Optimal decision-making under constraints and uncertainty

Latour, A.L.D.

3

Programming paradigms for optimisation

3.1 Introduction

Recall from Section 1.3 that exact *stochastic constraint (optimisation) problem (SCP)* solving involves two components: probabilistic inference and search. While in the previous chapter we discussed how to model probability distributions and typical inference tasks, in this chapter we discuss programming paradigms that help us model probabilistic optimisation problems, and on programming paradigms that help us traverse the search space of possible strategies efficiently. Specifically, in the next section, we discuss a ProbLog-derived tool for solving decision problems that involve probabilities, DT-ProbLog [178]. Then, in Sections 3.3 and 3.4, we discuss the fields of *constraint programming (CP)* and *mixed integer programming (MIP)* as optimisation paradigms that allow us to efficiently traverse search spaces.

Recall also from Section 2.1 that the work presented in this dissertation is broad in scope, meaning that we can build on many and varied existing tools, each with its own set of tunable parameters. Consequently, bringing these tools together in an effort to solve SCPs may result in complex systems that may require specialised configurations for different application domains for them to perform well. Therefore, later in this chapter, we provide a brief introduction to the paradigms of *programming by optimisation (PbO)* and *automated algorithm configuration (AAC)*, which enable us to find these configurations. We close this chapter in Section 3.6 with a brief conclusion.

3.2 Decision-theoretic probabilistic logic programming

We start this overview of programming paradigms for optimisation with yet another Prolog-based modelling language (recall Section 2.6): DT-ProbLog (short for *decision-theoretic ProbLog*) [178]. This extension of ProbLog [52] was proposed in 2010 in order to solve decision problems in which the aim is to maximise some kind of utility that is associated with (derived) facts. To this end, DT-ProbLog extends the ProbLog syntax and semantics by adding *decision variables* and functionality to assign a *utility* to events.

By setting decision variables to *true*, we can introduce facts to the program. Utilities associate with facts can be used to define the relative costs and benefits of facts and consequences. We illustrate this with the following example.

Program 3.1: *A DT-ProbLog program describing a spread of influence problem.*

```
    % Background knowledge
1.  person(alexa).              person(claire).
2.  person(behrouz).            person(daniel).

    % Probabilistic relation facts
3.  0.4::dir(alexa,behrouz).        0.8::dir(alexa,claire).
4.  0.1::dir(behrouz,claire).       0.3::dir(claire,daniel).

    % Relation rules
5.  influences(X,Y) :- dir(X,Y).
6.  influences(X,Y) :- dir(Y,X).

    % Decisions
7.  ?::gets_free_sample(P) :- person(P).

    % Utilities
```

```
 8. utility(buys(P), 1) :- person(P).
 9. utility(gets_free_sample(P), -1) :- person(P).

    % Probabilistic customer conversion rules
10. buys(X) :- gets_free_sample(X).
11. buys(X) :- influences(Y,X), buys(Y).
```

Example 3.2.1 (A simple DT-ProbLog program)**.** *We can modify and extend the probabilistic logic program described by Programs 2.1 and 2.3 by adding decision variables and utilities, resulting in Program 3.1. Note that this program describes a spread of influence problem like the one described in Section 1.1.*

Here, line 7 specifies the decision variables — a functionality that is new in DT-ProbLog. Lines 8 and 9 specify utilities for (derived) facts. Specifically, they state that converting a person into a customer who buys our product, has a utility of 1 per person (line 8). However, it costs 1 to give a person a free sample of our product (line 9).

DT-ProbLog does not provide a specific syntax for querying the program, but returns grounded facts that represent the decisions that have to be made in order to maximise the expected utility, along with the value of that utility. This utility is computed by simply summing the (expected) utilities present in or derived from the DT-ProbLog program. In the example above, the output of DT-ProbLog is `gets_free_sample(alexa) := True, gets_free_sample(behrouz) := False, gets_free_sample(claire) := False, gets_free_sample(daniel) := False, Expected utility: 1.4984`.

It comes to this conclusion based on an inference process that uses *algebraic decision diagrams* (ADDs) [11], data structures that are very similar to the *arithmetic circuits (ACs)*, described in Section 2.5, and can be used for optimisation problems. These can be constructed from *decision diagrams (DDs)* like *ordered binary decision diagrams (OBDDs)* and *sentential decision diagrams (SDDs)*.

These properties make DT-ProbLog an attractive programming paradigm for stochastic optimisation problems, for us to build on in our efforts of designing SCP solving methods.

Note that we do have to build on DT-ProbLog before we can use it to solve SCPs, since DT-ProbLog itself does not support constraints, only a maximisation criterion.

3.3 Constraint programming

As discussed in Section 1.3, in order to solve SCPs we do not just need efficient probabilistic inference, we also need an effective search mechanism. One of the

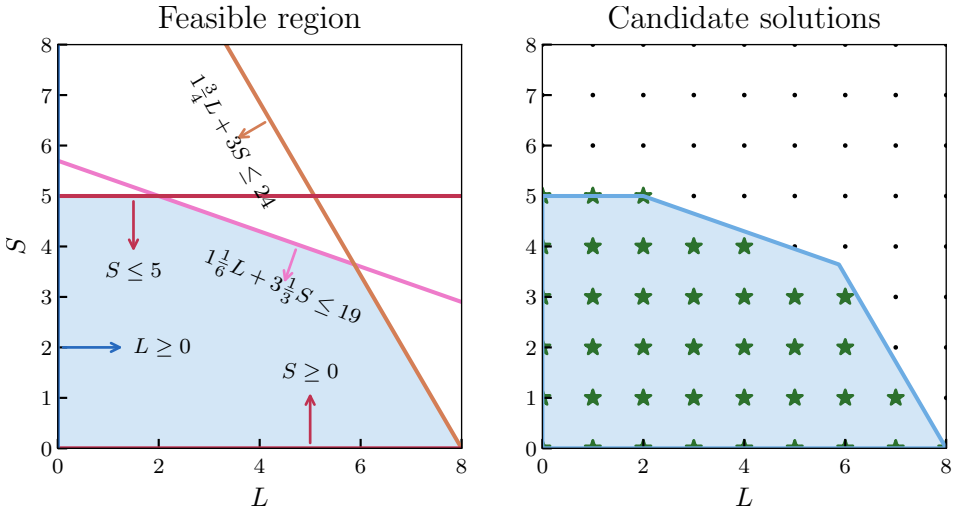

Figure 3.1: *Graphical representation of the constraints in the power grid reliability problem described in Example 3.3.1. The horizontal axis shows the number of power lines that we can choose to reinforce, and the vertical axis shows the number of stations. Stars indicate (L, S) combinations that satisfy the constraints and are thus candidate solutions of the optimisation problem.*

