched before, which is clearly very inefficient. It would be much better to store these "covered" (partial) states in a shared data structure to avoid duplicate searches by different threads. For this shared data structure we invented a kind of search tree.

3.1 Tree for Partial States

This search tree data structure and its operations are best explained by looking at an example, such as the one shown in Figure 3.1a. The nodes in this tree consist of a variable together with a *set* of values. A branch in this search tree represents a (partial) state. The branch starting at node $x_2 = \{1\}$ represents the partial state $\{x_2 = 1,\ x_4 = 1\}$, for example. In the case that a node has multiple values in its set, such as $x_1 = \{2,\ 3\}$, the branches (and therefore (partial) states) are defined as all possible branches that can be generated with the values in this set. In this case, the branches that are stored starting at node $x_1 = \{2,\ 3\}$ are:

- $\{x_1 = 2,\ x_2 = 3,\ x_3 = 1\}$
- $\{x_1 = 3,\ x_2 = 3,\ x_3 = 1\}$
- $\{x_1 = 2,\ x_3 = 4\}$
- $\{x_1 = 3,\ x_3 = 4\}$

The full table of all partial states that are stored in the search tree of Figure 3.1a can be found in the Figure 3.1b.



(a)

$x_1 = 2$	$x_2 = 3$	$x_3 = 1$	–
$x_1 = 3$	$x_2 = 3$	$x_3 = 1$	–
$x_1 = 2$	–	$x_3 = 4$	–
$x_1 = 3$	–	$x_3 = 4$	–
–	$x_2 = 1$	–	$x_4 = 1$
–	$x_2 = 2$	$x_3 = 0$	–
–	–	–	$x_4 = 0$

(b)

FIGURE 3.1: (a) A search tree. (b) A table depicting all the branches stored in (a).

Now that we have an overview of the structure of this search tree, let us take a look at how (and which) operations work on these trees.

3.2 Operations

Since we will be using these search trees to look up (partial) states of previous attempts, we only need a *search* and an *insert* operation. We do not need to delete partial states from these trees, as we are interested in *all* previous attempts.

A crucial precondition for these operations is that the (partial) states must be sorted alphabetically (i.e. lexicographically) on the variable names. We will explain later why this is important. For now, let us first take a look at these operations and look at some examples.

3.2.1 Search

Idea To find (parts of) an input state in this search tree, we traverse the children of the root node until we find a node that contains a variable that is also present in the input state. Moreover, the value of the variable in the input state must be in the set of values of the variable in the node. If such a match is found, we continue the search process by making this node the new root node and iterating over all its children until another match is found. An input state is (partially) present in the search tree if there exists a branch (up until a leaf) such that all nodes match with the corresponding variables in the state. If there exists no such branch, the input state is not present in the search tree.

Earlier, we mentioned that one of the preconditions for the operations on this search tree is that the input state must be sorted alphabetically on the variable names. The reason why this is important is to avoid duplicate branches in the tree. If we do not have this precondition, then we would have two (completely separate) branches

for the partial states $\{x_2 = 1,\ x_4 = 1\}$ and $\{x_4 = 1,\ x_2 = 1\}$, even though they represent the same partial states.

Let us now take a look at an example. Note that in these examples a green node represents a match between the node and a variable in the (partial) state, whereas a red node represents a mismatch.

Example 1 We would like to know whether (parts of) the state $\{x_1 = 1,\ x_2 = 2,\ x_3 = 0,\ x_4 = 1\}$ exists in the search tree shown in Figure 3.1a.

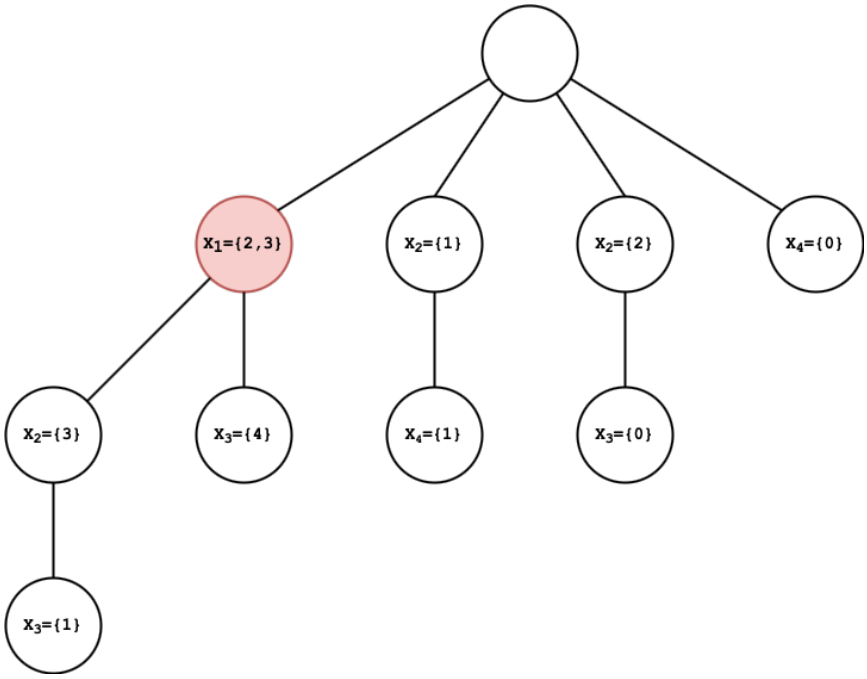


FIGURE 3.2: $x_1 = 1$ in our state, so no match for the node $x_1 = \{2, 3\}$.

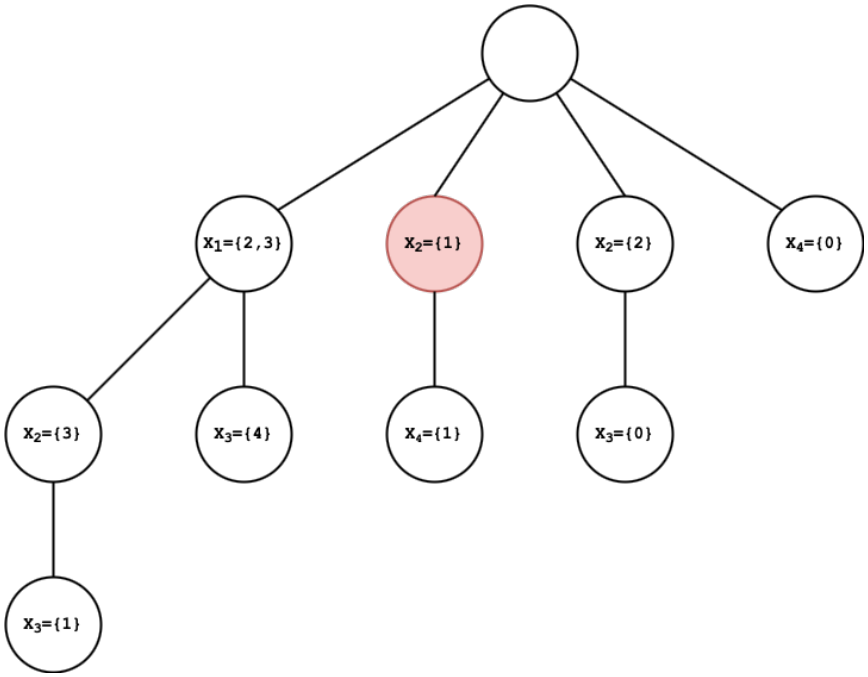


FIGURE 3.3: $x_2 = 2$ in our state, so no match for the node $x_2 = \{1\}$.

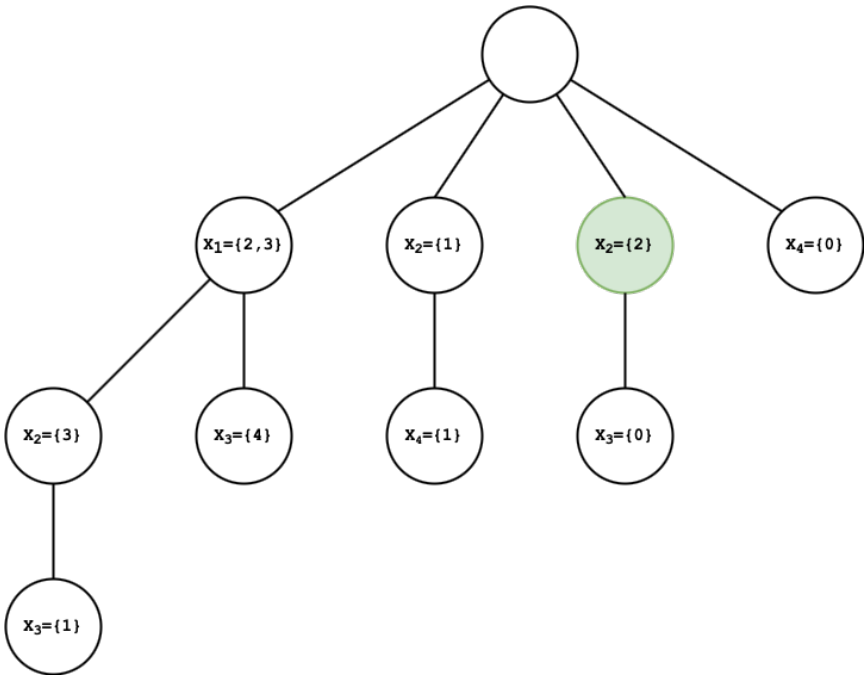


FIGURE 3.4: $x_2 = 2$ in our state, so a match for the node $x_2 = \{2\}$.

