Applied Computational Engines

Lecture Notes

Fachbereich 3 – Mathematics & Computer Science
University of Bremen
Version: April 9, 2018

Prof. Rüdiger Ehlers

Note: These lecture notes are work in progress. Please send comments & suggestions for improvements to:

`ruediger.ehlers@uni-bremen.de`.

Contents

Chapter 1

Introduction

Analyzing computational problems is at the heart of computer science. For example, the computational complexity of a problem tells us lower bounds on the running times and space requirements of the algorithms that we design to solve a problem. Likewise, the theoretical treatment of bijective functions tells us that we cannot design a data compression algorithm that is able to compress *any* possible data stream.

Yet, it can be observed that such theoretical limits are routinely ignored in practice. Modern data compression tools are able to shrink the size of a lot of files that can be found in the field. Such tools make use of the regular structure of many types of files, and are surprisingly robust in doing so. The existence of such tools does not contradict theory, however. In fact, when considering a random uniform distribution over the possible files of some length k, they are unable to compress *most* of the files. Yet, due to the high number of cases in which they *do* work, they are a valuable tool for the practitioner.

Data compression is an example which shows that despite the results from theory, there exist algorithms for some problems that can perform remarkably well on many problem instances found in the field. It may or may not come as a surprise that similar observations were made in other contexts as well. For example, many problems in *operations research* and *formal verification* are essentially *search problems*, and they are often *NP-complete*. Problems from this complexity class are often considered to be intractable in the literature. Yet, they are solved routinely in practice nowadays, which may seem to be a contradiction at first.

The key to this success is the application of modern *computational engines*. These are algorithms that efficiently solve comparably generic problems and allow many other problems to be encoded as input to the computational engine. They make use of the *structure* in their input instances, and typically yield exact results quickly on highly structured input instances.

An example computational engine is a *satisfiability (SAT) solver*, which is commonly applied when formally verifying two combinational circuits to be functionally equivalent. A SAT solver takes a boolean formula as input and checks whether there exists some valuation to its variables that makes the formula satisfied. The behavior of the two circuits can be represented as such a formula, and by a clever encoding, a valuation to the variables then represents an input signal valuation to the circuits on which their outputs differ. If the resulting boolean formula is then found to be unsatisfiable, this constitutes a proof of equivalence of the two combinational circuits from which we built the formula.

Apart from computational engines for problems that are NP-complete (or even harder), there also exist computational problems for some generic problems that are known to be solvable in polynomial time. Prime examples for this are *linear programming* and *maximum flow* algorithms. Many problems from operations research and other fields can be reduced to linear programming or maximum flow problems.

Computational engines do not replace theory. All impossibility and hardness results for some problem of course still apply. Also, computational engines do not replace the approximation algorithms that have been developed for some problems, and they also do not replace modern exact exponential-time algorithms that have been developed for commonly studied problems such as the traveling salesperson problem. Rather, they augment these results by providing platforms that are robustly usable for many practical problems. For an approximation algorithm, whenever we change a part of the problem, the algorithm is typically no longer applicable. Computational engines, on the other hand, can deal with many variations of a problem – we typically only need to change the *encoding* of the problem at hand into their input problem instances for the engine. Thus, they are particularly useful for tackling difficult computational problems for which due to their unique properties, no specialized algorithms exist.

1.1 Scope of the book

In this book, we study a variety of computational engines to tackle computationally difficult practical problems. The engines chosen to be in scope for this book have proven themselves to be well-suited for exploiting the structure of many practical problems. This makes the concepts taught in this book well-applicable to wide classes of challenges dealt with in practice. Apart from computational engines for problems with a high computational complexity, we also consider commonly used such engines for simpler problems.

For every computational engine, we consider the question what structure in a problem the engines exploit and how they deal with the residual difficulty of the problems. We also shed light on the basic working principles of the engines and best practices for applying them. Of course, known weaknesses of the computational engines will also be discussed.

To get the terminology straight, we would like to note that the word "problem" is overloaded in this book:

- It can refer to *practical problems*, but
- it can also refer to *computational problems*.

In the former case, the problems are motivated from various applications such as logistics and formal methods. In the latter case, the problem definition is concise and formally defined and general enough to allow encoding a variety of practical problems into the computational problem. In both cases, a problem describes a relationship between *problem instances* and their corresponding correct output. In a problem instance, all variables have completely fixed values, and problem instances of computational problems are thus inputs to computational engines.

We will cover the following concepts and computational engine classes in this book:

- Satisfiability (SAT) solvers
- Pseudo-boolean programming and constraint satisfaction problems
- Quantified Boolean formula solving
- Linear programming, integer linear programming, and mixed integer linear programming
- Network flow
- Satisfiability modulo theory (SMT) solvers
- Two-player games (of infinite duration)

At the end of the book, the reader will have a broad knowledge of methods to tackle difficult algorithmic problems and will be able to identify suitable methods to deal with such challenges whenever they arise in practice.

1.2 Literature

The following textbooks present the core topics of this book in more detail:

- Biere et al. (2009): **Handbook of Satisfiability**, IOS Press.
  - Chapter 2, CNF Encodings
  - Chapter 3, Complete algorithms
  - Chapter 8, Random Satisfiability
  - Chapter 15, Planning and SAT
- Kleinberg and Tardos (2006): **Algorithm Design**, Pearson Education.
  - Chapter 7, Network Flow

- Kroening and Strichman (2008): **Decision Procedures – An algorithmic point of view**, Springer.
  - Chapter 5.3 – Linear Arithmetic: The Branch and Bound Method
- Knuth (2015). **The art of computer programming. Volume 4B, Pre-fascicle 6A: A (Partial) Draft of Section 7.2.2.2: Satisfiability**. Available online.

Apart from these books, the course topics include results from the following primary literature:

- Bjork (2009): **Successful SAT Encoding Techniques**
- Williams et al. (2003): **Backdoors to Typical Case Complexity**

1.3 Acknowledgements

Thanks go to Maria Kyrarini, Francisco Palau, Heinz Riener, Marcel Walter, Anika Bracht, Diren Senger, and Eike Broda for proof-reading and/or suggesting improvements.

The following public domain images from `www.openclipart.org` have been used in this document:

- "*Chess tile - Queen*" by *portablejim*, 2008

Chapter 2

Satisfiability Solving

Let us consider the problem of solving a *Sudoku*. For some value of n, a Sudoku consists of a grid of cells with width n^2 and height n^2. Some cells of the Sudoku already contain numbers in the range $\{1, \ldots, n^2\}$. The grid in Figure 2.1 is an example for a $n = 3$ Sudoku.

Sudoku is a recreational game. The aim of *solving a Sudoku* is to complete the grid such that:

1. in every row of the Sudoku, each number in $\{1, \ldots, n^2\}$ occurs precisely once,
2. in every column of the Sudoku, each number in $\{1, \ldots, n^2\}$ occurs precisely once, and
3. when partitioning the Sudoku grid into n^2 many $n \times n$ large continuous blocks, in every block, each number in $\{1, \ldots, n^2\}$ occurs precisely once.

Sudoku has become a popular puzzle in the last years and is most commonly known in its $n = 3$ variant. Solving a Sudoku can however also be seen as a computational problem. It comes in two variants: (1) checking if a solution actually exists, and (2) computing the solution. Interestingly, it has been recently shown by Yato and Seta (2002) that checking if a solution to a Sudoku exists is NP-complete.[1] This may be surprising at first, as NP-complete problems are often considered to be intractable. Yet, Sudokus are routinely solved for recreational purposes. This suggests that many Sudokus found in the field have features or some regular structure that can be exploited for solving them.

Let us have a look at some techniques that are used to solve Sudokus in practice despite the NP-completeness of solving them. Perhaps this gives some insights into how – in some sense – the high complexity of computing their solutions is avoided in practice.