paradigms we employ in this work is CP. Here, we recall basic concepts of CP. For details we refer the reader to the literature, *e.g.*, the *Handbook of Constraint Programming* [154].

Constraint satisfaction problems (CSPs) are modelled using a set of variables $\mathbf{X} = \{X_1, \ldots, X_{|\mathbf{X}|}\}$, each of which is associated with a domain $dom(X_i)$, a set of constraints C on (subsets of) these variables and an objective function $f(\mathbf{X})$. In the context of this work, the objective function can be, *e.g.*, $\arg\max_\sigma P(\phi|_\sigma)$, or the linear constraint optimisation function in the next example, which we will consider for the scope of this section and the next, to illustrate some key concepts of optimisation techniques.

Example 3.3.1 (A power grid reliability problem)**.** *Recall the power grid reliability problem from Section 1.1. Suppose we drastically simplify this problem by defining some measure of 'reliability' and assuming that this is some linear combination of the number*

of power lines and stations that we have reinforced.

$$\textit{maximise reliability} \quad R = \frac{5}{4}L + S \tag{3.1}$$

$$\textit{subject to budget constraint} \quad \frac{7}{6}L + \frac{10}{3}S \leq 19 \tag{3.2}$$

$$\textit{and time constraint} \quad \frac{7}{4}L + 3S \leq 24 \tag{3.3}$$

$$\textit{with} \quad L, S \in \mathbb{N}_0, L \leq 10, S \leq 5 \tag{3.4}$$

where L and S are the numbers of power lines and power stations that we have reinforced. They require different amounts of money ($\frac{7}{6}$ million and $\frac{10}{3}$ million of your favourite currency, respectively), with a budget of 19 million. They also require different amounts of time per unit to reinforce ($\frac{7}{4}$ and 3 months, respectively), where we assume that there is only one team that does the reinforcements and they can only reinforce one unit at a time, and we have two years to complete the project. Finally, our small network has ten power lines and five power stations.

We provide a graphical representation of this problem in Figure 3.1.

In general, variables can have different kinds of domains (typically Boolean, categorical, ordinal, integer- and real-valued). In the example above, the variables have integer domains. In the SCP problems studied in this work, the relevant variables are all Boolean decision variables. However, the choices we make in Chapter 5 to model SCPs in CP solvers, result in constraint programs that also contain variables with real-valued domains.

In the next subsections, we continue with a discussion of two orthogonal solving techniques for CP: *search* and *inference*. Intuitively, the search process determines how the solver traverses the search space, by assigning values to variables to see if those variable assignments can be extended to a solution. The inference process, called *propagation*, helps to prune the resulting search tree by efficiently *inferring* consequences of these assignments. Propagation detects which values in the domains of free variables have to be removed from those domains because they are *inconsistent* with the assignments made by the search process, and thus can never lead to a solution.

This brings us to another important dichotomy that is identified in the CP literature: modelling versus solving. CP solvers provide the user with a range of different constraints to choose from. Each of these constraints has an associated *constraint propagator*: an algorithm that can efficiently *infer* consequences when variable domains change, due to choices made during the search process. We will describe propagation in more detail later in this section.

Since not all propagators are equally powerful, meaning that they cannot always make the same inferences using the same amount of computational effort, the actual choice of which constraints to use in *modelling* the problem can have a significant impact on how efficiently the problem can then be *solved*. Therefore, in order to successfully employ CP techniques, the user has to be smart about how they model the problem. A detailed discussion of (how to make good) modelling choices is outside the scope of this work, but we do now continue with a brief overview of search and propagation techniques.

3.3.1 Backtracking search

CP solvers employ one core algorithmic approach to finding a solution (or refutation) to the input problem: *backtracking search*. While CSPs can also be solved by techniques like *local search* and *dynamic programming*, we focus on backtracking search.

In this work, we require the search algorithm to be *complete*, meaning that it guarantees that a solution will be found if one exists, that it will show that a problem does not have a solution if none exists, and that it can be used to find a provably optimal solution. The backtracking search approach is complete (unlike most local search algorithms) and typically preferred over the (complete) dynamic programming approach, because backtracking requires only a polynomial amount of space, while dynamic programming might require exponential amounts of time and space [154, Chapter 4]. Backtracking search was first proposed in the 1960s [49, 73] and is still the main driver of modern-day CP solvers.

Typical backtracking search uses a depth-first search, inducing a *search tree* to find a solution to the CSP (or an optimal solution in an optimisation setting). The solver repeatedly selects an *unbound* (or *uninstantiated*, or *free*) variable X and assigns to it a value $a \in dom(X)$ (or a range or interval of values in case, *e.g.*, $dom(X) \subseteq \mathbb{R}$), thus building a *partial solution*. Repeatedly selecting an unbound variable and assigning a value to it is called *branching* and induces a *search tree*.