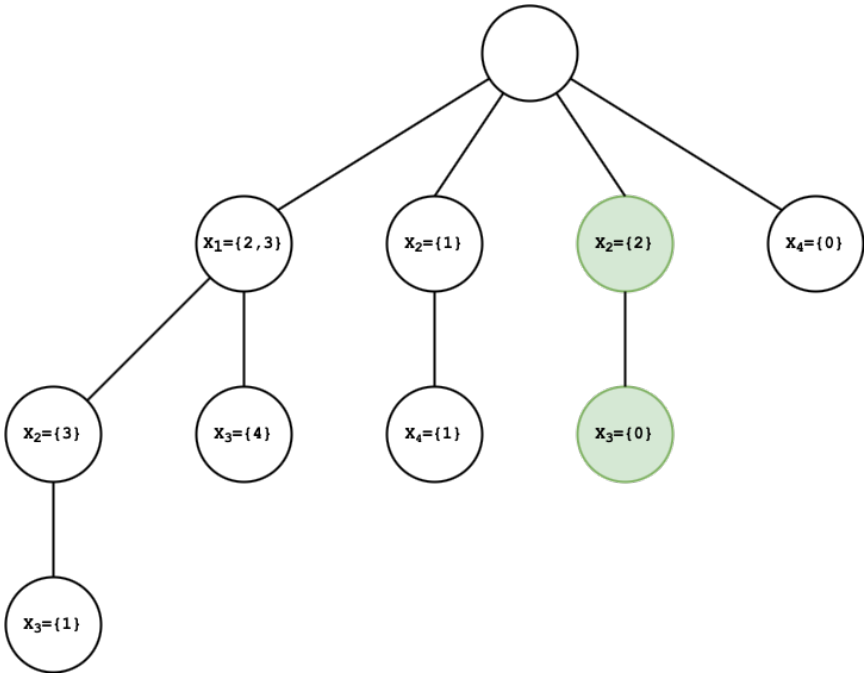
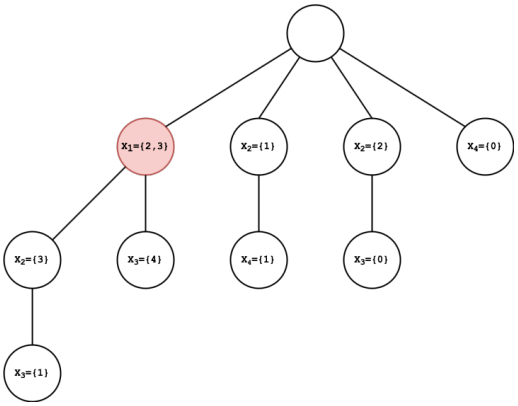


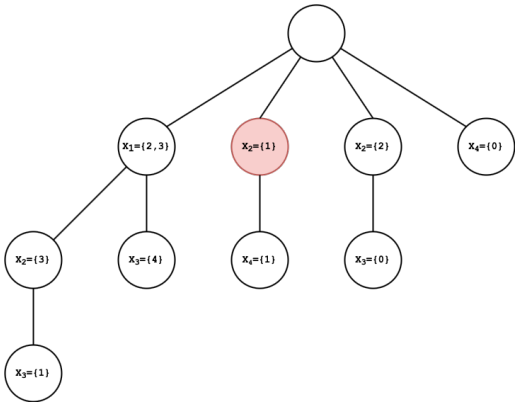
FIGURE 3.5: $x_3 = 0$ in our state, so a match for the node $x_3 = \{0\}$.

We conclude that the partial state $\{x_2 = 2,\ x_3 = 0\}$ is stored in the tree and is consistent with the state $\{x_1 = 1,\ x_2 = 2,\ x_3 = 0,\ x_4 = 1\}$ that we searched for.

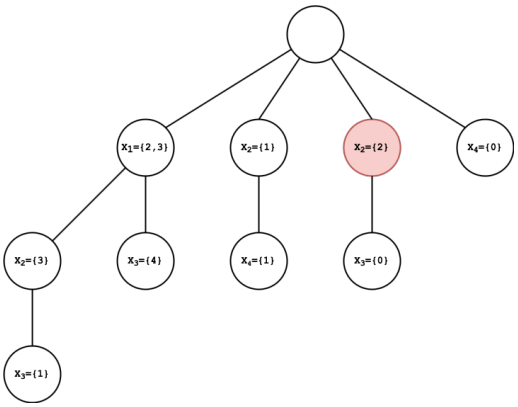
Example 2 We would like to know whether (parts of) the state $\{x_1 = 1,\ x_2 = 3,\ x_3 = 3,\ x_4 = 3\}$ exists in the search tree shown in Figure 3.1a. The search for this state is depicted in Figure 3.6. The conclusion for this search is clearly that there are no partial states in the tree that are consistent with the given state.



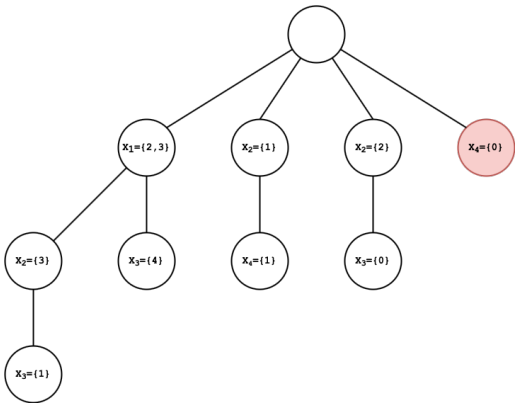
(a) $x_1 = 1$ in our state, so no match for the node $x_1 = \{2, 3\}$.



(b) $x_2 = 3$ in our state, so no match for the node $x_2 = \{1\}$.



(c) $x_2 = 3$ in our state, so no match for the node $x_2 = \{2\}$.



(d) $x_4 = 3$ in our state, so no match for the node $x_4 = \{0\}$.

FIGURE 3.6: Searching for $\{x_1 = 1,\ x_2 = 3,\ x_3 = 3,\ x_4 = 3\}$ in the search tree shown in Figure 3.1a.

3.2.2 Insert

Idea Inserting in this search tree is relatively simple; we follow existing branches as much as possible, and as soon as a node does not exist in a certain place in the search tree, we create a new sub-branch at that point. Let us now take a look at some examples. Note that in these examples a green node represents a newly added node, whereas an orange node represents an existing node that has changed (in other words, the set of values in that node has changed).

Example 1 We would like to insert the partial state $\{x_2 = 1,\ x_3 = 0\}$ in the search tree shown in Figure 3.1a. The insert operation is shown in Figure 3.7. The partial state $\{x_2 = 1\}$ was already present, and its branch is extended with a new child-node $\{x_3 = 0\}$.

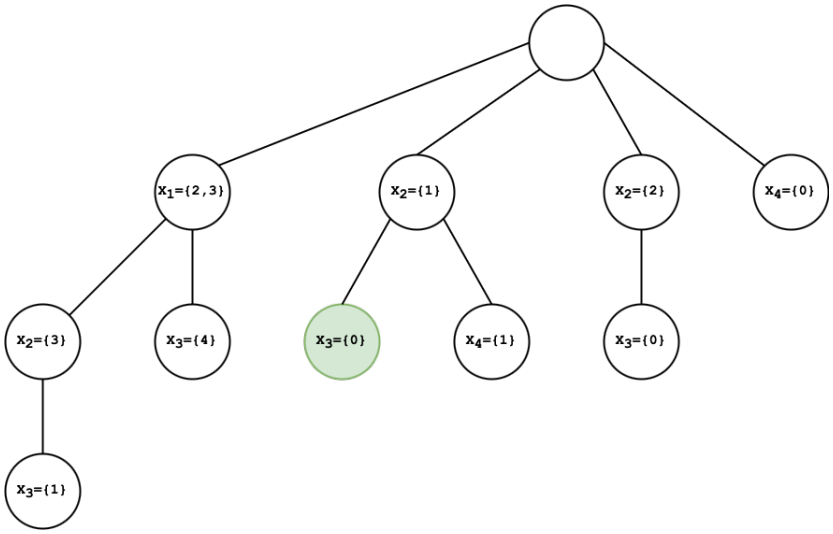


FIGURE 3.7: Inserting the partial state $\{x_2 = 1, x_3 = 0\}$ into the search tree shown in Figure 3.1a.

Example 2 Suppose we would now like to add the partial state $\{x_2 = 2, x_4 = 1\}$ to the tree that is shown in Figure 3.7. A naive way of doing this is displayed in Figure 3.8; the partial state $x_2 = 2$ was already present,and its branch is extended with a new child-node $x_4 = \{1\}$.

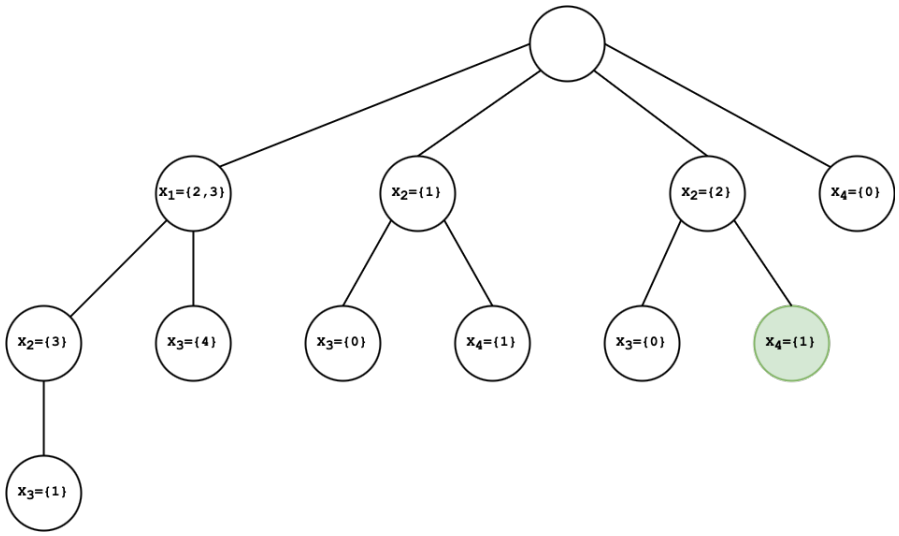


FIGURE 3.8: Inefficient insertion of $\{x_2 = 2,\ x_4 = 1\}$ into the tree that is shown in Figure 3.7.