The three *rules* of Sudoku that we mentioned above allow us to immediately fill in some parts of a Sudoku's solution. For the Sudoku in Figure 2.1, we can reason as follow:

- In the block D1-F3, the number 1 does not yet occur. There are five free cells available for the 1, namely cells D1, F1, E2, D3, and E3. We cannot put the 1 into cell D1 or cell F1, as there is already a 1 in cell A1. Thus, putting a 1 into D1 or F1 would lead to having two occurrences of the 1 in row 1. Likewise, we cannot put a 1 into cells E2 or cells E3, as these cells are 1-*blocked* by the 1 in cell E7. Thus, cell D3 is the only cell into which we can write a 1, so it has to be there.
- Then, we can put a 1 into cell F4. This follows from the fact that row 4 does not yet have a 1, and F4 is the only place in which we can put a 1. Cells B4 and G4-I4 already have numbers in them. Cells A4 and C4 are 1-blocked as the A4-C6 block already has a 1. Cell E4 must not contain a 1 as E7 already contains a 1. Cell D4 must not contain a 1 as cell D3 already has one. So the only possible cell for a 1 in row 4 that remains is F4.
- Now that F4 is filled with a 1, we can deduce that cell A4 must be filled with a 6. Row 4 does not yet contain a 6 and there are only four free cells. Cells D4 and E4 are 6-blocked by D5, and C4 is 6-blocked by C6. So the only remaining cell in row 4 into which we can put a 6 is cell A4.

In all three cases, the main idea was the following: we pick a number and a line, column, or block in which the picked number is still missing. If we now iterate over the cells within the line, column, or block, and are able to exclude all of them as possible cells for the number except for one cell, then we can safely fill the cell with the number. By following this approach, we cannot make any wrong decisions when solving the Sudoku, as we are only filling the Sudoku with numbers that we know to be safe.

[1] This statement requires n to be arbitrary. When fixing n to 3, the number of different Sudokus becomes finite, which implies that solving such Sudokus can actually be done in *constant time* by looking up the solution to the Sudoku in a list of solutions to *all* Sudokus.

	A	B	C	D	E	F	G	H	I
1	1				7				
2			9	3		8	2		
3			2			9		8	
4		8					4	5	2
5				6					
6			1					3	
7		4			1		3	7	
8			6	7			9		
9		1							

Figure 2.1: A Sudoku

	A	B	C	D	E	F	G	H	I
1	**1**		*8*		**7**				
2			**9**	**3**		**8**	**2**		
3			**2**	*1*		**9**		**8**	
4	*6*	**8**	*7*	*9*	*3*	*1*	**4**	**5**	**2**
5				**6**					
6			**1**					**3**	
7	*9*	**4**			**1**		**3**	**7**	
8			**6**	**7**			**9**		
9	*7*	**1**			*9*				

Figure 2.2: A *partially saturated Sudoku*. The original numbers in the Sudoku are boldfaced, while *inferred values* are italicized.

Figure 2.2 shows how the Sudoku looks like after we used this line of reasoning to also fill the cells C4, E4, D4, C1, A9, A7, and E9.

In the partially solved Sudoku in Figure 2.2, we can now apply a slightly different line of reasoning than in the three cases above in order to fill another cell with a *safe* value. In particular, we consider an empty cell and check how many different values are still possible for this cell.

We focus on cell C7 for this example. In the block in which C7 is located, the numbers 2, 3, 5, and 8 are still missing. Cell C7 is however 3-blocked by cell G7, 8-blocked by cell C1, and 2-blocked by cell C3. Thus, the only number that can be put into C7 is 5. Thus, by excluding all other possibilities for cell C7, we arrived at the conclusion that we have to put 5 there.

If we continue with applying all the techniques for filling in *safe* values that were described so far, we eventually end up with the partially solved Sudoku in Figure 2.3 (after filling in values into the cells A8, B8, C9, F8, C5, G5, H9, H5*, B6, and I1, where * denotes the line of reasoning used for filling cell C7). It can be seen that the Sudoku is not solved yet (i.e., there are still blank cells in the Sudoku).

	A	B	C	D	E	F	G	H	I
1	**1**		*8*		**7**				*9*
2			**9**	**3**		**8**	**2**		
3			**2**	*1*		**9**		**8**	
4	*6*	**8**	*7*	*9*	*3*	*1*	**4**	**5**	**2**
5			*4*	**6**			*1*	*9*	
6		*9*	**1**					**3**	
7	*9*	**4**	*5*		**1**		**3**	**7**	
8	*8*	*2*	**6**	**7**		*3*	**9**		
9	*7*	**1**	*3*		*9*			*2*	

Figure 2.3: A *saturated* Sudoku. The original numbers in the Sudoku are boldfaced, while *inferred values* are italicized.

So far, we have only made *safe* choices, i.e., we only filled cells with numbers that we knew to be correct. This means that if the Sudoku has a solution, any solution has to inherit our choices. The two lines of reasoning that we have used above do not suffice to fully solve the Sudoku, however.

So let us try something different now: we *guess* one number in one cell now, and continue with *saturating* the Sudoku. If at some point, we arrive at a conflict (i.e., we find that there is no way to fill the Sudoku correctly under the guess that we made), then we found the guess to be incorrect. Otherwise, we may be able to completely solve the Sudoku after the guess, or may find that more guesses are necessary. Every time we find a conflict, we know that the combination of guesses that we made is incorrect, i.e., does not admit a valid solution to the Sudoku.

The guess that we will be making now is to put a 2 into cell F1. There are only two cells in the first row in which a 2 may be put (D1 and F1), so we just picked one of them. Let us now saturate the resulting Sudoku. After the guess, we can add values to the cells F7*, D7, I7, G6, I6, I5, E5, F6, F9, F5, A5, B5, A6, E6, D6, D9, D1, E8, I9, I7, G3, and G1. Figure 2.4 shows the resulting grid. We now have a conflict in cell G1: because column G does not have a 7 yet, but cell G1 is the only remaining empty cell, we need to fill a 7 into that cell. However, there is already a 7 in row 1. This proves that our guess to fill cell F1 with a 2 is incorrect (provided that the Sudoku actually has a solution).

The only other cell in row 1 into which we can put the 2 instead is cell D1. Let us thus continue solving from the Sudoku in Figure 2.3 (that only contains *safe* choices) with our added guess. Note that since we know that cell F1 must not contain a 2, cell D1 is the only one into which we can put the 2. Thus, putting it there is actually a safe choice as well.

Figure 2.5 shows the Sudoku after applying our two lines of reasoning for saturating the Sudoku with safe choices (in the order D7*, F7, I7, G6, G9, G3, G1, F9, I3, F1*, B1, H1, B2, B3, B5, A5, A6, E2, E3, A3, A2, D9, E8, D6, E5, E6, I8, H8, I9, I2, H2, I5, I6, F5, and F6).

The Sudoku is completely solved now. Guessing *once* and the application of relatively simple *inference rules* for safe choices was sufficient. This example shows the power of combining guessing with the application of inference rules: despite the very large search space, the problem was solved relatively quickly.

The same approach has proven itself to be useful for a wide variety of practical problems. There is no need to implement them individually for each and every individual problem, though. Rather, many problems for which the combination of inference rules and guessing can be applied can be encoded as simple *satisfiability problems* and then solved by generic *satisfiability solvers*, which we discuss in this chapter.