If the domain of a variable X is reduced to a single value in this process, we consider X to be *fixed* (or *bound*) to that value. If a fixed variable or set of fixed variables violate(s) a constraint, we have encountered a *failure*: a partial assignment that cannot be extended to a solution. When this happens, the solver *backtracks* to an earlier point in the search tree by undoing variable assignments. The two main backtracking methods are *chronological backtracking*, where the solver simply returns to the closest node on the current path in the search tree where not all outgoing branches have been explored yet, and *non-chronological backtracking* or *backjumping*, where the solver 'jumps' back to a higher level in the search

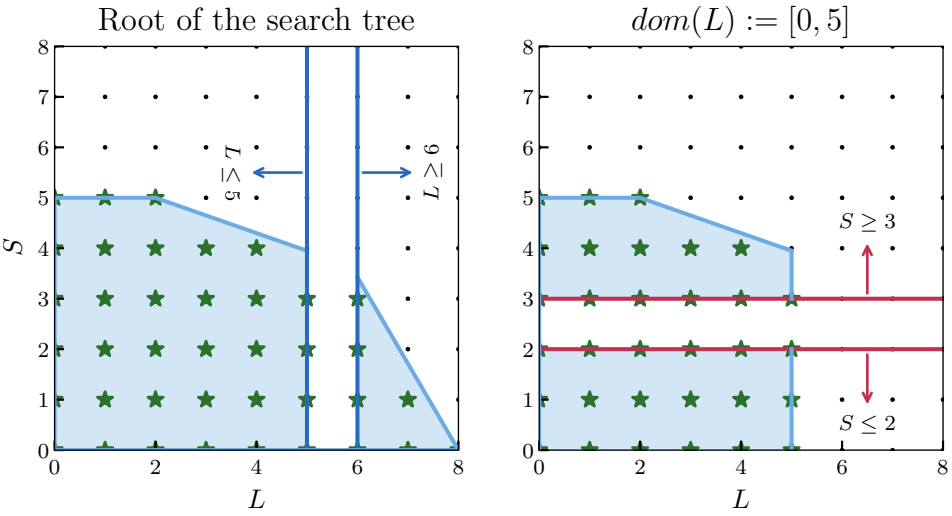

Figure 3.2: *Graphical representation of branching in the root and in the left child of the root in Figure 3.3.*

tree, thus skipping one or more nodes whose outgoing branches have not all been explored yet.

Let us now use the constraint optimisation problem in Example 3.3.1 to illustrate backtracking search in CP.

Example 3.3.2 (Branching and backtracking in CP)**.** *The constraint optimisation problem described in Example 3.3.1 has two variables: the number of power lines that we choose to reinforce L, and the number of power stations that we choose to reinforce S. A search algorithm will thus have to traverse the search space of (L, S) combinations to find the combination that satisfies Equations* 3.2 *to* 3.4 *and maximises Equation* 3.1*.*

Figure 3.3 illustrates just one way of traversing that search space. In each node, the domain of either L or S is split (roughly) in half, branching left on the lower side, and right on the higher side of the domain. Going down along a branch on the tree, nodes in which we split on the domain of L and the domain of S alternate. Let us assume we traverse the tree from left to right, and let us in this example only consider the left part of the search tree (rooted at the left child of the root).

Following the left-most branch, we first branch on $L \leq 5$ and then on $S \leq 2$. We can visualise this as adding extra constraints to focus on a specific region of the search space, as is illustrated in Figure 3.2. Note that, at this point in the search tree, with $dom(L) = [0, 5]$ and $dom(S) = [0, 2]$, all (L, S) combinations satisfy the constraints in Equations 3.2 *to* 3.4*. For the sake of legibility, we have omitted further descendents of this node and just listed the optimal combination in these domains: $R(L = 5, S = 2) = 8.25$.*