This yields, however, a tree with two (nearly) identical sub-trees: ($\{x_2 = 1\}$ and $\{x_2 = 2\}$). The only difference between these branches is the values in their parent node x_2. This means that we are storing the same sub-trees twice, with a minor difference. It would be much more efficient to merge these overlapping sub-trees into one by simply merging the values of the x_2 nodes, as can be seen in Figure 3.9.

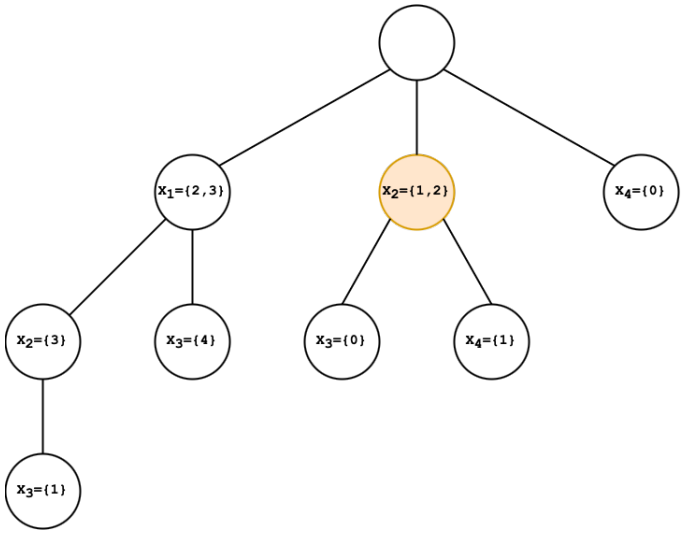


FIGURE 3.9: Efficient insertion of $\{x_2 = 2,\ x_4 = 1\}$ into the tree that is shown in Figure 3.7.

This compact notation saves a lot of memory. This is especially the case when the CSP consists of many variables with large domains since this would result in many extra branches if stored naively.

3.3 Access

We now have a data structure that can be used to avoid duplicate searches amongst all threads. In practice, we will only need one instance of this tree to store both the solutions and the partial states that have been covered before. However, the following question naturally arises; what should the access rights of the threads on these search trees be? Are there any complications when we allow all threads to insert and search in the search trees? Moreover, what happens if a thread wants to insert a new (partial) state while another thread is searching for another (partial) state?

3.3.1 Operations

Search Since each slave thread must be able to prematurely abort a search if (a part of) the current state has been covered before, it is crucial that the slave threads have at least search access to the tree. Otherwise, the benefits of storing previous attempts in the search trees to avoid duplicate searches would cease to exist.

Insert Since the master thread is used as a middle man to get new tasks, it can also be used as a middle man for inserting a new (partial) state in the search trees. This approach ensures that all states are inserted consecutively, which is crucial as we do not want to insert multiple states at the same time. Therefore, only the master thread has insert access to the tree and the slave threads will have to signal the master thread to request insertions in the tree.

The access rights of the threads on the search trees are shown in Figure 3.10.

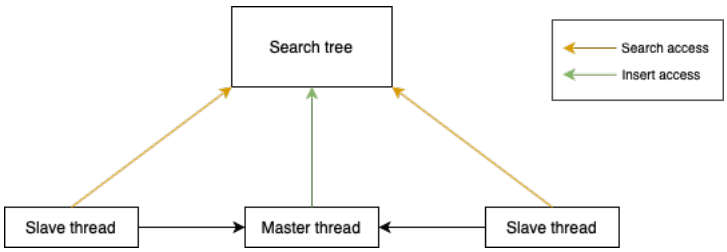


FIGURE 3.10: A diagram showing the access rights of the threads to the search tree.

3.3.2 Data Race

A common problem that may occur in a parallelized application is when multiple threads try to access the same *shared* memory simultaneously, while (at least) one of the threads is trying to modify the data stored in this memory location. This phenomenon is called a *data race* and can lead to unintended behaviour (when the operations are executed in the wrong order, for example). In the case of the search trees, a data race may occur when the master thread is inserting a (partial) state into the search tree, while a slave thread is trying to search for a (partial) state in the same branches. Luckily, data races can be solved by making use of objects called *mutexes*. A mutex serves as a lock on shared memory and allows only one thread access to this memory at a time. If a thread would like to access this shared memory, it will first request to lock the corresponding mutex. If the mutex is already locked, however, the thread will be blocked until the mutex is unlocked by another thread. Once this is the case, the thread now locks the mutex and can continue its work.

In our solver, we have one mutex for the entire search tree. Now, whenever a thread tries to read a node in a branch that is currently being processed by an insert operation (performed by the master), it will be blocked until the insert operation has finished.

Chapter 4

CSP specification language

In order to be able to specify CSPs in a compact manner, we designed a simple specification language. We will first take a look at an example CSP defined in this language, before covering the details of the language.

Recall that the 8-queens problem can be specified as a formal CSP as follows:

- $X = \{X_1, \ldots, X_n\}$, where each X_i is the column of the queen that is placed on row i of the chess board.
- For each X_i we have $D_i = \{1, \ldots, 8\}$ (i.e. column numbers).
- $C = \textit{AllDiff}(X_1, \ldots, X_n) \cup \{j - i \neq abs(X_i - X_j) \mid 1 \leq i < j \leq n\}$.

In our specification language, this would be represented as follows:

```
# Find all solutions for the N queens problem. In this case, N = 8
const N = 8;

type
  column = {0..N-1};

csp diagonalCheck(board[0..N-1] : column) begin
  constraint
  {
    forall (i in [0..N-1]:
      forall (j in [i+1..N-1]:
        j - i != abs(board[i] - board[j]);
      )
    )
  }
end

csp main() begin
  solutions all;

  var
    board[0..N-1] : column;

  constraint
  {
    alldiff(board);
    diagonalCheck(board);
  }
end
```

We will use this example to explain the details of the specification language.

4.1 Constants

The first line of code is `const N = 8;`. As one might expect, this defines a global constant. Global constants are declared at the very top of the file and their scope is the entire file, including sub-CSPs. Their names must start with a capital letter, followed by numbers or lowercase letters. Moreover, their values must be constant (either an integer, a Boolean value, or an enumeration value).

4.2 Types

This (optional) section can be used to define types, which can be used to reuse domains that are used for multiple variables. Types (and therefore domains) are either sets of integers or sets of enumerable values:

- **Integers** contains only integers, ranges or constants. This is present in the 8 queens problem CSP: `column = {0..N-1}`. The type `column` now represents the set $\{0, 1, \ldots, N-2, N-1\}$.
- **Enumeration** contains strings of options that a variable can be. An example of this can be found in the implementation of the map-colouring problem in this specification language (found in Appendix A.1.2). In there, we have `colour = {red, green, blue, yellow}`. This is more user-friendly than having a type `colour = {0, 1, 2, 3}`, where each number would represent a colour.

Types defined in this section can be used in all CSPs.

4.3 CSPs

The specification language supports multiple (sub-)CSPs in one file, as can be seen in the example. There must be at least one CSP, though, named `main` with no parameters. The solver will always start by solving this CSP. All other sub-CSPs are then only solved if there is a path between the `main` CSP and the given sub-CSP. As an example, let us take a look at the following input file:

```
csp d(x) begin
  constraint
  {
    x != 5;
  }
end

csp c(x) begin
  constraint
  {
    x != 4;
    d(x);
```

```
  }
end

csp b(x) begin
  constraint
  {
    x != 3;
  }
end

csp a(x) begin
  constraint
  {
    x != 2;
    b(x);
  }
end

csp main() begin
  var
    x : {1..10};

  constraint
  {
    a(x);
  }
end
```

The CSP `c` is never solved, as there is no reference to it. The CSP `d` does have a reference from `c`, but since `c` ixtself cannot be reached from `main`, `d` is consequently also never solved, as it is only referenced in `c`.

A CSP declaration starts with its name and its (optional) parameters. Only the name of the parameters is provided (including the size if a parameter is an array). An example of a sub CSP with parameters:

```
csp diagonalCheck(board[0..N-1]) begin
  constraint
  {
    forall (i in [0..N-1]:
      forall (j in [i+1..N-1]:
        j - i != abs(board[i] - board[j]);
      )
    )
  }
end
```

Aside from its heading, a (sub-)CSP also has a body. The body (may) contain three sections, each of which we will cover next.

4.3.1 Solutions

This (optional) section can be used to define the number of solutions that are requested. The options are `solutions all`, `solutions single` and `solutions` n, for some natural number n. If no option is declared, the default option will be used, which is `solutions all`. Moreover, if there are only m solutions, with $m < n$, then the system will only show those m solutions (in other words, n is an upper bound).