	A	B	C	D	E	F	G	H	I
1	**1**		*8*	*5*	**7**	2			*9*
2			**9**	**3**		**8**	**2**		
3			**2**	*1*		**9**	*5*	**8**	
4	*6*	**8**	*7*	*9*	*3*	*1*	**4**	**5**	**2**
5	*2*	*3*	*4*	**6**	*8*	*5*	*1*	*9*	*7*
6	*5*	*9*	**1**	*4*	*2*	*7*	*8*	**3**	*6*
7	*9*	**4**	*5*	*2*	**1**	*6*	**3**	**7**	*8*
8	*8*	*2*	**6**	**7**	*5*	*3*	**9**		
9	*7*	**1**	*3*	*8*	*9*	*4*	*6*	*2*	*5*

Figure 2.4: A partially *saturated* Sudoku after applying the guess that cell F1 contains number 2 to the Sudoku from Figure 2.3.

	A	B	C	D	E	F	G	H	I
1	**1**	*3*	*8*	*2*	**7**	*4*	*5*	*6*	*9*
2	*5*	*7*	**9**	**3**	*6*	**8**	**2**	*4*	*1*
3	*4*	*6*	**2**	*1*	*5*	**9**	*7*	**8**	*3*
4	*6*	**8**	*7*	*9*	*3*	*1*	**4**	**5**	**2**
5	*3*	*5*	*4*	**6**	*2*	*7*	*1*	*9*	*8*
6	*2*	*9*	**1**	*4*	*8*	*5*	*6*	**3**	*7*
7	*9*	**4**	*5*	*8*	**1**	*2*	**3**	**7**	*6*
8	*8*	*2*	**6**	**7**	*4*	*3*	**9**	*1*	*5*
9	*7*	**1**	*3*	*5*	*9*	*6*	*8*	*2*	*4*

Figure 2.5: The final completely solved Sudoku.

2.1 The satisfiability problem

Let us now formally define the satisfiability problem.

Definition 1. *Let $\mathcal{V}$ be a set of boolean variables and ψ be a boolean formula over $\mathcal{V}$. The satisfiability problem for ψ is to decide whether there exists an assignment to the variables in V such that substituting the variables in ψ by their values in the assignment results in a boolean formula that is equivalent to* **true***.*

Example 1. Consider the following boolean formula over the variables $\{x, y, z\}$:

$$\psi(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee (y \wedge z)) \wedge (\neg x \vee \neg z)$$

The formula is *satisfiable*. Applying the assignment $f : \{x \mapsto \mathbf{true}, y \mapsto \mathbf{true}, z \mapsto \mathbf{false}\}$ to ψ results in:

$$\begin{aligned} &\psi(\mathbf{true}, \mathbf{true}, \mathbf{false}) \\ = &(\mathbf{true} \vee \mathbf{true} \vee \neg\mathbf{false}) \wedge (\mathbf{true} \vee (\mathbf{true} \wedge \mathbf{false})) \wedge (\neg\mathbf{true} \vee \neg\mathbf{false}) \end{aligned}$$

$= \textbf{true}$

★

Example 2. Consider the following boolean formula over the variables $\{x, y, z\}$:

$$\psi'(x, y, z) = \neg y \wedge (\neg x \vee z) \wedge (x \vee y) \wedge (\neg z \vee y)$$

The formula is *unsatisfiable*. For any assignment to the variables x, y, and z, we have $\psi'(x, y, z) = \textbf{false}$. This can be proven, for example, by constructing a *truth table* for ψ' and checking if every cell in the $\psi'(x, y, z)$ column of the resulting table contains a **false** value. ★

Definition 2. *A boolean formula ψ over a set of variables $\mathcal{V} = \{v_1, \ldots, v_n\}$ is in* conjunctive normal form *(CNF) if it is of the form*

$$\psi(v_1, \ldots, v_n) = \bigwedge_{i \in \{1,\ldots,m\}} c_i,$$

where each c_i is a clause *over $\mathcal{V}$, i.e., a disjunction of* literals*. A literal is a variable or its negation.*

Example 3. Reconsider the boolean formula from Example 2. It is in conjunctive normal form and has the clauses $\neg y$, $\neg x \vee z$, $x \vee y$, and $\neg z \vee y$. The first of these clauses consists of only one literal, namely $\neg y$. The second clause consists of the literals $\neg x$ and z.

The boolean formula from Example 1 is not in conjunctive normal form, as its conjunct $(x \vee (y \wedge z))$ is not a clause. However, we can apply the distributivity law to translate it into CNF form: as $(x \vee (y \wedge z)) = (x \vee y) \wedge (x \vee z)$, we can replace $(x \vee (y \wedge z))$ by the clauses $(x \vee y)$ and $(x \vee z)$ and obtain the following boolean formula that is equivalent to ψ from Example 1:

$$\tilde{\psi}(x, y, z) = (x \vee y \vee \neg z) \wedge (x \vee y) \wedge (x \vee z) \wedge (\neg x \vee \neg z)$$

It can easily be seen that the formula is now in conjunctive normal form. ★

A *SAT solver* is a tool that takes as input a boolean formula and checks whether it is satisfiable or not. In case of a positive answer, a SAT solver also computes a satisfying assignment. The boolean formula used as input is also called a *SAT instance* in the literature. Most solvers require the SAT instance to be in conjunctive normal form.

SAT solvers are said to be able to efficiently solve instances with tens of thousands of variables and millions of clauses. While in its full generality, this statement is incorrect, they are able to solve a surprisingly high number of practical problems in comparably short time. Since the satisfiability problem is *NP-hard* (Cook, 1971; Levin, 1973), we cannot expect SAT solvers to terminate quickly in general. Rather, their usefulness for tackling practical problems has only been validated experimentally.

Before diving into *how* SAT solvers operate, let us look at a few example problems and how they can be encoded into SAT instances.

2.2 Some example problems

2.2.1 Graph coloring

A classical NP-complete problem is *graph coloring*. Here, we are given a graph $\mathcal{G} = (V, E)$, where V is a finite set of *vertices*, and $E \subseteq V \times V$ is a set of *edges*. The problem is to search for a function $z : V \to \{1, \ldots, n\}$ such that for all $(v, v') \in E$, we have $z(v) \neq z(v')$. In this context, $\{1, \ldots, n\}$ are called the *colors*, and n is the number of colors. If for a graph $\mathcal{G}$, we find such a function, we say that $\mathcal{G}$ is *n-colorable*.

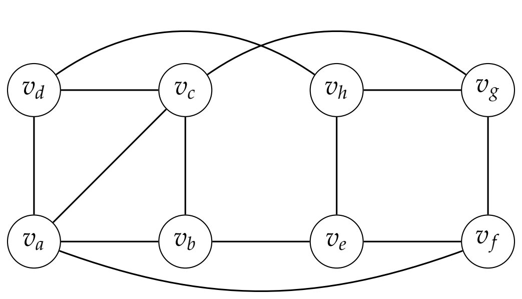


Figure 2.6: Example graph for the graph coloring problem.

Example 4. Figure 2.6 shows an example graph. Let us check if we can manually find a three-coloring to this graph. Without loss of generality, we can color the vertices v_a, v_b, and v_c by 1, 2, and 3, respectively, as they form a *clique* (i.e., each vertex in the set $\{v_a, v_b, v_c\}$ has edges to every other vertex in the set), so they all need to have different colors. Using color 1 for vertex v_a and color 2 for color v_b is without loss of generality: as long as all vertices in $\{v_a, v_b, v_c\}$ get different values, this is a safe decision, as all colors have the same properties. We can then immediately see that v_d has to have a color of 2 (if there is a three-coloring for this graph), as it is connected to vertices that already have colors 1 and 3.