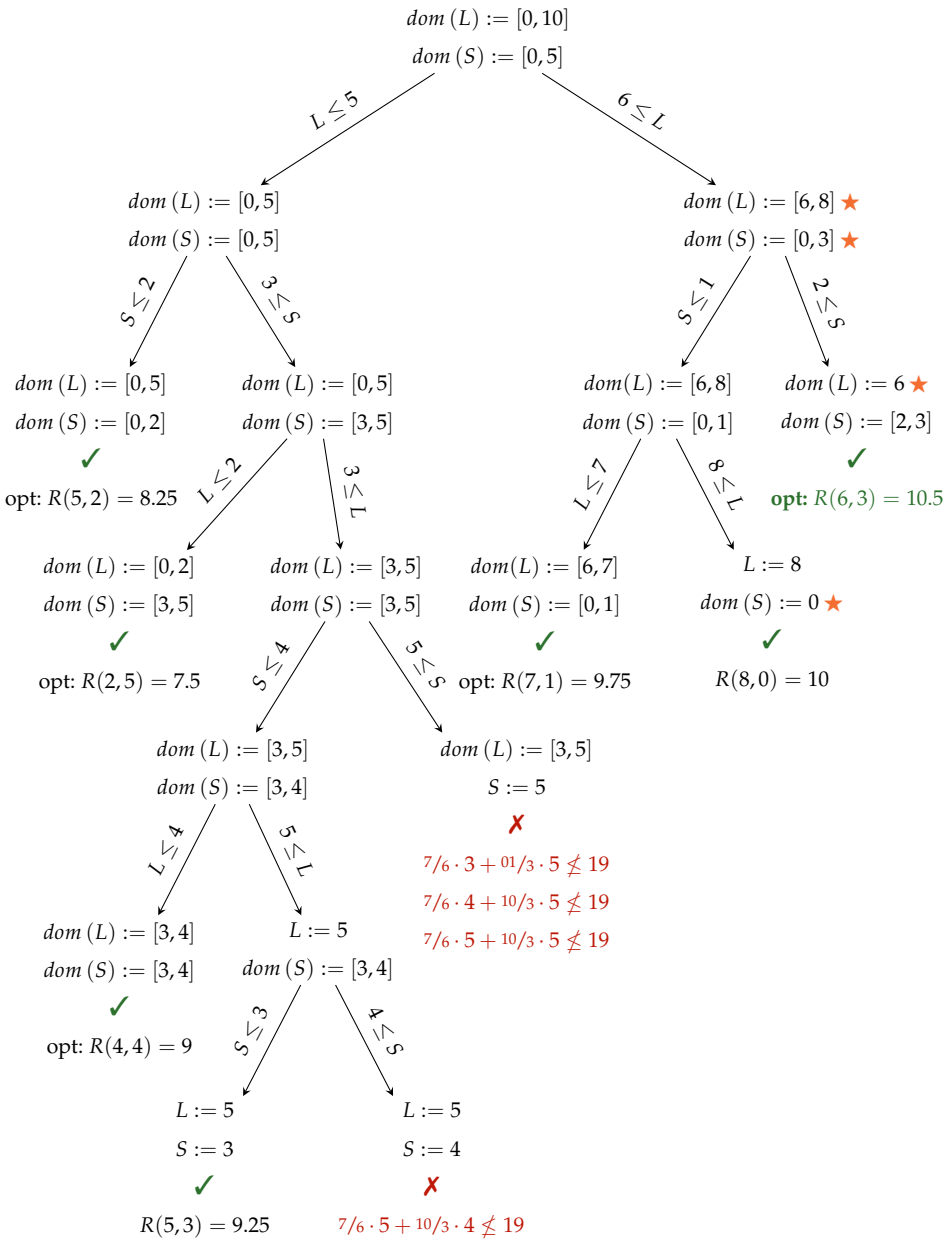

Figure 3.3: *An example of a small search tree for the constraint optimisation problem described in Example 3.3.1. We use ✗ to indicate failures and ✓ to indicate a solution to the constraints in Equations 3.2 to 3.4. Domains marked with ★ are relevant for Example 3.3.3.*

After finding this solution, we backtrack to the parent node and branch right, on $3 \leq S$ *to continue the search. We continue branching and finding solutions, until the node in which both domains have been reduced to just one value:* $L := 5$, $S := 4$, *which is a combination that violates the budget constraint in Equation* 3.2. *We backtrack and continue the search for a better solution than the best one found so far (*$R(L = 5, S = 3) = 9\frac{1}{4}$*), until we have traversed the entire search tree.*

Typically, CP solvers employ a process called *branch and bound*, which uses cheap-to-compute heuristics to determine if branches of the search tree can still contain a better solution than the best solution found so far, or even any solution at all. This helps to keep the size of the search tree from becoming too large.

In the above example, we made a rather arbitrary choice to alternate between splitting the domains of L and S and to always split them (roughly) in half. Which variable (and value or domain) to branch on next, is typically decided by a *branching heuristic*. The aim of these heuristics is to find a variable and value/domain branching order that minimises the size of the search tree, and thus likely also the running time of the solver. Since even finding the *first* variable of an *optimal* variable order is at least as hard as solving the input problem itself [107], finding and using the optimal variable order is infeasible.

Therefore, CP solvers use heuristics that give no guarantees of optimality to decide which (variable, value/domain) pair to branch on next during the search. These heuristics can either be universal or domain-specific, and can be either static (determined before the search starts) or dynamic (determined during the search). In this work we use existing universal branching heuristics, and present new ones that are either static or dynamic and designed specifically for a new constraint propagator that we introduce in Chapter 6. Additionally, we introduce static, domain-specific branching heuristics in Chapter 7.

3.3.2 Constraints, consistency and pruning

The second mechanism that drives typical CP solvers is *constraint propagation*, or *inference*, which is orthogonal to search. Informally, constraint propagation helps the solver to eliminate *inconsistencies*, which are values in domains of free variables that cannot be part of a solution to the CSP, given the current assignment to the bound variables. By enforcing consistency, constraint propagation helps the solver to avoid branching on (variable, value) pairs that are inconsistent with the current partial assignment and the constraints, and thus to prune parts of the search space that do not contain any solutions, reducing the size of the search tree.

Constraint propagation operates as follows. After each time a backtracking search solver branches on a (variable, value/domain) pair, propagators update the domains of the remaining unbound variables by removing values that would violate the constraints of the problem instance, given the current partial solution. This helps to shrink the domains of the remaining free variables, and thus to prune the search tree. If the domain of a variable X is reduced to a single value in this process, this variable is automatically bound to that value. If for any variable X we find that $dom(X) = \varnothing$ after propagation, we have found a failure and must backtrack. Note that constraint propagation can also be called *before* the search starts, in which case it serves as a preprocessing step to reduce the sizes of variable domains [154, Chapter 4].

An important type of consistency is that of *generalised arc consistency (GAC)*. A (variable, value) pair (X, x) with $x \in dom(X)$ is considered *generalised arc consistent (GAC)* with respect to a constraint $c \in C$ iff there exists an assignment in the current domains of the other variables in the scope of c that satisfies c and in which $X = x$ [114]. Propagation establishes GAC for a constraint c if all remaining values of all variables in the scope of c are GAC.

Example 3.3.3 (Search space pruning in CP)**.** *Note that the left side of the search tree in Figure 3.3 is quite large, despite us omitting many nodes for reasons of legibility and instead simply giving the optimal value of the objective function possible for the domains of L and S in that node. Crucially, in Example 3.3.2, we did not perform any propagation. Had we done any propagation, we could have pruned the search space.*

Consider the right branch of the root of the search tree in Figure 3.3. It branches on $6 \leq L$*, reducing L's domain from* $[0, 10]$ *to* $[6, 10]$*. A quick glance at Figure 3.1 tells us that any L that exceeds* 8*, violates the time constraint in Equation* 3.3*. A constraint propagation algorithm may detect that this is the case, and exclude the values* 9 *and* 10 *from L's domain, which we have done in the right child of the root of the search tree. Similarly, for values of L that are larger than* 4*, there are no solutions in which* $S = 4$ *or* $S = 5$*. Since we branched on* $L \geq 6$ *in the right child of the root, we can also exclude the values* 4 *and* 5 *from the domain of S, which we have also done in that node.*

Note that these actions kept the domains of L and S GAC with respect to the budget constraint in Equation 3.2*, and the time constraint in Equation* 3.3*, respectively. The domains in Figure 3.3 that have been pruned using propagation, are marked with* ★*.*

3.3.3 Local and global constraints

A detailed discussion of backtracking and propagation, as well as of other techniques, such as randomisation, restarts, local search, value selection heuristics

and more, is outside the scope of this work. However, we must mention a few important concepts about *local* and *global* constraints.