4.3.2 Variables

In this section, the user can declare the variables that are used in the constraints. A variable declaration has one of the following forms:

- `variable : type`
- `variable : boolean` (note that `{boolean} is a built-in type`)
- `variable : set` (for example, `variable : {0..N}`)

Moreover, a variable can also be an array. The syntax for arrays is similar to the formats above;

- `variable[0..N] : type`
- `variable[0..N] : boolean`
- `variable[0..N] : set`

In the case of the 8-queens problem, we find the declaration `board[0..N-1] : column`, which is equivalent to `board[0..7] : {0..7}`. Arrays of higher dimensions are also supported by separating the indices by commas. An example of this can be seen in the sudoku implementation (found in Appendix A.1.3):

```
sudoku[1..9, 1..9] : {1..9}
```

Note that the user is free to choose the starting and ending indices of arrays. This allows the user to start indexing at any non-zero value, should they want to do this. Additionally, the specification language supports declaring multiple variables of the same type in a single line as follows:

```
variable1, variable2 : type
```

Accessing Array Elements

Apart from directly accessing elements from an array by using indices, the specification language also allows the user to access section(s) of an array. For the variable `sudoku[1..9, 1..9]` and `board[1..8]`, we may do the following:

```
max(board) = n;

alldiff(sudoku[1, 1..9]);

sum(sudoku[1..3, 1..3]) = m;
```

which, respectively, correspond to:

```
max(board[1], board[2], board[3], board[4],
    board[5], board[6], board[7], board[8]) = n;

alldiff(sudoku[1,1], sudoku[1,2], sudoku[1,3],
    sudoku[1,4], sudoku[1,5], sudoku[1,6],
    sudoku[1,7], sudoku[1,8], sudoku[1,9]);

sum(sudoku[1,1], sudoku[1,2], sudoku[1,3],
    sudoku[2,1], sudoku[2,2], sudoku[2,3],
    sudoku[3,1], sudoku[3,2], sudoku[3,3]) = m;
```

4.3.3 Constraints

A constraint can consist of several options:

- **Binary comparison**: a binary comparison of the form `Expression operator Expression`, where `operator` $\in \{=, !=, <, <=, >, >=\}$. An `Expression` can either be a Boolean expression, or a numeric expression. An example of a binary comparison using numeric expressions is

  ```
  j - i != abs(board[i] - board[j]);
  ```

  There are a few built-in mathematical functions available; `abs`, `min`, `max` and `sum`. For these functions, you can also pass ranges of variables (as discussed in Chapter 4.3.2):

  ```
  sum(square[i, 0..SIZE-1]) = RESULT;
  ```

  which is equivalent to:

  ```
  sum(square[0,0], square[0,1],
      square[0,2], square[0,3]) = 34;
  ```

  for `i=0`, `SIZE=4` and `RESULT=34`.

- **CSP call** a call to a sub-CSP, including its parameters. An example that we have seen already is `alldiff(board)`.

- **Forall statement** a for-loop alike construct that allows the user to iteratively define (parts of) a constraint. The syntax is:

  ```
  forall (iterVar in [start..end]:
      constraint;
  )
  ```

  where `start` and `end` are numeric expressions. An example of such a forall statement can be found in the 8 queens problem:

```
forall (i in [0..N]:
    forall (j in [i+1..N]:
        j - i != abs(board[i] - board[j]);
    )
)
```

which represents the set of constraints $\{j-i \neq abs(X_i-X_j) \mid 1 \leq i < j \leq n\}$, where $X_i =$ `board[i-1]`.

- **Constraint block** a block of constraints, encapsulated by curly brackets. A constraint block can be looked upon as a conjunction; all constraints in the block must hold in order for the block to hold.
- **Disjunction** a disjunction of two constraint blocks. This can be used if only one of the two blocks needs to be satisfied. A simple example that illustrates the SAT as described in Chapter 1.2.6:

```
{
  x = true;
  y = true;
}
OR
{
  x = false;
  z = true;
}
```

4.3.4 Miscellaneous

Note that whitespace (i.e. tabs, spaces, and newlines) is ignored. Moreover, the language supports single-line comments that start with a #. The following code:

```
forall (i in [0..SIZE-1]:
  # Column has no duplicates
  alldiff(sudoku[i, 0..SIZE-1]);

  # Row has no duplicates
  alldiff(sudoku[0..SIZE-1, i]);
)
```

is, therefore, equivalent to:

```
forall(i in [0..SIZE-1]:alldiff(sudoku[i,0..SIZE-1]);
alldiff(sudoku[0..SIZE-1,i]);)
```

4.4 Further Examples

The full list of CSP implementations of the examples described in Chapter 1.2 can be found in Appendix A.1.

Chapter 5

Workflow

5.1 Reading and parsing

To read and parse CSP specification files, we make use of the parser generator *Bison* in combination with the lexical analyser generator *Flex*. Loosely speaking, Flex tokenizes the input text and provides these tokens to a parser that was generated by Bison. This parser determines whether the sequence of tokens is according to the specified grammar. If this is the case then the parser returns an array of `structs` that is constructed (in so-called *grammar action rules*) during the parsing process. If the input does not follow the specified grammar, however, the parser will throw an error and let the user know which token resulted in an error. In addition to the regular error reporting provided by Bison, we have extended this to also show in which line number the incorrect token is found. This, of course, takes into account white space and comment lines.

On success, the parser returns a `struct Program` which was built while parsing the input. This struct is a data type that contains all the data of the CSP specification file and we may proceed to the next step of the program, which is parsing the program arguments.

We have added support for the following run-time flags:

- `-t=n`: the flag `-t` allows the user to specify how many slave threads may be used by the solver. Note that in the case that `-t=1` is specified, then the program will run sequentially;
- `-st=n`: the flag `-st` allows the user to specify whether they want to use the search trees to store intermediate, partial states in, and $\texttt{n} \in \{0, 1\}$;
- `-a=n`: the flag `-a` allows the user to specify whether they want to use arc consistency inference, and $\texttt{n} \in \{0, 1\}$;
- `-n=n`: the flag `-n` allows the user to specify whether they want to use node consistency inference, and $\texttt{n} \in \{0, 1\}$;
- `-p=n`: the flag `-p` allows the user to specify whether they want to solver to print the solutions, and $\texttt{n} \in \{0, 1\}$.

Note that these program arguments may be given in any order, and if duplicate arguments exist, the last ones will be used. If some program arguments are not provided, however, their default values will be used;

- `-t=numAvailableThreads` for Linux, and `-t=8` for macOS;
- `-st=0`, `-a=0`, `-n=1`, and `-p=1` for all operating systems.

5.1.1 Setting program scope

Lastly, we set the scopes in the `Program` accordingly. A *scope* is a `struct` that contains a list of all constants, sub-CSPs, variables, and parameters that can be called or accessed at a certain location in the code. Each CSP has its own scope containing all parameters and variables that are locally declared. These parameters and variables should not be accessible to other CSPs, and should, therefore, only occur in the scope of the corresponding CSP. Aside from these scopes, the `Program` also contains a global scope that contains a list of all constants and sub-CSPs defined in a file.

The next step is generating a reduced (simplified) variant of the CSP and solving this variant.

5.2 Reduced Form

So far, we have seen what a CSP looks like in the specification language. The actual solver, however, does not directly solve CSPs defined in this language. Instead, it first translates the input into a reduced variant of the specification language. This language is a simpler version of the original language and is used as a *normal form* by the solver. This intermediate, reduced form is never seen by the user, and the simplifications in this reduced form make the solving process significantly easier.

The following changes have been made in the reduced specification language (with respect to the original language):

- There is only one (large) CSP, namely `main`. All CSP calls are (recursively) replaced by the constraints defined in the corresponding CSPs, and the variables in these sub-CSPs are also copied into `main`. In order to avoid duplicate variable names in `main`, we prefix all variables with the CSP identifier they were defined and an underscore symbol ('_'). Note that underscores are not allowed in the original specification language. For example, a local variable `x` in sub-CSP `sub` is renamed into `sub_x`. For the solutions of the CSP, we only print the variables defined in the original `main` CSP, since we are not interested in the values for local variables and intermediate variables.
- Constants have been removed and all occurrences are simply replaced by their values.
- Arrays have been removed. An array of size n is expanded into n separate variables. For example, an array variable `x[1,2]` from the `main` CSP is translated into `main_x_1_2`. Note that when a solution is printed, all variables will be displayed in their original form (i.e. `main_x_1_2` is printed as `x[1,2]`).

- Domain type sections have been removed and all domains (sets) are written explicitly in the variable declarations.
- Domains are expanded to an array of values, rather than an array of ranges and values. Moreover, if there were any duplicates before, they are removed.

In the reduced format, constraints are also expanded as much as possible:

- Array ranges are expanded into n binary comparisons, where n is the number of elements in the array range.
- All `forall` statements are replaced by n copies of the body, where $n =$ `end-start`.
- `alldiff` is expanded into n binary comparisons, where n is the number of elements in the parameter variable(s).