After assigning colors to $\{v_a, v_b, v_c, v_d\}$, there is no simple safe decision that we can make, and we have to guess a values for a vertex in $\{v_e, v_f, v_g, v_h\}$. Let us try to set v_h to 1. From the partial valuation $\{v_a \mapsto 1, v_b \mapsto 2, v_c \mapsto 3, v_d \mapsto 2, v_h \mapsto 1\}$, we can then make the safe decisions $v_g \mapsto 2$ (as v_g is connected to vertices with colors 1 and 3), and then $v_f \mapsto 3$. But now vertex v_e has connections to vertices with colors 1, 2, and 3. So we know that either one of the guesses that we made was incorrect, or there is no three-coloring. So we undo the guess $v_h \mapsto 1$ and try $v_h \mapsto 3$ instead (which is the only remaining possible color for v_h). We can now make a further safe choice, namely $v_e \mapsto 1$. There are then no more safe choices again, so we have to guess a value again. Selecting, for example, $v_g \mapsto 2$ allows us to immediately deduce $v_f \mapsto 3$, which completes the coloring. A quick check reveals that the resulting overall coloring $\{v_a \mapsto 1, v_b \mapsto 2, v_c \mapsto 3, v_d \mapsto 2, v_e \mapsto 1, v_f \mapsto 3, v_g \mapsto 2\}$ is indeed valid. ★

We can encode graph coloring into a satisfiability problem as follows. We start with $\mathcal{V} = \{x_{v,c}\}_{v \in V, c \in \{1,\ldots,n\}}$ as the set of variables. So for every combination of vertex and color, we have one boolean variable, and for a vertex v and a color c, $x_{v,c}$ should be **true** whenever $z(v) = c$. To make sure that every vertex has at least one color assigned, we use the following constraints in our SAT instance:

$$\bigwedge_{v \in V} \bigvee_{c \in \{1,\ldots,n\}} x_{v,c} \tag{2.1}$$

Then, we want to make sure that every vertex has at most one color assigned. To achieve this, we use the following constraints:

$$\bigwedge_{c,c' \in \{1,\ldots,n\}, c' \neq c} \bigwedge_{v \in V} (\neg x_{v,c} \vee \neg x_{v,c'}) \tag{2.2}$$

So we iterate over all pairs of distinct colors and all vertices, and for every vertex we exclude the possibility of having multiple colors assigned. The requirement $c' \neq c$ can also be replaced by $c' > c$. While we iterate over all pairs of color values (c, c') for which c and c' are different, the clauses generated for (c, c') and (c', c) for some $c \neq c'$ are equivalent. By generating only clauses for $c' > c$, we can thus save half of them.

Finally, we add constraints that require the solution to not have the same color assigned to two vertices that are connected by an edge.

$$\bigwedge_{(v,v') \in E} \bigwedge_{c \in \{1,\ldots,n\}} (\neg x_{v,c} \vee \neg x_{v',c}) \tag{2.3}$$

This completes the encoding of the graph coloring problem. Note that all conjuncts in our overall SAT formula are clauses, so the overall SAT instance is in conjunctive normal form. Thus, it can be fed directly to a SAT solver.

The example graph from Figure 2.6 has 8 nodes and 13 edges. For checking if the graph is 3-colorable using the above scheme, we need $8 \cdot 3$ SAT variables. We have 8 clauses of the form in Equation 2.1, and $(2+1) \cdot 8$ clauses of the form in Equation 2.2 when using the slight simplification explained above. For encoding the edges, we need $13 \cdot 3$ clauses. All in all, we thus have 71 clauses. Applying a SAT solver to this instance yields *some* solution to the problem (there are multiple solutions), which can, for instance, be $\{v_a \mapsto 1, v_b \mapsto 2, v_c \mapsto 3, v_d \mapsto 2, v_e \mapsto 1, v_f \mapsto 3, v_g \mapsto 1, v_h \mapsto 3\}$.

2.2.2 Sudoku

The problem of solving a Sudoku that we have used as example earlier in this chapter is in fact also a graph coloring problem. It does however have a fixed graph: all cells have edges that connected them with all other cells in the same row, the same column, and the same $n \times n$ block. This tells us immediately that at least n^2 colors are needed to solve a Sudoku. Also, some colors are already fixed in a Sudoku, which constraints the space of solutions.

For the sake of completeness, let us give a complete encoding of this problem now. Let $n \in \mathbb{N}$ be the width or height of a block in the Sudoku, and $s : \{1, \ldots, n^2\} \times \{1, \ldots, n^2\} \to (\{\bot\} \cup \{1, \ldots, n^2\})$ be a function that describes the fixed values of a given Sudoku, where $\bot$ denotes that a cell has no pre-defined value.

We use the following set of variables:

$$\{v_{x,y,c}\}_{1 \leqslant x \leqslant n^2, 1 \leqslant y \leqslant n^2, 1 \leqslant c \leqslant n^2}$$

So for a standard $n = 3$ Sudoku, we need $9 \cdot 9 \cdot 9$ SAT variables. We start by encoding that every cell should be filled with at least one number:

$$\bigwedge_{1 \leqslant x \leqslant n^2, 1 \leqslant y \leqslant n^2} \bigvee_{1 \leqslant c \leqslant n^2} v_{x,y,c}$$

Furthermore, we encode s into the SAT instance:

$$\bigwedge_{1 \leqslant x \leqslant n^2, 1 \leqslant y \leqslant n^2, s(x,y) \neq \bot} v_{x,y,s(x,y)}$$

Then, we require that for every color/number, in every row, there may only be one cell in which we put the number.

$$\bigwedge_{1 \leqslant y \leqslant n^2, 1 \leqslant c \leqslant n^2} \bigvee_{1 \leqslant x \leqslant n^2, 1 \leqslant x' \leqslant n^2, x \neq x'} (\neg v_{x,y,c} \vee \neg v_{x',y,c})$$

Note that these clauses encode that there is at most one cell in row y that is marked with color/number c. As we have n^2 cells per row and n^2 colors, we however do not need to explicitly encode that every number/color has to appear once in a row – there is simply no other way to satisfy the constraints that we have written so far.

Next, we encode that in every column, a number may only appear at most once:

$$\bigwedge_{1 \leqslant x \leqslant n^2, 1 \leqslant c \leqslant n^2} \bigvee_{1 \leqslant y \leqslant n^2, 1 \leqslant y' \leqslant n^2, y \neq y'} (\neg v_{x,y,c} \vee \neg v_{x,y',c})$$

Finally, we encode that every number may occur at most once in any block.

$$\bigwedge_{0 \leqslant a < n, 0 \leqslant b < n, 1 \leqslant c \leqslant n^2} \bigvee_{1 \leqslant x \leqslant n, 1 \leqslant x' \leqslant n, 1 \leqslant y \leqslant n, 1 \leqslant y' \leqslant n, ((x \neq x') \vee (y \neq y'))} (\neg v_{a \cdot n + x, b \cdot n + y, c} \vee \neg v_{a \cdot n + x', b \cdot n + y', c})$$

Note that this encoding is actually a bit wasteful. Apart from the fact that unlike for graph coloring, we did not remove clauses with duplicate sets of literals (e.g., in a $n = 3$ Sudoku, we both have the clauses $\neg v_{1,2,3} \vee \neg v_{2,2,3}$ and $\neg v_{2,2,3} \vee \neg v_{1,2,3}$), we also have clauses that contain $\neg v_{x,y,c}$ as a literal in cases in which $s(x,y) \neq \bot$. These clauses are either trivially satisfied (in case $s(x,y) \neq c$), or the literal can be removed from the clause (in case we have $s(x,y) = c$). While this makes the encoding unnecessarily large, we will see later that this poses no substantial problem for SAT solvers, though.