The main difference between local and global constraints is their scope. Local constraints are between a *fixed* number of variables. For example: $X < Y$ is a local constraint, since it is always between two variables.

Global constraints, on the other hand, can involve an arbitrary subset of the variables present in a problem. Arguably the best-known global constraint is the *AllDifferent* constraint, which requires all variables in the scope of the constraint to have a different value. Note that, contrary to the constraint above, the size of the scope of an AllDifferent constraint is not determined by the form that the constraint takes.

The constraint that is central to this work, the one in Equation 1.1, is a global constraint. The constraints in Equations 3.2 to 3.4 on the other hand, are local constraints.

Note that enforcing GAC for global constraints can be more computationally expensive in both time and space than enforcing consistency on local constraints, but also potentially more powerful in the amount of pruning it makes possible. Some work has been done on *decomposing* global constraints such that the decomposition has the same propagation power as the original global constraint, meaning that the decomposition prunes the same values from the domains of the involved variables as does the original global constraint. This is possible for some global constraints, but not for all (at least not in polynomial time and polynomial space). We refer the interested reader to the literature on which global constraints can be decomposed, whether those decompositions preserve the solutions to the CSP, whether they preserve GAC, and whether they preserve the time and space complexity of enforcing GAC, *e.g.*, [10, 15] and [154, Chapter 3].

3.3.4 Advantages of CP technology

CP solvers typically support many different types of constraints, where each constraint has a dedicated propagator, designed specifically to propagate changes in domains of variables in the scope of that constraint efficiently. As one propagator removes values from a variable's domain, this may trigger other propagators to also remove values from domains. Thus, even though propagators themselves are designed to solve specific constraints, their interaction can be quite powerful in finding solutions to the input problem very quickly. A user simply has to specify the relevant constraints, and the dedicated propagators take care of the inference tasks.

Note that, while we have so far only discussed CSPs, we can also use CP for

solving constraint *optimisation* problems. We can straightforwardly turn a constraint optimisation problem into a CSP as follows.

Suppose we have an optimisation problem with non-negative objective function $f(\mathbf{X})$ and constraint $c(\mathbf{X})$. The first step is to turn the objective function into a constraint, by setting $f(\mathbf{X}) > 0$. If there exists a solution to the resulting CSP, $\mathbf{X}_{\text{sol}}$, it has value $\theta_{\text{sol}} := f(\mathbf{X}_{\text{sol}})$. We now update the constraint we derived from the objective function to $f(\mathbf{X}) > \theta_{\text{sol}}$ and *continue* the search. Note that, because $\mathbf{X}_{\text{sol}}$ represents the first solution we found, we do not have to restart the search, but can simply continue building the existing search tree, now in pursuit of a new solution $\mathbf{X}_{\text{new}}$ such that $f(\mathbf{X}_{\text{new}}) > \theta_{\text{sol}}$. We continue this process until the CSP becomes infeasible, in which case the last solution that was found represents the solution to the original constraint optimisation problem. This makes CP a declarative, flexible, convenient, general and fast programming paradigm for modelling and solving a wide range of problems.

Unsurprisingly, therefore, the CP community has produced a wide range of tools for modelling and solving CSPs, of which we name a few here. First of all, as described above, CSPs must be modelled before they can be solved. MiniZinc is a free and open source tool, especially designed to model CSPs in a high-level and solver-independent way.[1] All solvers that we name in this section can not only solve CSPs modelled with MiniZinc, but also have an interface that directly connects MiniZinc to the solver.

The powerful open source C++ toolkit Gecode[2] has proven to be a time- and memory-efficient CP solving tool for well over a decade, winning all gold medals in all categories of the MiniZinc Challenge[3] five years in a row. IBM's commercial ILOG CP Optimizer provides state-of-the art support for both real-world, and purely academic constraint optimisation problems.[4] The ILOG-inspired, and recent MiniZinc Challenge gold medallist, Scala library OscaR [132] provides an open source toolkit for constraint solving and constraint optimisation, including functionality for visualising the search tree.[5] Google's OR-Tools provides Python, C++, Java and C# interfaces for users to model CP problems and then solve them by solvers such as CP-SAT.[6] The open-source constraint logic programming system ECLiPSe was particularly designed to be a generic programming tool, especially suitable for rapid prototyping, and provides a Python interface.[7]

[1] Available at `www.minizinc.org`.

[2] Available at `www.gecode.org`.

[3] See `www.minizinc.org/challenge.html`.

[4] Available at `www.ibm.com/analytics/cplex-cp-optimizer`.

[5] Available at `bitbucket.org/oscarlib/oscar`.

[6] Available at `developers.google.com/optimization`.

[7] Available at `eclipseclp.org` and `pyclp.sourceforge.net`.

The discussion above represent just a small selection of the wide range of CP solvers available. Later in this work, in Chapter 5, we will explore how we can use off-the-shelf CP solvers to solve SCPs. Then, in Chapter 6, we present two variants of a propagation algorithm that is specifically designed for a special kind of stochastic constraint.

3.4 Mixed integer programming

As an alternative to CP solvers, we can also employ *mixed integer programming (MIP)* solvers to solve the SCPs studied in this work. We recall basic concepts of MIP. For details we refer the reader to the literature, *e.g.*, Bradley *et al.*'s *Applied Mathematical Programming* [25].

3.4.1 Mixed-integer linear programs

Again, we can model discrete optimisation problems with a set of variables, corresponding domains, a set of constraints and an objective function. Unlike CP solvers, MIP solvers support a limited range of different constraints. *Mixed integer-linear programming (MILP)* solvers – arguably the most widely used type of MIP solvers – support only linear constraints; as a result, they can only be used for solving problems that can be modelled using linear constraints. Note that even MILP is $\mathcal{NP}$-hard [71].