Example Given the implementation of the 8 queens problem in the regular specification language, the reduced form variant would be as follows:

```
csp main() begin
 solutions all;

 var
  main_board_0 : {0,1,2,3,4,5,6,7};
  main_board_1 : {0,1,2,3,4,5,6,7};
  main_board_2 : {0,1,2,3,4,5,6,7};
  main_board_3 : {0,1,2,3,4,5,6,7};
  main_board_4 : {0,1,2,3,4,5,6,7};
  main_board_5 : {0,1,2,3,4,5,6,7};
  main_board_6 : {0,1,2,3,4,5,6,7};
  main_board_7 : {0,1,2,3,4,5,6,7};

 constraint
 {
  main_board_0 != main_board_1;
  main_board_0 != main_board_2;
  main_board_0 != main_board_3;
  main_board_0 != main_board_4;
  main_board_0 != main_board_5;
  main_board_0 != main_board_6;
  main_board_0 != main_board_7;
  main_board_1 != main_board_2;
  main_board_1 != main_board_3;
  main_board_1 != main_board_4;
  main_board_1 != main_board_5;
  main_board_1 != main_board_6;
  main_board_1 != main_board_7;
  main_board_2 != main_board_3;
  main_board_2 != main_board_4;
  main_board_2 != main_board_5;
  main_board_2 != main_board_6;
  main_board_2 != main_board_7;
  main_board_3 != main_board_4;
  main_board_3 != main_board_5;
  main_board_3 != main_board_6;
  main_board_3 != main_board_7;
  main_board_4 != main_board_5;
  main_board_4 != main_board_6;
  main_board_4 != main_board_7;
  main_board_5 != main_board_6;
  main_board_5 != main_board_7;
  main_board_6 != main_board_7;
  ((1)-(0)) != abs(((main_board_0)-(main_board_1)));
  ((2)-(0)) != abs(((main_board_0)-(main_board_2)));
```

```
    ((3)-(0)) != abs(((main_board_0)-(main_board_3)));
    ((4)-(0)) != abs(((main_board_0)-(main_board_4)));
    ((5)-(0)) != abs(((main_board_0)-(main_board_5)));
    ((6)-(0)) != abs(((main_board_0)-(main_board_6)));
    ((7)-(0)) != abs(((main_board_0)-(main_board_7)));
    ((2)-(1)) != abs(((main_board_1)-(main_board_2)));
    ((3)-(1)) != abs(((main_board_1)-(main_board_3)));
    ((4)-(1)) != abs(((main_board_1)-(main_board_4)));
    ((5)-(1)) != abs(((main_board_1)-(main_board_5)));
    ((6)-(1)) != abs(((main_board_1)-(main_board_6)));
    ((7)-(1)) != abs(((main_board_1)-(main_board_7)));
    ((3)-(2)) != abs(((main_board_2)-(main_board_3)));
    ((4)-(2)) != abs(((main_board_2)-(main_board_4)));
    ((5)-(2)) != abs(((main_board_2)-(main_board_5)));
    ((6)-(2)) != abs(((main_board_2)-(main_board_6)));
    ((7)-(2)) != abs(((main_board_2)-(main_board_7)));
    ((4)-(3)) != abs(((main_board_3)-(main_board_4)));
    ((5)-(3)) != abs(((main_board_3)-(main_board_5)));
    ((6)-(3)) != abs(((main_board_3)-(main_board_6)));
    ((7)-(3)) != abs(((main_board_3)-(main_board_7)));
    ((5)-(4)) != abs(((main_board_4)-(main_board_5)));
    ((6)-(4)) != abs(((main_board_4)-(main_board_6)));
    ((7)-(4)) != abs(((main_board_4)-(main_board_7)));
    ((6)-(5)) != abs(((main_board_5)-(main_board_6)));
    ((7)-(5)) != abs(((main_board_5)-(main_board_7)));
    ((7)-(6)) != abs(((main_board_6)-(main_board_7)));
  }

end
```

5.2.1 Preparing CSP

Before we can start solving the CSP `main`, we first need to make some preparations, as the CSP is not ready to be solved straight after having been parsed. We do this by performing the following actions:

Setting scope We set the scope for the CSP in this step in a similar way as discussed in Chapter 5.1.1. Since the reduced specification language only consists of one CSP `main` with no parameters, and constants are removed, the scope simply consists of all the variables that are defined in `main`.

Infer variable types Next, we iterate over all variables and infer the *type* of the variable by looking at the values in its domain. Based on these values, we assign one of the following types to the variable: `INT_TYPE`, `BOOL_TYPE`, or `ENUM_TYPE`. Moreover, if a variable is of type `BOOL_TYPE`, we assign the domain $\{0, 1\}$ to it. Note that we could also assign the enum domain $\{$`true,false`$\}$ to it, but since a `BOOL_TYPE` is treated as an `INT_TYPE`, it is simply easier to assign the set $\{0, 1\}$ to it.

Set assignments The following step is to look at all the constraints and check whether there are any assignments present, that are not part of any disjunctions, of the form `variable = constant` or `constant = variable`, where a constant value is either a Boolean value, an integer, or an enumeration value (string). If this is the case, we assign this constant value to this variable and disable this constraint such that it will not be inspected anymore in the future, to avoid unnecessarily looking at these constraints and thereby increasing the performance.

Remove duplicates Since the types were removed and replaced by sets, it may be the case that there are overlapping numbers in the set. For example, the type `tType : {1..9, 7}` will be translated into `{1,2,3,4,5,6,7,8,9,7}`, in which `7` occurs twice. Therefore, we must get rid of all duplicates values in the domains of the variables to avoid (unnecessary) duplicate searches. After the removal of duplicates, the domains are simple arrays containing values (and no ranges), such that we can iterate over the values in domains.

Link constraints In this step, we visit all constraints. If there is a constraint in which a variable name occurs, we find the corresponding variable that is stored in the scope and assign it to the variable in the constraint. We can now directly use the variables stored in the constraints, rather than searching it in the scope every time we need to access it.

Simplify constraints The constraints that are given in the reduced form can (sometimes) be simplified. For example, in the following (reduced) constraint from the 8-queens problem

```
((1)-(0)) != abs(((board_0)-(board_1)))
```

the term `((1)-(0))` can clearly be simplified to `1`. Since all constraints are binary comparisons in their core, we iterate over all these binary comparisons and check whether the expressions on the left-hand side and right-hand side can be simplified. If this is the case, we replace the expressions with these simplifications.

5.3 Parallelization

For the parallelization of the solver, we use the POSIX `pthreads` library. We will spawn the slave threads from the master thread using `pthread_create` and start the solving process on each thread separately. As mentioned in Chapter 3.3.2, we use mutexes to ensure that multiple `pthreads` do not cause a data race. We only have two mutexes; one for the search tree, and one for the array of tasks that is generated by the master thread, which will be discussed in Chapter 5.3.2.

5.3.1 Load balancing

Let us take a look at a CSP with the following variables:

```
var
  a : {1..10}
  b : {1..10}
  c : {1..1000000}
```

If we would like to solve this in parallel, with each thread starting at a different variable, we see that the threads that start solving at variables `a` and `b` (T_a and T_b, respectively) will likely be done relatively quickly compared to the thread that starts solving at variable `c` (T_c). Once T_a and T_b are done, they will have to wait for T_c to finish, without being able to help spread the workload. In this case, T_c might take

a long time to terminate and the resources of T_a and T_b will go to waste, as they are not helping T_c. Instead, we can create tasks for the slave threads to start solving from, rather than entire variables. A *task* consists of a variable with a (sub) domain of the original variable. This approach allows multiple threads to be able to start the solving process for the same variable, but at a different part of its original domain. This means that the load will be equally balanced amongst all available threads.

5.3.2 Master Thread

The master thread (which is the thread that starts the program) will first create an array of all the *tasks* for the slave threads, beforehand. This array is created by first determining a sufficient task size `taskSize` such that load balancing is feasible.

```
int stateSpaceSize = getTotalDomainSize(mainCSP);
int taskSize = (stateSpaceSize / n) / 100;
```

We do this by dividing the total state space size (in other words, the product of the sizes of all domains) by n, the number of threads available. This is, in its turn, divided by a modifier to ensure that there will be enough tasks to distribute amongst all threads. We found 100 to be a good modifier in practice.

Once `taskSize` has been determined, the solver iterates over all variables in `main`. Each variable is split into multiple variables with domains of (at most) size `taskSize`. If a variable has a domain that is smaller than `taskSize`, it is simply added to the array of tasks without getting split.

As an example, let `a` be a variable with domain `{1..10}`. If `taskSize=3`, the array of tasks will consist of the following four elements:

```
a : {1..3}
a : {4..6}
a : {7..9}
a : {10}
```

Once the array of tasks has been made, we will spawn n slave `pthreads` and start the solving process. We only spawn these threads if $n > 1$. If $n = 1$, we run the sequential version of the program.

5.3.3 Slave Thread

Each slave thread first creates a copy of the main CSP to make sure that the changes that they make remain local and do not affect the global main CSP. Once this is done, the slave thread starts grabbing tasks that the master thread pre-computed. In order to avoid multiple threads taking the same tasks, we use a mutex `task_mutex` to limit access to the array of tasks. Once the slave thread is able to lock the mutex, it grabs the next task and assigns it to a local variable. We then look for the variable in the main CSP that matches the name of the task variable and make a copy of this matching variable. We need this copy in order to reset the CSP back to its original state after returning from the recursive solving procedure.

After we have made the copy, we look for the index of the variable in the CSP that matches the identifier (name) of the task variable. Once this match has been found, we replace the variable at this index with the task variable and start the solving process. Once the slave thread is done trying to find solutions for this task, we reset the variable in the CSP back to the original variable and move onto the next task in the task array. We continue this process until either the task array is empty (there are no more tasks left), or until a global, *atomic* boolean variable `isDone` is set to `true`. This variable is set to true when the solver has found the desired number of solutions. This is only the case when `solutions single` or `solutions n`, of course.

Note that `isDone` is of type `atomic_bool`. Therefore, we can safely assign a value to this variable and inspect the variable without explicitly using mutexes for this, as C provides this functionality for us already.

The code for the slave threads is as follows:

```
/// Slave thread which grabs tasks and starts solving the CSPs using those variables
     as starting points
void *consumer(void *arg) {
   Program program = (Program)arg;

   // Make a backup of the mainCSP
   pthread_mutex_lock(&program->mainCSP.mutex);
   CSP mainCSP = copyCSP(program->mainCSP);
   pthread_mutex_unlock(&program->mainCSP.mutex);

   // Continue until either all solutions have been found, or there are no more
       tasks left
   while (!isDone) {
      // If there are no more tasks left, we are done
      pthread_mutex_lock(&task_mutex);
      if (taskArrayPos == taskArraySize) {
         pthread_mutex_unlock(&task_mutex);
         break;
      }

      // Select the next task as the starting variable
      Variable startVar = taskArray[taskArrayPos];
      taskArrayPos++;
      pthread_mutex_unlock(&task_mutex);

      // Get the index of the identifier of the starting variable in the CSP
      int i;
      for (i = 0; i < mainCSP.body.variables.size; i++) {
         if (strcmp(mainCSP.body.variables.vars[i]->ident, startVar->ident) == 0) {
            break;
         }
      }

      // Make a copy of the variable at this index and reset the state of the CSP
      Variable copy = copyVariable(mainCSP.body.variables.vars[i]);
      mainCSP.body.variables.vars[i] = startVar;
      mainCSP.body.variables.vars[i]->hasVal = true;
      mainCSP.state = makeState();
      mainCSP.state = initialiseState(mainCSP);

      // Remove the starting variable from the list of unassigned variables
      mainCSP.state->unassigned = removeVarNode(mainCSP.body.variables.vars[i],
          mainCSP.state->unassigned);

      // Start solving!
      backtrackingSearch(program, mainCSP, startVar);

      // Reset the values back
      mainCSP.body.variables.vars[i] = copy;
```

```
      mainCSP.body.variables.vars[i]->hasVal = false;
      freeState(mainCSP.state);
      mainCSP.state = NULL;
   }

   // Clean up the copy that was created
   freeCSP(mainCSP);

   return NULL;
}
```

5.4 Solving

The actual solving part of the program is rather straightforward. We have a function called `solve` which implements the backtracking search algorithm as shown in Algorithm 1. We select an unassigned variable from the CSP and start iterating over all the values in its domain. If the user wants to use inferences such as arc consistency and node-consistency, the solver will first perform the specified inferences. In order to make the variables in a CSP arc consistent, we apply the AC-3 algorithm [16]. To make the variables node-consistent (which is a simpler task), we iterate over all unary constraints and update the domains of the variables accordingly.