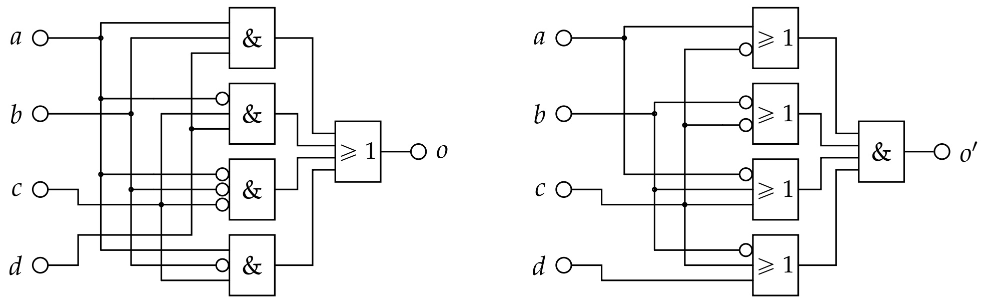


Figure 2.7: Two combinational circuits

2.2.3 Combinational circuit equivalence checking

When designing digital circuits, there are often many ways to express the same boolean functions. Consider, for example, the two circuits in Figure 2.7. The left-hand one of these circuits may be a *reference circuit*, and the right-hand circuit may have been proposed as a more compact alternative to it. The alternative circuit is slightly smaller. Thus, it may be beneficial to replace the reference circuit by the alternative one in order to reduce the manufacturing costs of integrated circuits (ICs) that contain the reference circuit as a component. In order to make sure that this does not introduce a design error, it makes sense to verify that both circuits are equivalent before performing the substitution in a design. Equivalence means that for every input signal valuation to $\{a, b, c, d\}$, the values of o and o' are the same. We can encode the equivalence problem for the two circuits as follows.

We first of all allocate variables $\{a, b, c, d, o, o'\}$. Our SAT instance will require the o and o' variables to encode the outputs of the two circuits for the valuation of the input signals a, b, c, and d. We require that $o_1 \neq o_2$ holds, so that if our SAT instance is satisfiable, this proves that the circuits are *not* equivalent. We use the following clauses for this:

$$(\neg o \vee \neg o') \wedge (o \vee o')$$

Then we need to encode that o and o' actually represent the correct values computed by the two circuits. For this, we add clauses that represent that o computes the OR function from its input ports, and o' computes the AND function from its input ports. However, the values at these input ports are not yet encoded. To fix this, we complete the set of variables in our encoding and allocate one variable for the output of every gate. For the AND gates in the left circuit, we use the variables $\{g_1, g_2, g_3, g_4\}$ (from top to bottom) and for the OR gates in the right circuit, we use $\{g'_1, g'_2, g'_3, g'_4\}$. So overall, we have 14 variables now.

In order to describe that o should encode the disjunction of g_1, g_2, g_3, and g_4, we can now build a truth table for the possible relationships between the values for these variables and encode the truth table as a CNF formula. This yields

$$(\neg g_1 \vee o) \wedge (\neg g_2 \vee o) \wedge (\neg g_3 \vee o) \wedge (\neg g_4 \vee o) \wedge (g_1 \vee g_2 \vee g_3 \vee g_4 \vee \neg o)$$

as new clauses in our SAT instance. After performing such an encoding of all gates in both circuits, we obtain a SAT instance with 40 clauses overall.

Running a SAT solver on it yields the assignment $\{a \mapsto \textbf{true}, b \mapsto \textbf{true}, c \mapsto \textbf{true}, d \mapsto \textbf{true}, g_1 \mapsto \textbf{true}, g_2 \mapsto \textbf{false}, g_3 \mapsto \textbf{false}, g_4 \mapsto \textbf{false}, o \mapsto \textbf{true}, g'_1 \mapsto \textbf{true}, g'_2 \mapsto \textbf{false}, g'_3 \mapsto \textbf{true}, g'_4 \mapsto \textbf{true}, o' \mapsto \textbf{false}\}$. Thus, the circuits are actually not equivalent and differ on the input $\{a \mapsto \textbf{true}, b \mapsto \textbf{true}, c \mapsto \textbf{true}, d \mapsto \textbf{true}\}$.

2.3 SAT solving in practice

Let us now discuss the use of a SAT solver in practice. We use the solve `picosat` for this purpose. Later in Section 2.5.2, we will also use the solver `minisat`. Both of them read SAT instances in the DIMACS format. Files in this format are text files and start with a line of the form

```
p cnf <NumberOfVariables> <NumberOfClauses>
```

This line tells the SAT solver how many variables and clauses are used in the instance. The characters `p cnf` represent that the problem is in CNF format.

The variables are numbered consecutively starting with 1. The header line is followed by the *clause lines*. These consist of space-separated numbers that represent literals in the clause. If a variable is used with negative polarity, the variable number is prefixed with a `-` sign. The clause is terminated with a `0`, which does not represent a literal. As an example, the line `-3 4 -7 0` represent the clause $\neg x_3 \vee x_4 \vee \neg x_7$ (if we assume that the variables in the instance are named x_1 to x_n for some number of variables n).

Finally, comments can be added at any point in a DIMACS input file. They start with a `c` and end at the end of the line.

As a complete example to a DIMACS encoding, we consider the following SAT problem, which we already discussed on page 15:

$$\tilde{\psi}(x,y,z) = (x \vee y \vee \neg z) \wedge (x \vee y) \wedge (x \vee z) \wedge (\neg x \vee \neg z)$$

If x has variable number 1, y has variable number 2, and z has variable number 3 in the SAT instance in DIMACS format, we can represent it as follows:

```
p cnf 3 4
1 2 -3 0
1 2 0
1 3 0
-1 -3 0
```

Running `picosat` on this problem instance yields the following output:

```
s SATISFIABLE
v 1 2 -3 0
```

So the SAT instance is satisfiable and one possible satisfying assignment is $\{x \mapsto \textbf{true}, y \mapsto \textbf{true}, z \mapsto \textbf{false}\}$.

2.3.1 The 4-queens problem

As a more sophisticated example for applying a SAT solver in practice, let us now consider the *4-queens problem*. In that problem, we try to place four queens onto a 4x4 cell checkerboard such that in every row, there is at most one queen, in every column, there is at most one queen, and no queen is diagonally above or below another queen. Figure 2.8 shows an example position of a queen and the positions *blocked* by this queen, i.e., at which no other queen may be placed.

We want to find out if we can place four queens onto the field such that no queen is blocked by another queen. We use one variable per position on the field to indicate whether a queen is present on a field or not, which allows us to use a relatively simple encoding based on 16 variables, where variables 1 to 4 represent the first row of the field, variables 5 to 8 represent the second row of the field, and so on.

Our SAT instance file should start with `p cnf 16 ???` because we have 16 variables, but we do not know the number of clauses yet. Since we expect more than just a few clauses, we may want to automate the process of deriving the clauses. This is normally easy when using a scripting language. For the example in this section, we use the programming language Python.

Since the indices of the positions on the board are two-dimensional, but the variables are successive in a single dimension, it makes sense to start with a *variable encoder* function:

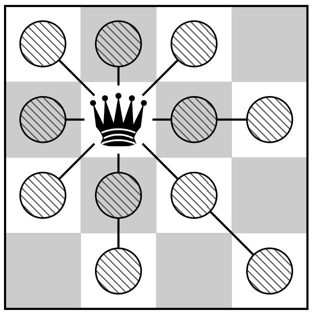


Figure 2.8: A queen on a 4×4 chess board. All positions on the field that can be reached by the queen are marked.

.

```
#!/usr/bin/env python2

def ev(x,y):
    assert x >= 0
    assert x <= 3
    assert y >= 0
    assert y <= 3
    return y*4+x+1
```

The function returns for each pair (x, y) of coordinates on the board what the number of the SAT variable is that represents if a queen is placed there. The `assert` statements are there to make sure that `ev` never returns illegal variable numbers, not even when the values for x and y are illegal. They aid with debugging in case the `ev` function is called with illegal parameters.