The constraint optimisation problem in Example 3.3.1 also happens to be a MILP, since both the constraints and the objective function are simply linear combinations of variables.

3.4.2 Solving a MILP

Despite the limitation of only being able to deal with *linear* constraints, MILP solvers can be more powerful than CP solvers in solving linear programs, because of their ability to *relax* MILPs. Relaxing a MILP instance means relaxing the integrality constraint of the decision variables with integer domains, meaning that they are allowed to take real values. The resulting linear program has an optimal solution that is guaranteed to be on one of the corner points of the convex hull of all feasible solutions. This is illustrated in the left figure of Figure 3.4. It shows the feasible region, where all constraints are satisfied, and shows the only allowed solutions, which are the integer ones. The optimal continuous solution is on the outer hull of the feasible region, and is indicated in the figure. The figure

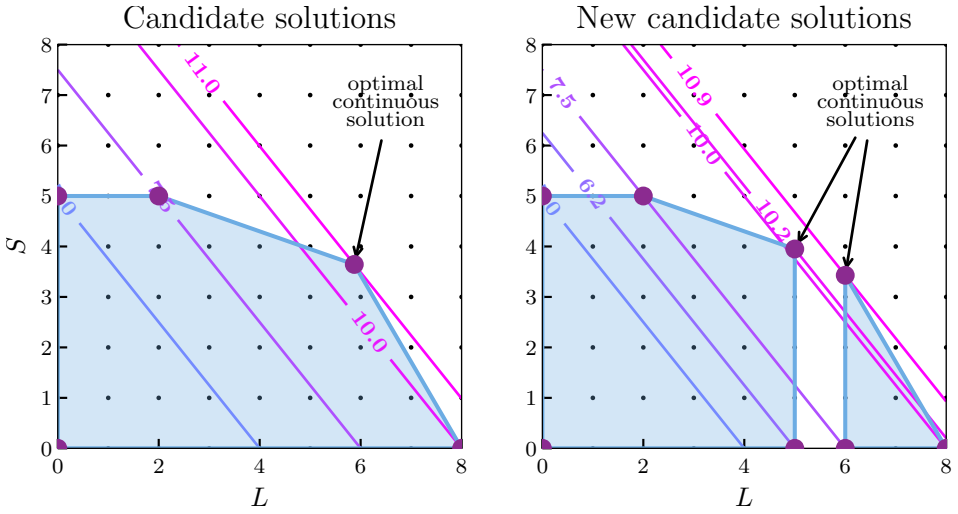

Figure 3.4: *Candidate solutions (solid disks) on the outer hull of the feasible region of the MILP in Example 3.3.1, along with the corresponding values of the optimisation criterion in Equation* 3.1 *(diagonal lines). In the left figure there is one optimal continuous solution. In the right figure, there is one optimal continuous solution for each sub-domain of* L *that is obtained after branching.*

also shows the values of the reliability function (Equation 3.1) for the different corner points of the convex hull.

Since the optimal continuous solution is usually not a valid solution because of the integrality constraints on L and S in Example 3.3.1, a MILP solver must narrow down the space of feasible optimal solutions, until it finds an integer one. Note that simply rounding the continuous solution to an integer one may not be feasible, since that might violate constraints. MILP solvers employ three main techniques for this [25, Chapter 9]: *branch-and-bound*, *cutting planes* and *group-theoretic* approaches. We will give a short intuition for how the first two techniques work, again using the example problem in Example 3.3.1.

The branch-and-bound technique is also employed by CP solvers and simply adds extra constraints to recursively narrow down the search space to an integral one. This process induces a search tree, which allows for pruning similar to what we described in Section 3.3.1.

Example 3.4.1 (Branch-and-bound)**.** *Consider the left figure in Figure 3.4. The MILP solver has used the knowledge that the optimal continuous solution is on a corner points of the convex outer hull of the feasible region, which in this case happens to be at* $(L = {}^{1122}/_{191}, S = {}^{696}/_{191})$*. Using this as a starting point for the search, it might make*

sense to split the problem into two parts by adding two more constraints, either:

$$S \leq 3 \quad \textit{and} \quad S \geq 4, \quad \textit{or} \quad L \leq 5 \quad \textit{and} \quad L \geq 6.$$

Note that neither choice excludes any integer solutions from the feasible region.

In the figure, we have chosen to first branch on the latter set of constraints, similar to what we did in Example 3.3.2 and the left plot of Figure 3.2. This divides the search space up into two sub problems, that can be solved individually. In the left part of Figure 3.4, we have indicated the reliability values of Equation 3.1 *for the corner points of the convex hulls, and the new optimal continuous solutions.*

Note that, if we now continue with the $L \geq 6$ part of the sub-problem, we can immediately refine the optimal continuous solution of $(L = 6, S = {}^{25}/_{7})$ to $(L = 6, S = 3)$. We do this by noting that the optimal continuous solution is not in between two integer solutions, but that its value for S can be rounded down to an integer solution, and obtain $R(6, 3) = 10.5$. Since this value is higher than the optimal continuous solution of the $L \leq 5$ part of the search space ($R\,(5, {}^{79}/_{20}) = 10.2$), we have found the optimal integer solution and do not have to explore that part of the search space.

An alternative way of narrowing down the space of feasible optimal solutions, is the *cutting plane* technique, introduced by Gomory in the 1950s [76], which is used in all modern MILP solvers. After finding an optimal solution to the relaxed MILP, the MILP solver checks if the decision variables in that solution take integer values. If not, it introduces a new linear constraint (cutting plane), separating this solution from the convex hull of feasible solutions to the MILP. Then, it solves the resulting (relaxed) linear program, obtains a new optimal solution, checks it for integrality, and so on. Plenty of research effort has been spent on creating techniques for finding cutting planes that are fast to compute and that reduce the feasible region by as much as possible, without excluding any integer solutions. While a detailed discussion of these efforts is outside the scope of this work, we illustrate the cutting planes technique with the following example.