Once this is done, we check whether the CSP does not contain any variables that have an empty domain. If this is the case, we do not even need to check whether the CSP is consistent, as there are no possible values for this variable.

Once we find a value for which the state remains consistent, we move onto the next variable in the CSP by recursively calling the `solve` function again. We continue doing this until we are either at the base case (in which we have found a solution), or until we have run out of values.

The code for this function can be found below:

```
/// Finds all solutions of a CSP using backtracking
void solve(Program program, CSP csp, Variable startVar, bool isFirst) {
   // If sufficient solutions have been found, we are done
   if (isDone) {
      return;
   }

   // If the CSP is complete (no more unassigned variables left), we have a solution
      .
   // If the state of assigned variables already exists in the search tree, however,
        we simply skip this solution as it has been found by another thread before.
   // Else, we print the solution and insert it into the state
   if (isComplete(csp)) {
      pthread_mutex_lock(&tree_mutex);
      if (!treeContainsState(csp.state->assigned, program->history)) {
         insertState(csp.state->assigned, program->history);
         pthread_mutex_unlock(&tree_mutex);

         // Atomic variable, so can safely increment
         program->solNum++;

         if (printSolutionsFlag) {
            pthread_mutex_lock(&program->print_mutex);
            printSolutionWrapper(csp.state->assigned, program->solNum);
            pthread_mutex_unlock(&program->print_mutex);
         }
```

```
            if (program->solNum == program->solTotal) {
               isDone = true;
            }
         } else {
            pthread_mutex_unlock(&tree_mutex);
         }
         return;
      }

      // If it is the first iteration, we assign the starting variable to startVar,
          which will be one of the tasks. Else, we simply select the first unassigned
          variable
      Variable var;
      if (isFirst && startVar != NULL) {
         var = startVar;
      } else {
         var = selectUnassignedVariable(csp);
      }

      csp.state->assigned = addVarItem(var, csp.state->assigned);

      // Iterate over all the elements in the set
      for (int i = 0; i < var->decl.set->size; i++) {
         // If sufficient solutions have been found, we are done
         if (isDone) {
            return;
         }

         SetElement se = var->decl.set->elements[i];
         assignVariable(var, se);

         // Make a backup of the variables and initialise the CSP's state
         VariableSection backupVariables = copyVariableSection(csp.body.variables);
         csp.state = initialiseState(csp);

         // Use the tree to store intermediate results in
         if (useTreeFlag) {
            pthread_mutex_lock(&tree_mutex);
            if (treeContainsState(csp.state->assigned, program->history)) {
               pthread_mutex_unlock(&tree_mutex);
               resetCSP(csp, var, backupVariables);
               continue;
            }
            pthread_mutex_unlock(&tree_mutex);
         }

         // Make CSP node-consistent
         if (makeNodeConsistentFlag) {
            makeNodeConsistent(csp.body.variables, csp.body.constraints.start);

            if (containsEmptyDomains(csp.state)) {
               resetCSP(csp, var, backupVariables);
               continue;
            }
         }

         // Make CSP arc-consistent. Uses the AC-3 algorithm
         if (makeArcConsistentFlag) {
            makeArcConsistent(csp.body.variables, csp.body.constraints.start);

            if (containsEmptyDomains(csp.state)) {
               resetCSP(csp, var, backupVariables);
               continue;
            }
         }

         // If the assignment leads to a consistent CSP, we go into recursion and try
             to solve other variables
         if (isConsistent(csp.state, csp.body.constraints.start)) {
            solve(program, csp, startVar, false);
         }
```

```
      // Use the tree to store intermediate, partial states in
      if (useTreeFlag) {
         if (!isDone && csp.state->unassigned != NULL) {
            pthread_mutex_lock(&tree_mutex);
            if (!treeContainsState(csp.state->assigned, program->history)) {
               insertState(csp.state->assigned, program->history);
            }
            pthread_mutex_unlock(&tree_mutex);
         }
      }

      // Reset the value and domain of the set
      resetCSP(csp, var, backupVariables);
   }
}
```

Chapter 6

Performance

We now take a look at the performance of the solver on the example CSPs. For every CSP (except the Boolean SAT problem, as this CSP was too small), we ran the solver multiple times with different settings on a personal computer with an Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz. We turned off printing the actual solutions to not include I/O time in the timings. The used settings are shown in the tables below and are as follows:

- n denotes the number of slave threads used by the solver
- T denotes whether the tree was used for the intermediate partial states (instead of just using it to store the solutions in).
- A denotes whether the inference *arc consistency* was used.
- N denotes whether the inference *node consistency* was used.

The run-time was measured by making use of the `timer.c/timer.h` files, which store a `start` and `stop`, expressed as elapsed time since Unix Epoch. The run-time is then calculated by subtracting these two times from each other.

6.1 8 queens problem

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	0.090	0.405	0.209	0.124	0.105
$T = 0,\ A = 0,\ N = 1$	0.090	0.435	0.224	0.124	0.109
$T = 0,\ A = 1,\ N = 0$	0.091	0.751	0.402	0.227	0.190
$T = 0,\ A = 1,\ N = 1$	0.091	0.774	0.406	0.226	0.186
$T = 1,\ A = 0,\ N = 0$	0.090	0.473	0.364	0.351	0.456
$T = 1,\ A = 0,\ N = 1$	0.092	0.478	0.356	0.335	0.438
$T = 1,\ A = 1,\ N = 0$	0.091	0.602	0.395	0.363	0.483
$T = 1,\ A = 1,\ N = 1$	0.091	0.597	0.388	0.350	0.459

TABLE 6.1: Run-time (in seconds) for solving the 8 queens problem from Appendix A.1.1.

6.2 Map-colouring problem

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	0.484	3.097	1.600	0.922	0.841
$T = 0,\ A = 0,\ N = 1$	0.483	0.984	0.501	0.291	0.280
$T = 0,\ A = 1,\ N = 0$	0.487	4.969	2.604	1.495	1.376
$T = 0,\ A = 1,\ N = 1$	0.488	1.374	0.708	0.413	0.381
$T = 1,\ A = 0,\ N = 0$	0.482	3.352	2.186	2.065	2.888
$T = 1,\ A = 0,\ N = 1$	0.480	1.110	0.737	0.671	0.828
$T = 1,\ A = 1,\ N = 0$	0.486	4.330	2.591	2.203	3.065
$T = 1,\ A = 1,\ N = 1$	0.487	1.335	0.770	0.608	0.723

TABLE 6.2: Run-time (in seconds) for solving the map-colouring problem from Appendix A.1.2.

6.3 Sudoku puzzle

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	2.418	71.967	38.656	23.962	21.190s
$T = 0,\ A = 0,\ N = 1$	2.446	76.149	39.842	27.539	25.579s
$T = 0,\ A = 1,\ N = 0$	2.416	335.261	176.947	115.590	103.580s
$T = 0,\ A = 1,\ N = 1$	2.456	244.926	131.865	82.668	77.387s
$T = 1,\ A = 0,\ N = 0$	2.463	10.866	5.876	5.255	19.371s
$T = 1,\ A = 0,\ N = 1$	2.479	1.943	1.787	1.855	2.971s
$T = 1,\ A = 1,\ N = 0$	2.421	25.930	13.484	12.441	51.955s
$T = 1,\ A = 1,\ N = 1$	2.403	5.264	4.901	5.093	7.393s

TABLE 6.3: Run-time (in seconds) for solving the Sudoku puzzle from Appendix A.1.3.

6.4 Magic square puzzle

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	73.199	73.804	41.479	42.539	36.733s
$T = 0,\ A = 0,\ N = 1$	73.169	86.087	47.780	47.372	41.376s
$T = 0,\ A = 1,\ N = 0$	72.984	152.656	86.694	88.591	72.666s
$T = 0,\ A = 1,\ N = 1$	73.426	168.106	98.429	103.492	80.920s
$T = 1,\ A = 0,\ N = 0$	74.378	145.711	155.162	311.193	301.956s
$T = 1,\ A = 0,\ N = 1$	73.161	154.292	151.786	311.955	302.320s
$T = 1,\ A = 1,\ N = 0$	73.015	205.445	156.898	320.637	320.181s
$T = 1,\ A = 1,\ N = 1$	75.301	221.518	165.780	322.486	322.039s

TABLE 6.4: Run-time (in seconds) for solving the magic square puzzle from Appendix A.1.4.