The first set of constraints that we want to generate ensure that every row has at most one queen. In clause form, this can be represented as

$$\bigwedge_{y\in\{0,1,2,3\}} \bigwedge_{x_1\in\{0,1,2,3\},x_2\in\{0,1,2,3\},x_2>x_1} (\neg v_{x_1,y} \vee \neg x_{x_2,y}),$$

where a variable $v_{x,y}$ represents whether a queen is present at position (x, y). The clauses state that for every row, if we take two positions in the row such that the second position is right of the left one, we cannot find queens on both of them. The following Python code prints the set of these clauses:

```
for y in xrange(0,4):
    for xA in xrange(0,4):
        for xB in xrange(xA+1,4):
            print -1*ev(xA,y), -1*ev(xB,y), 0
```

Now we want to add the constraints that in every column, there is at most one queen. The Python code to do so is very similar to the last few code lines:

```
for x in xrange(0,4):
    for yA in xrange(0,4):
        for yB in xrange(yA+1,4):
            print -1*ev(x,yA), -1*ev(x,yB), 0
```

Now we need to state that there are at least four queens on the chess board. Since every row may contain at most one, and there are four rows, we can simply encode that every row has at least one queen. So for each row we want to build a disjunction of the variables in that row. This can be done with the following code:

```
for y in xrange(0,4):
    for x in xrange(0,4):
        print ev(x,y),
    print 0
```

```
p cnf 16 80    -10 -12 0    -6 -10 0     9 10 11 12 0    -6 -3 0
-1 -2 0        -11 -12 0    -6 -14 0     13 14 15 16 0   -6 -16 0
-1 -3 0        -13 -14 0    -10 -14 0    -1 -6 0         -10 -15 0
-1 -4 0        -13 -15 0    -3 -7 0      -1 -11 0        -10 -7 0
-2 -3 0        -13 -16 0    -3 -11 0     -1 -16 0        -10 -4 0
-2 -4 0        -14 -15 0    -3 -15 0     -5 -10 0        -14 -11 0
-3 -4 0        -14 -16 0    -7 -11 0     -5 -2 0         -14 -8 0
-5 -6 0        -15 -16 0    -7 -15 0     -5 -15 0        -3 -8 0
-5 -7 0        -1 -5 0      -11 -15 0    -9 -14 0        -7 -12 0
-5 -8 0        -1 -9 0      -4 -8 0      -9 -6 0         -7 -4 0
-6 -7 0        -1 -13 0     -4 -12 0     -9 -3 0         -11 -16 0
-6 -8 0        -5 -9 0      -4 -16 0     -13 -10 0       -11 -8 0
-7 -8 0        -5 -13 0     -8 -12 0     -13 -7 0        -15 -12 0
-9 -10 0       -9 -13 0     -8 -16 0     -13 -4 0
-9 -11 0       -2 -6 0      -12 -16 0    -2 -7 0
-9 -12 0       -2 -10 0     1 2 3 4 0    -2 -12 0
-10 -11 0      -2 -14 0     5 6 7 8 0    -6 -11 0
```

Figure 2.9: SAT instance for the 4-queens problem split into five columns.

Note that the trailing comma in the third line of the code snippet prevents the clause line to end before the `0` is also written. Finally, we want to prevent diagonal placement of queens. The code to achieve this is a bit more complicated:

```
for x in xrange(0,4):
    for y in xrange(0,4):
        for inc in xrange(1,4):
            if x+inc<=3 and y+inc<=3:
                print -1*ev(x,y), -1*ev(x+inc,y+inc), 0
            if x+inc<=3 and y-inc>=0:
                print -1*ev(x,y), -1*ev(x+inc,y-inc), 0
```

The idea of this code is that it iterates over all positions on the board and for each of them computes clauses that make sure that either a queen is not at the position, or no queen is found `inc` positions away in any diagonal direction. By symmetry, the code only needs to look into two diagonal directions instead of all four.

Copying all the code snippets into one Python script and executing it yields 80 clauses in total. We prefix these lines with `p cnf 16 80` and obtain the SAT instance in DIMACS format given in Figure 2.9.

The SAT solver `picosat` returns the following result when applying it to this SAT instance:

```
s SATISFIABLE
v -1 -2 3 -4 5 -6 -7 -8 -9 -10 -11 12 -13 14 -15 -16 0
```

Interpreting the variable assignments gives us the queen distribution shown in Figure 2.10.

Apparently it is a valid solution. But how many are there? To find this out, let us ask the SAT solver to compute another solution. To achieve this, we add a clause that prevents the same solution from being found. We can achieve this by taking the generated assignment, complementing each literal, and adding it as a clause to the solution. In this particular problem, it suffices to only consider the variables that were assigned the value of **true** in the solution. So we add the following line to the SAT instance:

```
-3 -5 -12 -14 0
```

Furthermore, we need to change the number of clauses in the SAT instance file. After having done that, running `picosat` again on the problem instance yields:

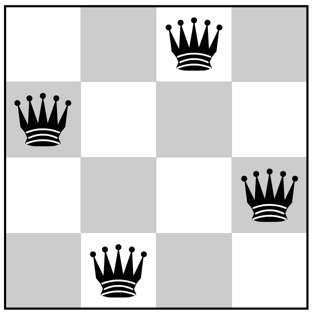


Figure 2.10: A solution to the 4-queens problem
.

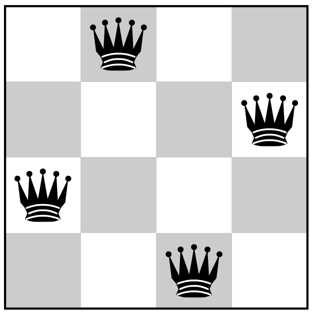


Figure 2.11: Another solution to the 4-queens problem
.

```
s SATISFIABLE
v -1 2 -3 -4 -5 -6 -7 8 9 -10 -11 -12 -13 -14 15 -16 0
```

The corresponding chess board configuration is shown in Figure 2.11. It can be observed that the second solution is the same as the first solution, only that it has been mirrored along either the horizontal or the vertical axis. To check if there is yet another solution, we add another clause ruling out this solution (namely the clause `-2 -8 -9 -15 0`) and call `picosat` again. This time, `picosat` returns:

```
s UNSATISFIABLE
```

So we know now that the 4-queens problem has exactly 2 solutions.

2.4 Solving the SAT problem

After seeing a couple of examples for practical problems that can be encoded as satisfiability problems, we now discuss how such problems can actually be solved efficiently in practice. While there are many approaches to check the safisfiability of a boolean formula (the interested reader is referred to Franco and Martin, 2009, for an overview), most modern SAT solvers build on the same set of basic operating principles, which are augmented by solver-specific heuristics. Knowing these principles is helpful in order to use a SAT solver for tackling practical problems, which motivates the following discussion.