Example 3.4.2 (Cutting planes)**.** *We illustrate the cutting planes technique in Figure 3.5. The red line in the left figure represents the cut. The part of the feasible region that is above the cut does not contain any integer solutions, and can thus safely be cut off from the feasible region, thus allowing the solver to narrow down its search.*

In the right figure we have indicated the reliability scores for the corner points of the new outer hull of the feasible region. Note that the new optimal continuous solution, $R\,({}^{108}/_{17}, {}^{48}/_{17}) = 10.8$, is smaller than the optimal continuous solution in the right plot of Figure 3.4.

Many MIP solvers grew from extending CP solving techniques such as

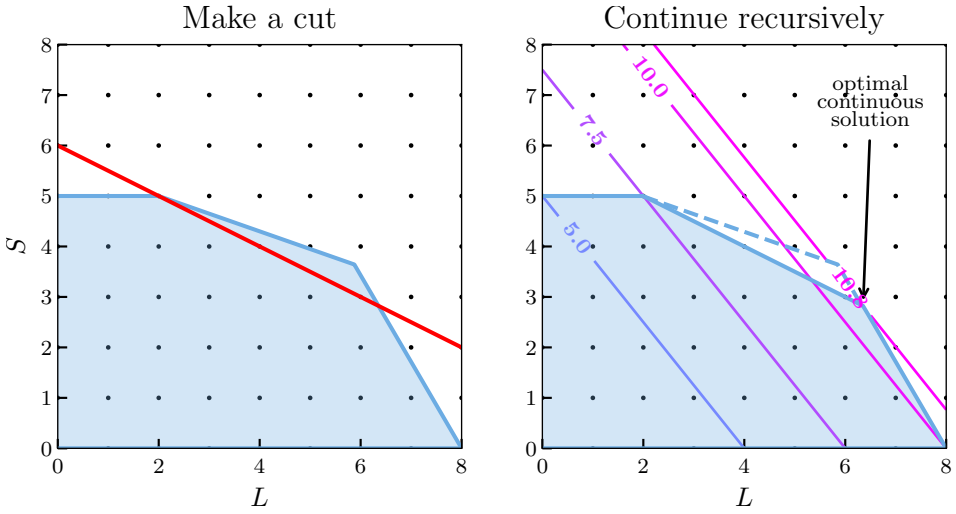

Figure 3.5: *Visualisation of the cutting planes technique. The red line in the left figure cuts off a part of the feasible region, without excluding integer solutions.*

branch-and-bound with MIP solving techniques such as cutting planes. One example of a (commercial) solver (with free academic license) that evolved in this way is IBM's CPLEX Optimizer.[8] After extensive integration of cutting planes techniques into this solver in 1999, it showed a dramatic decrease both in solving time and in optimality gap on MIPLIB examples.[9] Another commercial MIP solver with free academic license, Gurobi, provides a wide range of cutting plane techniques, whose parameters can be tuned either by hand or by Gurobi's automated parameter optimiser.[10] Some systems, such as the non-commercial SCIP Optimization Suite[11], or Google's OR-Tools[12] offer a general framework for modelling MIPs and then providing an interface to the user to have the resulting programs solved by other MIP solvers.

3.4.3 Quadratic programs and linearisation

Later, in Chapter 5 we will demonstrate how to encode SCPs as MIPs. The encodings that we use result in MIP models that are *not* linear, but contain quadratic constraints. While state-of-the-art MIP solvers, such as CPLEX and Gurobi, can

[8] Available at `www.ibm.com/analytics/cplex-optimizer`.
[9] Available at `miplib.zib.de`.
[10] Available at `www.gurobi.com`.
[11] Available at `scipopt.org`.
[12] Available at `developers.google.com/optimization`.

also deal with quadratic constraints under certain conditions, we limit ourselves to MILPs; we do this, because those conditions are currently not guaranteed by all MIP encodings obtained from SDD representations of probability distributions.

Under certain circumstances, it is possible to *linearise* quadratic constraints. We will reflect on how to linearise quadratic constraints that are obtained from OBDD or SDD representations of constraints on probability distributions in Chapter 5. While linearisation typically comes at the cost of increasing the size of the model, it may very well be worth it, because linearised models are potentially very quick to solve by a MILP solver, because of the relaxation, branch-and-bound and cutting planes techniques described above.

3.5 Programming by optimisation

In this work, specifically in Chapters 5 and 6, we introduce *modular* methods for solving SCPs. This is in large part a consequence of the fact that this work is broad in scope. In this and the previous chapter, we have discussed relevant techniques from the fields of propositional logic, probabilistic inference, knowledge compilation, logic programming, CP solving and MIP solving. These different fields have their own states of the art, implemented in different tools and solvers. Thus, if we want to combine the crème de la crème of the technologies brought forth by these fields, a promising attempt at combining them into SCP solvers may be to click them together like LEGO bricks, building SCP solvers in a modular manner.[13]

3.5.1 One size does not really fit anybody

This prompts us to employ yet another programming paradigm for optimisation: *programming by optimisation (PbO)* [80].

Just like LEGO bricks come in different colours, so we can choose which colour to use every time we add one to the thing we are constructing, there are of-

[13] LEGO bricks are things that, when stepped on while barefoot, induce a hellish pain that requires excessive screaming to soothe. Many wheelchair users remain blissfully unaware of the pain inflicted by these specific instruments of torture. Optionally, the small, brightly coloured, interlocking plastic bricks can be used by children and adults alike to construct various objects. Anything constructed can be taken apart again, and the pieces reused to make new things. Much like this dissertation, a LEGO brick presents a choking hazard to anyone unwise enough to stick it in their mouth. Unlike a LEGO brick, however, this dissertation presents a challenge to anyone attempting to stick it far enough up their nose (or anybody else's) to require a hospital visit for its removal from the relevant nose. At the time of writing, these last two statements remain purely speculative, since this dissertation has not been printed yet. That being said, we do not encourage the reader to attempt an empirical verification or falsification of the truth of those statements, even after this dissertation has gone to print.

ten multiple available solutions for solving the same sub task in our SCP solving pipelines. For example, for modelling probability distributions, we could either choose to model them with an OBDD or with an SDD. These *design choices* have no effect on correctness, but can affect performance, especially for computationally challenging problems, such as SCPs.