6.5 Crypt-arithmetic puzzle

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	0.647	8.337	4.340	2.405	2.278s
$T = 0,\ A = 0,\ N = 1$	0.647	8.444	4.257	2.329	2.257s
$T = 0,\ A = 1,\ N = 0$	0.646	0.789	0.409	0.217	0.216s
$T = 0,\ A = 1,\ N = 1$	0.651	0.622	0.322	0.174	0.167s
$T = 1,\ A = 0,\ N = 0$	0.646	10.185	7.868	7.847	10.635s
$T = 1,\ A = 0,\ N = 1$	0.646	10.567	8.667	15.167	11.293s
$T = 1,\ A = 1,\ N = 0$	0.647	0.757	0.399	0.231	0.220s
$T = 1,\ A = 1,\ N = 1$	0.654	0.611	0.318	0.181	0.174s

TABLE 6.5: Run-time (in seconds) for solving the crypt-arithmetic puzzle from Appendix A.1.5.

6.6 Pythagorean triples (extra)

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	2.595	3.627	1.867	1.016	0.884s
$T = 0,\ A = 0,\ N = 1$	2.584	3.640	1.854	1.006	0.895s
$T = 0,\ A = 1,\ N = 0$	2.593	1.487	0.743	0.420	0.371s
$T = 0,\ A = 1,\ N = 1$	2.600	1.491	0.749	0.405	0.375s
$T = 1,\ A = 0,\ N = 0$	2.590	5.812	6.465	6.608	7.060s
$T = 1,\ A = 0,\ N = 1$	2.623	5.823	6.491	6.614	7.231s
$T = 1,\ A = 1,\ N = 0$	2.600	1.917	1.719	1.879	2.215s
$T = 1,\ A = 1,\ N = 1$	2.607	1.944	1.661	1.880	2.198s

TABLE 6.6: Run-time (in seconds) for finding all Pythagorean triples a, b, c such that $a^2 + b^2 = c^2$ with $a, b, c \in \{1, \ldots, 100\}$, from Appendix A.1.7 (but the domains of $a, b, c : \{1, \ldots, 100\}$ in this case).

	n				
	1	2	4	8	16
$T = 0,\ A = 0,\ N = 0$	53.540	70.236	35.542	19.598	18.632s
$T = 0,\ A = 0,\ N = 1$	53.568	70.309	35.948	20.516	19.173s
$T = 0,\ A = 1,\ N = 0$	53.713	27.890	14.169	8.215	7.250s
$T = 0,\ A = 1,\ N = 1$	53.867	27.313	13.874	7.921	6.550s
$T = 1,\ A = 0,\ N = 0$	53.855	183.501	165.553	178.496	187.053s
$T = 1,\ A = 0,\ N = 1$	53.645	183.076	171.349	180.851	185.382s
$T = 1,\ A = 1,\ N = 0$	53.497	43.456	44.963	46.592	48.839s
$T = 1,\ A = 1,\ N = 1$	52.967	43.461	44.998	47.807	49.054s

TABLE 6.7: Run-time (in seconds) for finding all Pythagorean triples a, b, c such that $a^2 + b^2 = c^2$ with $a, b, c \in \{1, \ldots, 250\}$, from Appendix A.1.7.

6.7 Evaluation

Based on the tables shown above in Chapter 6, we can construct the following table:

	n	T	A	N	Total speed up
8 queens problem	16	0	0	0	0.86
Map-colouring problem	16	0	0	1	1.72
Sudoku puzzle	4	1	0	1	1.39
Magic square puzzle	16	0	0	0	1.99
Crypt-arithmetic puzzle	16	0	1	1	3.89
Pythagorean Triples {1,...,100}	16	0	1	0	6.98
Pythagorean Triples {1,...,250}	16	0	1	1	8.22

TABLE 6.8: Solver settings with the largest speed-ups for each CSP and the total speed up, compared to 1 thread.

In Table 6.8, we see that the run-time for nearly all CSPs already improved for 4 threads on the best settings! The only exception to this rule is the 8 queens problem, but this is expected as this CSP has a small number of variables, a small domain, and very “basic" constraints that are easily satisfied. Moreover, we observe that nearly all CSPs had the fastest run-time at 16 threads, which is the number of threads that our CPU has. We did not have access to servers with more threads to test whether the performance would have improved even more, unfortunately.

Additionally, it has come to our attention that the majority of these optimal settings make use of node consistency. This was also expected, given that most constraints in the reduced form are binary constraints. This means that as soon as a variable is assigned a value, this binary constraint turns into a unary constraint, and since most variables occur in multiple of these (originally) binary constraints, making the CSP node consistent yields in much smaller domains for many variables.

Similarly, we see that arc consistency only had a positive influence on the CSPs that had constraints that involved three (or more) variables. As soon as one variable is assigned a value, these constraints are turned into binary constraints, which are the constraints that can highly benefit from becoming arc consistent. The reason why this is not the case for constraints that are “originally" binary constraints is simply that the probability of one of the variables getting assigned a value (and therefore reducing the constraint to a unary constraint) is much higher.

Lastly, we see that using the search tree to store intermediate partial states lowered the performance for most CSPs. This can be explained by the fact that most of the example CSPs do not contain many variables with large domains. In these cases, searching in (and inserting into) the search tree is not worth the overhead caused by these operations, as the height of the tree is simply not that significant. For the Sudoku puzzle, however, we see that using the search trees did in fact improve the performance (which also happened to be the only option that leads to an improvement). This makes sense, as a Sudoku puzzle $9 \times 9 = 81$ variables, each with a domain of $\{1, \ldots, 9\}$.

In short, we can conclude that the solver benefited from the parallelization and, in some cases, also benefited from the usage of the search trees on larger CSPs (such as the Sudoku puzzle). The real performance gains can be seen in the larger, more complex CSPs, such as the crypt-arithmetic puzzle and the two Pythagorean triples CSPs (with domains $\{1, \ldots, 100\}$ and $\{1, \ldots, 250\}$, respectively).

Chapter 7

Conclusion

The goal of this bachelor thesis was to write an efficient, multi-core solver for Constraint Satisfaction Problems (CSPs). Solving CSPs is still a relevant task to this day, and having a solver that can efficiently run on commodity hardware can be useful in several fields (like computer science, artificial intelligence, economics, operational research, mathematics, etc.).

As we have seen in Chapter 6.7, our solver is able to solve relatively large CSPs quite quickly. We ran the solver on the example CSPs in Appendix A.1 with different options. We saw that for nearly all CSPs, there was a configuration of options and threads that led to a significant speed-up compared to only using one thread with no options. We noticed that the solver already started to show improvements for most CSPs when 4 threads were assigned, which is available in the vast majority of commodity hardware. The only exceptions to this are the relatively small CSPs, for which the overhead of initiating the parallelization resulted in slightly slower run times, as is expected. Moreover, we saw that assigning more threads to the solver yielded better performance for the majority of the CSPs (with an exception of the map-colouring problem, the sudoku puzzle, and the magic square puzzle).

We can therefore conclude that our solver successfully met the requirements that we initially had and is able to offer a speed-up for the vast majority of Constraint Satisfaction Problems on commodity hardware.

7.1 Future work

Even though the solver fulfils our requirements, there is still room for improvement. Concerning usability, the main limitation is the fact that the solver only does very basic semantic checking on the user input. It simply assumes that the input is of the correct form. If this is not the case, the solver might crash. Moreover, the program currently consists of two separate executable files; one for translating the user input into the reduced form, and the second one for solving this reduced variant.

Aside from this, there are also some changes that can be made to improve the efficiency of the solver. Currently, we have not implemented the heuristics described in Chapter 1.3.1. The solver simply picks the next available variable in its list of unassigned variables, regardless of their properties. Implementing these heuristics may lead to significant improvement in performance.

Moreover, we currently make a deep copy of all variables in a CSP in the recursive step, whereas it is only necessary to make a copy of the variables that have been affected by the inferences. This implementation can lead to worse performance if the inferences hardly affect the variables since they will be reset in each iteration of the algorithm.

Another optimization that can be made is to disable the unary and binary constraints after having made a CSP node consistent and arc consistent, respectively. These constraints can be enabled again after we move onto the next value of the domain of the current variable, and will likely improve the performance as some constraints are not evaluated at all.

Lastly, as discussed in Chapter 3.3.2, we only use one mutex to lock the entire history tree. This is may be inefficient, however, as we only need to lock down the branches that are being inserted, not the entire search tree. Other branches may remain intact during an insertion. Therefore, the solver may perform better if it had a mutex on every node of the tree, rather than having just one global mutex.