2.4.1 The Davis–Putnam–Logemann–Loveland (DPLL) procedure

Let us consider the following boolean formula over the variables $\mathcal{V} = \{a, b, c, x, y, z\}$:

$$\psi(x, y, z, a, b, c) = \underbrace{x}_{c_1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c_2} \wedge \underbrace{(\neg y \vee z)}_{c_3} \wedge \underbrace{(y \vee a)}_{c_4} \wedge \underbrace{(y \vee b)}_{c_5}$$

Algorithm 1 A first, very simple, SAT solving algorithm

```
1: function SEARCH($\mathcal{V}$,$\psi$,$\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\}$)
2:     if some clause in $\psi$ is falsified by $\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\}$ then
3:         return $\varnothing$
4:     end if
5:     if $n = |\mathcal{V}|$ then
6:         return $\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\}$
7:     end if
8:     $A \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n, v_{n+1} \mapsto$ false$\}$)
9:     if $A \neq \varnothing$ then return $A$
10:    $A' \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n, v_{n+1} \mapsto$ true$\}$)
11:    if $A' \neq \varnothing$ then return $A'$
12:    return $\varnothing$
13: end function
```

$$\wedge \underbrace{(\neg a \vee \neg b)}_{c_6} \wedge \underbrace{(\neg y \vee a \vee b)}_{c_7} \wedge \underbrace{(\neg a \vee b \vee c)}_{c_8} \wedge \underbrace{(a \vee b \vee \neg c)}_{c_9}$$
$$\wedge \underbrace{(a \vee \neg b \vee c)}_{c_{10}} \wedge \underbrace{(a \vee \neg b \vee \neg c)}_{c_{11}}$$

It can be seen that ψ is in conjunctive normal form. In order to reference the clauses of ψ in the following, they have been named by c_1 to c_{11}. We want to determine the satisfiability of this formula. One easy way to do so is to build a truth table of the formula. As it has six variables, the truth table has $2^6 = 64$ entries. While building it is doable for this small example, it is easy to imagine that this approach does not scale to larger examples. For instance, when encoding a Sudoku as in Section 2.2.2, we need $9 \cdot 9 \cdot 9 = 729$ variables. Surely, writing down a truth table with 2^{729} entries is infeasible.

2.4.1.1 Basic version

So let us try something different here, namely let us *gradually guess* a satisfying assignment. We fix an order of the variables. Let this order be x, y, z, a, b, c for this example. We build an assignment to the variables step by step. We call an assignment to some variables a *partial assignment*. After adding one more entry $v_p \mapsto b_p$ to a partial assignment $v_1 \mapsto b_1, \ldots, v_{p-1} \mapsto b_{p-1}$, we check if there is a clause c_i in ψ all of whose variables have already obtained values and that is already known to evaluate to **false**. If we find such a clause, then we know that $v_1 \mapsto b_1, \ldots, v_p \mapsto b_p$ cannot be part of any possible assignment to $\mathcal{V}$ that makes ψ valid. In such a case, we would then try the other possible value for v_p, namely $\neg b_p$. If we still get such a *conflict*, we know that it is already the assignment to $v_1 \mapsto b_1, \ldots, v_{p-1} \mapsto b_{p-1}$ that is incorrect. In such a case, we *backtrack* and try the other possible value for v_{p-1} instead (or backtrack even further if this is already the second value that is being tried for v_{p-1}). Algorithm 1 describes the pseudo-code of a recursive procedure to find a satisfying assignment in this way.

Algorithm 1 has two main differences to naive value table construction:

- It requires much less space: effectively, it requires space linear in the number of variables (in addition to storing ψ)
- It stops the construction of some parts of the value table *early*: as soon as the current partial assignment falsifies a clause, there is no need to consider *completions* of it.

Figure 2.12 depicts how Algorithm 1 operates on our running example: The tree branches according to the choices that the algorithm makes. The valuation at a point in the tree can be read off from the labels of all of the nodes from the root. If a conflict is found (in line 2 of the algorithm), then the tree ends at that point, and the clause that led to the conflict is given. It can be seen in Figure 2.12 that the SAT instance is found to be satisfiable for the assignment $\{x \mapsto$ **true**$, y \mapsto$ **true**$, z \mapsto$ **true**$, a \mapsto$ **true**$, b \mapsto$ **false**$, c \mapsto$ **true**$\}$ after 10 conflicts during the search. This is already a lot better than building a value table with 64 entries.

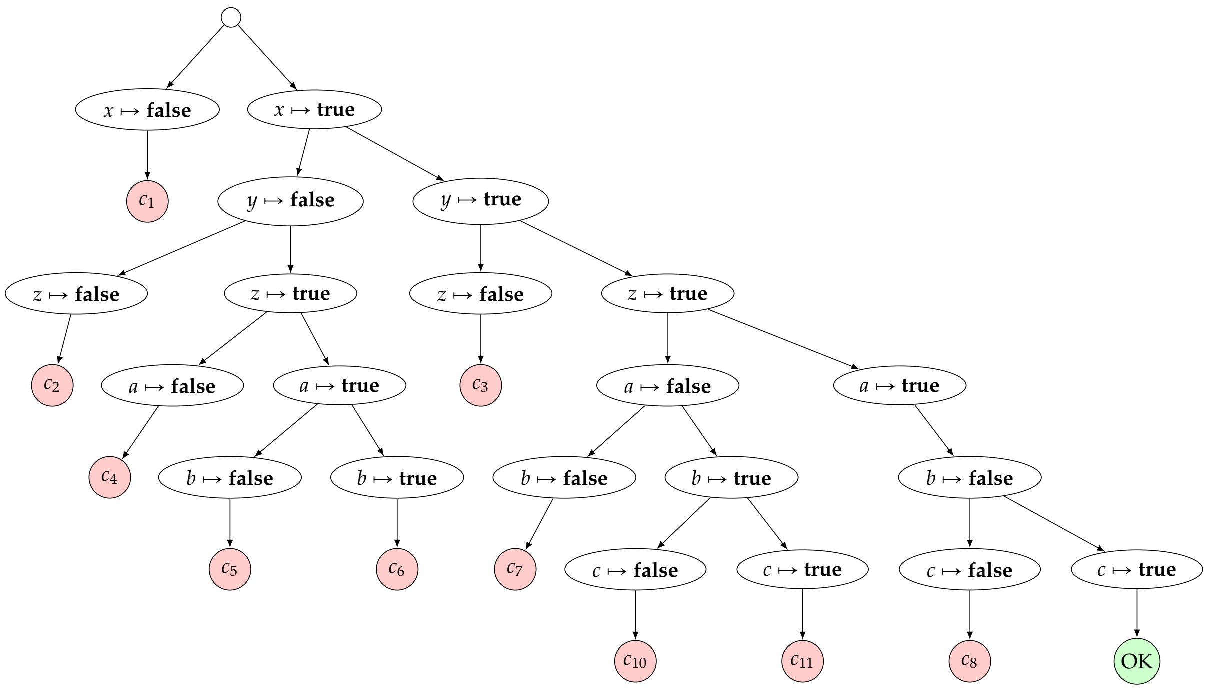


Figure 2.12: A graphical representation of a run of Algorithm 1 on the SAT instance from Section 2.4.1

2.4.1.2 With unit propagation

But we can do even better by applying *unit propagation*. This means that whenever for some clause, the current partial assignment $\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\}$ falsifies all literals except for one, then without loss of generality, we can just assign the variable of the remaining literal the value that makes the literal satisfied. Any solution to the SAT problem that contains $\{v_1 \mapsto b_1, \ldots, v_n \mapsto b_n\}$ as a part of it needs to have this additional assignment in place in order to be valid.

Example 5. Let us reconsider the boolean formula ψ from the beginning of this section. If we start with the partial assignment $p = \emptyset$ and check if there is a clause in ψ that only has one literal left that is not yet falsified by p, we can see that clause c_1 fulfills this criterion (as it only has one literal anyway). Thus, we can immediately extend p to $\{x \mapsto \textbf{true}\}$ without excluding any possible solution to the satisfiability problem of ψ.

Now let us assume that Algorithm 1 subsequently choose $y = \textbf{false}$, so that we have the partial assignment $p' = \{x \mapsto \textbf{true}, y \mapsto \textbf{false}\}$ now. By clause c_4, we can immediately deduce that we also need to have $a = \textbf{true}$, and by clause c_5, we can also deduce that p' needs to be extended by $b \mapsto \textbf{true}$. Thus, we can extend p' to the partial assignment $\{x \mapsto \textbf{true}, y \mapsto \textbf{false}, a \mapsto \textbf{true}, b \mapsto \textbf{true}\}$. However, this assignment contains concrete values for all literals of c_6, and yet it does not satisfy c_6. We can thus immediately see that $\{x \mapsto \textbf{true}, y \mapsto \textbf{false}\}$ cannot be a part of a satisfying assignment for ψ. ★

Figure 2.13 depicts the run of Algorithm 1 after it has been augmented by code to apply unit propagation. The run is based on the same SAT instance as before. Variable valuations that have been obtained by unit propagation are given in rectangle-shape nodes.[2]

It can be seen that the number of *conflicts* in Figure 2.13 is much lower than in the tree in Figure 2.12. Also, the overall number of nodes in Figure 2.13 is a lot lower, which suggests a shorter computation time.