Note also that one-size-fits-all solutions are rare in this world. There is a reason that LEGO bricks come in different shapes and sizes. We often find that certain approaches work well for solving problems from one domain, but are much less suited to solve problems from another domain. For example: branching heuristics in CP solvers may be domain-specific. However, in practice, only one of these *design choices* is implemented in the final version of an algorithm or software system. The choice is often made based on limited experimentation, with a specific application in mind.

In this work, we try to avoid making that mistake. Rather, we want to exploit the fact that there are often multiple possible ways of achieving (sub)tasks readily available for us to use. As we will describe in Chapter 7, we have therefore constructed various parts of our SCP solving pipeline in such a way that it has access to different methods for solving subtasks and can be tuned for problems from specific application domains.

This approach, implementing different design choices such that the configuration of the resulting solver can be optimised for specific problem types is called PbO [80].

3.5.2 Automated algorithm configuration

Taking a PbO-based approach to software or algorithm design, developers provide the end user with the choice between these options, by exposing them as configurable parameters. A potential downside of this is that the user is left with myriad choices of possible parameter settings (the *configuration* of the algorithm or software system), with even more possible combinations. An end user might not have the specialised expertise to make an optimal choice of these parameter settings, while the algorithm's configuration can have a substantial impact on its performance. Additionally, the optimal configuration may vary for different sets of problem instances.

This also applies to many existing state-of-the-art algorithms that naturally come with many parameters. Using suitable parameter settings is then critical for reaching state-of-the-art performance — especially for $\mathcal{NP}$-hard problems, such as ones studied in this work.

A solution to this problem lies in *automated algorithm configuration (AAC)* [79],

which is the process of automatically finding an optimised configuration of an algorithm's parameters for solving problem instances from a specific problem set, and critically enables PbO-based algorithm design.

After applying AAC to a *target algorithm* A with parameters $q_1, \ldots, q_n$ on a set of problem instances I, we obtain a configuration c^* that is expected to perform well, according to a given performance metric m, on new instances that are similar to those in I. In this work, since we are studying exact optimisation methods and therefore cannot optimise for, *e.g.*, quality of approximation, our performance metric is always running time.

There are two main types of configurators [86]: *model-free* configurators, such as (iterative) F-Race [12, 17] and paramILS [87, 88], and *model-based* configurators, such as SMAC [86] and GGA++ [3].

Model-free configurators are relatively simple. For example, in its most basic form, the well-known configurator F-Race operates by first choosing a set of configurations according to some kind of distribution, and then 'racing' them against each other to see which solves the problem instances the best, according to the performance metric [174]. Once it becomes clear that a configuration is too far behind the others to ever catch up, it is eliminated from the set of candidate configurations. At the end of its configuration run, F-Race returns a set of 'elite' configurations whose performances are statistically equivalent to each other, and statistically better than performances of the configuration outside the set of 'elite' configurations. paramILS [87], on the other hand, uses a process of iterated local search to find optimised parameters.

An advantage of this model-free approach is that it is well-suited for parallelisation. On the other hand, the different racers do not exchange information, thus missing the opportunity to learn about less successful configurations. Consequently, a configurator may lose efficiency by learning the same information more than once, or learning more slowly than it could have with information sharing.

Model-based configurators, on the other hand, sequentially build a model that captures the dependency of the performance of the target algorithm on its configuration. This model is used to predict the performance of configurations on multiple instances and to select promising candidate configurations, which is useful for identifying good configurations more quickly than model-free configurators, because of its ability to learn.

Another important property of configurators is the type of parameters that they support. For example, F-Race focuses on numerical parameters (integer- and real-valued) [174], while paramILS supports numerical and categorical parame-

ters, as well as conditional parameters, whose activations depend on the values assigned to other parameters. An advantage of model-based configurators is that they support many different types of parameters.

This is useful in the context of this work, because in many cases the alternative designs that we have implemented come with very specific sets of parameters. Consequently, the resulting configuration space has many nooks and crannies that are only relevant to explore under specific circumstances. By taking that into account, the search for optimised parameters can often be carried out.

In Chapter 7, we choose SMAC [86] as the configurator for our experiments, because it is one of the best-performing configurators that are freely available.

3.6 Conclusion

In this chapter we described tools for modelling and solving constraint (optimisation) problems, and motivated why we have chosen these specific tools to build on in this work.

Specifically, we described the probabilistic logic programming language DT-ProbLog, which is especially designed to program problems that involve optimal decision making under uncertainty. We described that we need to add functionality for constraints and other types of optimisation than just maximisation in order to use a DT-ProbLog-like language to program the kinds of SCPs studied in this work.

We then proposed to use CP solving as a well-established and powerful search mechanism. We briefly reflected on the two main processes that drive CP solvers: back-tracking search and propagation, and how those relate to the concept of consistency. Finally, we discussed the difference between local and global constraints. Then, we discussed MIP solving as another technique for optimisation. We focused specifically on branch-and-bound techniques and cutting planes techniques, used by MILPs solvers in particular to find integer solutions to mixed-integer linear programs. We also, very briefly, reflected on the existence of quadratic local constraints, and argued why we limit ourselves to linearisable local constraints in this dissertation.

Finally, we motivated why we apply the paradigm of PbO to create a SCP solving pipeline, and gave a brief introduction to this idea, and to AAC, which enables us to take a PbO-based approach. In short, our use of PbO is motivated by the observations that different subtasks in an algorithm can often be completed in different ways (without affecting correctness), and that different design choices that we make there can be better suited for some problems than for others. In

order to fully exploit the potential power of the solver, we therefore implement many of these alternatives, and use AAC to automatically determine optimised configurations for problems from different application domains. We believe that this approach is particularly useful when developing tools for solving hard problems, such as the $\mathcal{NP}$-hard SCPs that we study in this work.