Appendix A

Specification language examples

A.1 Input CSPs

A.1.1 8 queens problem

```
# Find all solutions for the N queens problem. In this case, N = 8
const N = 8;

type
  column = {0..N-1};

csp diagonalCheck(board[0..N-1] : column) begin
  constraint
  {
    forall (i in [0..N-1]:
      forall (j in [i+1..N-1]:
        j - i != abs(board[i] - board[j]);
      )
    )
  }
end

csp main() begin
  solutions all;

  var
    board[0..N-1] : column;

  constraint
  {
    alldiff(board);
    diagonalCheck(board);
  }
end
```

A.1.2 Map colouring problem

```
# Find one solution to the map-colouring problem for the provinces of the
    Netherlands

type
  colours = {red, green, blue, yellow};

csp main() begin
  # no solutions-section, showcase that this is optional

  var
    gr, fr, dr, ov, ge, fl: colours;
    nh, zh, ut, ze, nb, li: colours;

  constraint
```

```
  {
    # Groningen's neighbours
    gr != fr; gr != dr;

    # Friesland's neighbours
    fr != gr; fr != dr; fr != ov; fr != fl; fr != nh;

    # Drenthe's neighbours
    dr != gr; dr != fr; dr != ov;

    # Overijssel's neighbours
    ov != dr; ov != fl; ov != ge;

    # Flevoland's neighbours
    fl != fr; fl != ov; fl != ge; fl != ut; fl != nh;

    # Noord-Holland's neighbours
    nh != fr; nh != fl; nh != ut; nh != zh;

    # Gelderland's neighbours
    ge != ov; ge != fl; ge != ut; ge != nb; ge != li;

    # Utrecht's neighbours
    ut != fl; ut != ge; ut != nb; ut != zh; ut != nh;

    # Zuid-Holland's neighbours
    zh != nh; zh != ut; zh != ge; zh != nb; zh != ze;

    # Zeeland's neighbours
    ze != zh; ze != nb;

    # Noord-Brabant's neighbours
    nb != ze; nb != zh; nb != ge; nb != li;

    # Limburg's neighbours
    li != nb; li != ge;
  }
end
```

A.1.3 Sudoku puzzle

```
# Find all solutions for a given sudoku

const SIZE = 9;
const BLOCKSIZE = 3;

type
  sType = {1..SIZE};

csp blockCheck (sudoku[0..SIZE-1, 0..SIZE-1] : sType) begin
  constraint
  {
    forall (j in [0..BLOCKSIZE-1]:
      forall (k in [0..BLOCKSIZE-1]:
        alldiff(
          sudoku[3*j, 3*k], sudoku[3*j, 3*k+1],
          sudoku[3*j, 3*k+2], sudoku[3*j+1, 3*k],
          sudoku[3*j+1, 3*k+1], sudoku[3*j+1, 3*k+2],
          sudoku[3*j+2, 3*k], sudoku[3*j+2, 3*k+1],
          sudoku[3*j+2, 3*k+2]
        );
      )
    )
  }
end

csp main() begin
  solutions all;
```

```
  var
    sudoku[0..SIZE-1, 0..SIZE-1] : sType;

  constraint
  {
    sudoku[0,1] = 3; sudoku[0,2] = 4; sudoku[0,3] = 1;
    sudoku[0,5] = 5; sudoku[0,6] = 6; sudoku[0,8] = 9;

    sudoku[1,1] = 9; sudoku[1,3] = 6; sudoku[1,4] = 3;
    sudoku[1,5] = 2; sudoku[1,8] = 1;

    sudoku[2,0] = 1; sudoku[2,1] = 6; sudoku[2,2] = 5;
    sudoku[2,3] = 4; sudoku[2,5] = 9; sudoku[2,7] = 2;
    sudoku[2,8] = 3;

    sudoku[3,1] = 8; sudoku[3,3] = 5; sudoku[3,4] = 1;
    sudoku[3,5] = 6;

    sudoku[4,2] = 3; sudoku[4,3] = 8; sudoku[4,4] = 9;
    sudoku[4,7] = 6; sudoku[4,8] = 5;

    sudoku[5,0] = 9; sudoku[5,1] = 5; sudoku[5,3] = 3;
    sudoku[5,5] = 7;

    sudoku[6,8] = 6;

    sudoku[7,1] = 4; sudoku[7,5] = 1;

    sudoku[8,2] = 1; sudoku[8,4] = 5; sudoku[8,5] = 8;

    forall (i in [0..SIZE-1]:
      # Column has no duplicates
      alldiff(sudoku[i, 0..SIZE-1]);

      # Row has no duplicates
      alldiff(sudoku[0..SIZE-1, i]);
    )
    # Blocks have no duplicates
    blockCheck(sudoku);
  }
end
```

A.1.4 Magic square puzzle

```
# Find a solution for a 4x4 magic square whose columns, rows and diagonals have a
    sum of 34 each

const SIZE = 4;
const RESULT = 34;

type
  mgrid = {1..16};

csp main() begin
  solutions single;

  var
    square[0..SIZE-1, 0..SIZE-1] : mgrid;

  constraint
  {
    alldiff(square);

    forall (i in [0..SIZE-1]:
      sum(square[i, 0..SIZE-1]) = RESULT;
      sum(square[0..SIZE-1, i]) = RESULT;
    )

    sum(square[0,0], square[1,1], square[2,2], square[3,3]) = RESULT;
```

```
    sum(square[0,3], square[1,2], square[2,1], square[3,0]) = RESULT;
  }
end
```

A.1.5 Crypt-arithmetic puzzle

```
# Solves crypt-arithmetic puzzle TWO + TWO = FOUR

type
  letter = {0..9};


csp checkCarry(f : letter, t : letter, o : letter, w : letter, u : letter, r :
     letter) begin
  var
    c10, c100, c1000 : letter;

  constraint
  {
    2*o = r + 10*c10;
    c10 + 2*w = u + 10*c100;
    c100 + 2*t = o + 10*c1000;
    c1000 = f;
  }
end

csp main() begin
  solutions all;

  var
    f, t, o, w, u, r : letter;

  constraint
  {
    alldiff(f, t, o, w, u, r);
    f != 0;
    t != 0;
    checkCarry(f, t, o, w, u, r);
  }
end
```

A.1.6 Boolean SAT problem

```
# Find all the solutions for the logical expression (x and y) or (not(x) and z)

csp main() begin
  solutions all;

  var
    x, y, z : boolean;

  constraint
  {
    {
      x = true;
      y = true;
    }
    OR
    {
      x = false;
      z = true;
    }
  }

end
```

A.1.7 Pythagorean Triples

```
# Finds all Pythagorean triples in domain {1..N}. In this case, N = 250
const N = 250;

type
  pType = {1..N};

csp main() begin
  solutions all;

  var
    a, b, c : pType;

  constraint
  {
    a^2 + b^2 = c^2;
  }

end
```

Bibliography

[1] J. Bakker. "A generic solver for Constraint Satisfaction Problems". Groningen: University of Groningen, 2015. URL: http://fse.studenttheses.ub.rug.nl/13059/.

[2] P. J. Campbell. "Gauss and the eight queens problem: A study in miniature of the propagation of historical error". In: *Historia Mathematica* 4.4 (1977), pp. 397–404. ISSN: 0315-0860. DOI: 10.1016/0315-0860(77)90076-3.

[3] J. Cohen. "Constraint Logic Programming Languages". In: *Commun. ACM* 33.7 (July 1990), pp. 52–68. ISSN: 0001-0782. DOI: 10.1145/79204.79209.

[4] K. D. Cooper. "Making Effective Use of Multicore Systems A Software Perspective: The Multicore Transformation (Ubiquity Symposium)". In: *Ubiquity* 2014.September (Sept. 2014). DOI: 10.1145/2618407.

[5] B. Craenen. "Solving Constraint Satisfaction Problems with Evolutionary Algorithms". PhD thesis. Nov. 2005.

[6] R. De Landtsheer. "Solving CSP Including a Universal Quantification". In: *Multiparadigm Programming in Mozart/Oz*. Ed. by P. Van Roy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 200–210. ISBN: 978-3-540-31845-3.

[7] R. Dechter and J. Pearl. "Network-Based Heuristics for Constraint-Satisfaction Problems". In: *Search in Artificial Intelligence*. Ed. by L. Kanal and V. Kumar. New York, NY: Springer New York, 1988, pp. 370–425. ISBN: 978-1-4613-8788-6. DOI: 10.1007/978-1-4613-8788-6_11.

[8] I. Dony and B. Le Charlier. "A Program Verification System Based on Oz". In: *Multiparadigm Programming in Mozart/Oz*. Ed. by P. Van Roy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 41–52. ISBN: 978-3-540-31845-3.

[9] A. E. Eiben et al. "Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function". In: *Parallel Problem Solving from Nature — PPSN V*. Ed. by A. E. Eiben et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 201–210. ISBN: 978-3-540-49672-4.

[10] E. C. Freuder. "A Sufficient Condition for Backtrack-Bounded Search". In: *J. ACM* 32.4 (Oct. 1985), pp. 755–761. ISSN: 0004-5411. DOI: 10.1145/4221.4225.

[11] D. H. Frost and R. Dechter. "Algorithms and Heuristics for Constraint Satisfaction Problems". AAI9811432. PhD thesis. 1997. ISBN: 0591621274.

[12] I. P. Gent et al. “A review of literature on parallel constraint solving”. In: *Theory and Practice of Logic Programming* 18.5-6 (2018), pp. 725–758. DOI: 10.1017/S1471068418000340.

[13] J. Jaffar and M. J. Maher. “Constraint logic programming: a survey”. In: *The Journal of Logic Programming* 19-20 (1994). Special Issue: Ten Years of Logic Programming, pp. 503–581. ISSN: 0743-1066. DOI: 10.1016/0743-1066(94)90033-7.

[14] R. Kowalski. “The Early Years of Logic Programming.” In: *Commun. ACM* 31 (Jan. 1988), pp. 38–43. DOI: 10.1145/35043.35046.

[15] V. Kumar. “Algorithms for Constraint Satisfaction Problems: A Survey”. In: *A.I. Mag* 13 (Oct. 1998).

[16] A. K. Mackworth. “Consistency in Networks of Relations”. In: *Artif. Intell.* 8 (1977), pp. 99–118.

[17] P. Meseguer. “Constraint Satisfaction Problems: An Overview”. In: *AI Commun.* 2.1 (Jan. 1989), pp. 3–17. ISSN: 0921-7126.

[18] U. Montanari. “Networks of constraints: Fundamental properties and applications to picture processing”. In: *Inf. Sci.* 7 (1974), pp. 95–132.

[19] C. C. Rolf and K. Kuchcinski. “Distributed Constraint Programming with Agents”. In: *Adaptive and Intelligent Systems*. Ed. by A. Bouchachia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 320–331. ISBN: 978-3-642-23857-4.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597. URL: https://dl.acm.org/doi/10.5555/1671238.

[21] V. A. Saraswat and M. Rinard. “Concurrent Constraint Programming”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 232–245. ISBN: 0897913434. DOI: 10.1145/96709.96733.

[22] E. P. K. Tsang. *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press, 1993, pp. I–XVIII, 1–421. ISBN: 978-0-12-701610-8.