[2]Labelling these nodes by the clauses that led to the propagation of the unit literals is left as an exercise to the reader.

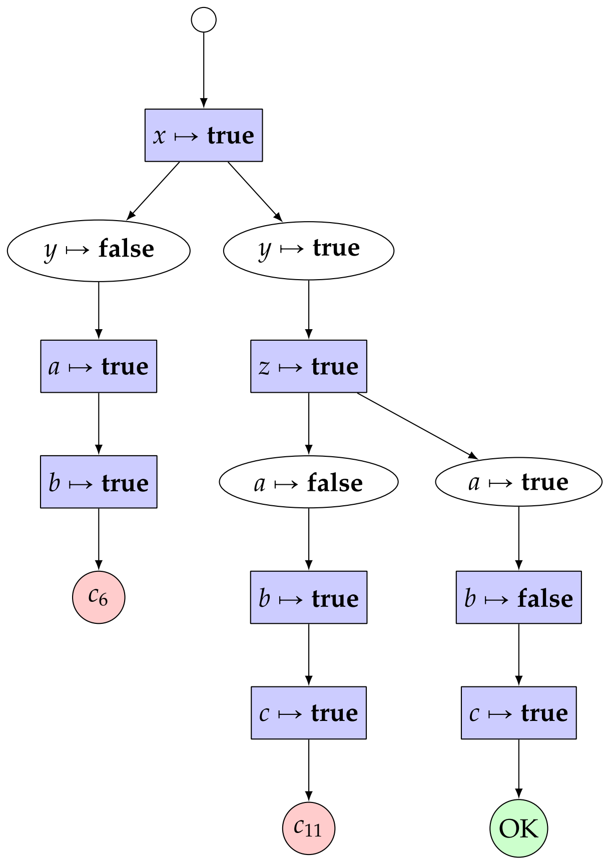


Figure 2.13: A graphical representation of a run of a modification of Algorithm 1 that also uses unit propagation (on the SAT instance from Section 2.4.1)

2.4.1.3 With pure literal elimination

In addition to unit propagation, the DPLL algorithm, as originally described by Davis et al. (1962), featured another rule to speed up the solution of the satisfiability problem. It is nowadays called the *pure literal elimination* rule, and as the unit propagation rule, is based on the idea to assign *safe* values to some variables.

Consider for example the following instance to the satisfiability problem:

$$\psi(a,b,c,d,e) = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg c \vee e) \wedge (\neg d \vee \neg e)$$

Note that we cannot immediately apply the unit propagation rule, as there is no clause in ψ that only has a single literal. However, it can be seen that the variable a only occurs in negated form. Thus, setting $a =$ **false** can only make finding a satisfying assignment to ψ easier and not harder. So by setting $a =$ **false**, one clause is already satisfied, while the other clauses are not affected.

We call $\neg a$ in ψ a *pure literal*, as this literal does not occur in ψ in negated form.

Note that the pure literal rule can sometimes be applied several times. In our example SAT instance ψ, this is the case. After we have set $a =$ **false**, and the first clause effectively vanished, we are left with the modified instance

$$\psi(a,b,c,d,e) = (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg c \vee e) \wedge (\neg d \vee \neg e)$$

Now, $\neg b$ is a pure literal, so we can set $b =$ **false** and remove the clause $(\neg b \vee c)$ from consideration. Afterwards, $\neg c$ is a pure literal in several clauses. We can remove all of them and only $\neg d \vee \neg e$ remains as a clause, so we can apply the rule again for either $\neg d$ or $\neg e$. Since we kept track of all assignments that we made along the way, we ended up with the assignment $\{a \mapsto$ **false**$, b \mapsto$ **false**$, c \mapsto$ **false**$, d \mapsto$ **false**$\}$.

Algorithm 2 The complete DPLL algorithm with *unit propagation* and with the *pure literal rule*. The parameter $\mathcal{V}$ represents the set of variables, ψ is the CNF formula, and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment.

```
1: function Search($\mathcal{V}$,$\psi$,$A$)
2:     for all assignments $v_k = b_k$ implied by $(\psi, A)$ by unit propagation do
3:         $A := A \cup \{v_k \mapsto b_k\}$
4:     end for
5:     for all assignments $v_k = b_k$ implied by $(\psi, A)$ by pure literal elimination do
6:         $A := A \cup \{v_k \mapsto b_k\}$
7:     end for
8:     if some clause in $\psi$ is falsified by $A$ then
9:         return $\emptyset$
10:    end if
11:    if $|A| = |\mathcal{V}|$ then
12:        return $A$
13:    end if
14:    Pick a variable $v$ that is not yet in the domain of $A$
15:    $A' \leftarrow$ Search($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ false$\}$)
16:    if $A' \neq \emptyset$ then return $A'$
17:    $A' \leftarrow$ Search($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ true$\}$)
18:    if $A' \neq \emptyset$ then return $A'$
19:    return $\emptyset$
20: end function
```

Note that we do not have a value for e yet, but there is no clause left to be satisfied. So the value for e can be arbitrary.

The pure literal rule can be incorporated into Algorithm 1 in the same way as the unit propagation rule: at the beginning of the procedure, the current partial assignment is extended whenever there is a rule that allows to extend the partial assignment by safe values.

When using *back-tracking* together with the unit propagation rule and the pure literal elimination rule, we get the *DPLL algorithm*, which is given in its entirety in Algorithm 2.

2.4.2 Conflict-driven clause learning (CDCL)

The DPLL algorithm that we learned about above is actually quite old, as it has been introduced already in 1962. Yet, for many years, the term “intractable” was used in the literature when “NP-complete” was actually meant. This observation can be explained by the fact that when the algorithm was introduced, computers were actually not powerful enough to execute any SAT solving algorithms on SAT instances of interesting size. However, even after personal computers became available, SAT solving was still a niche topic for many years in computer science research. One of the reasons was that the DPLL algorithm still does not offer good scalability on SAT instances of practical interest.

In order to achieve scalability, it had to be augmented by *conflict-driven clause learning* first (Knuth, 2015, p.62, introductory paragraph on CDCL).

Consider the following SAT instance, which is an adaptation of an example by Prestwich (2009):

$$\begin{aligned}\varphi(a,b,c,d,x,y,z) = {} & (a \vee \neg b) \\ & \wedge (\neg a \vee d) \\ & \wedge (\neg a \vee y) \\ & \wedge (b \vee \neg c) \\ & \wedge (d \vee z) \\ & \wedge (\neg d \vee \neg z)\end{aligned}$$

$$\begin{tikzcd}
1{:}\ \neg a \arrow[r] & \neg b \arrow[r] & \neg c \arrow[dddrr, bend left] \arrow[dddrrrrr, bend left] & & & & & \\
2{:}\ \neg d \arrow[rrr] & & & z & & & & \\
& & & & & v \arrow[dr] & & \\
3{:}\ \neg x \arrow[rrrr] \arrow[rrrrrrr, bend right] & & & & \neg u \arrow[r] \arrow[ur] & w \arrow[r] & \neg y \arrow[r] & \textbf{false}
\end{tikzcd}$$

Decisions Implied Literals/Facts

Figure 2.14: A conflict graph.

$$
\begin{aligned}
&\wedge (c \vee x \vee \neg u)\\
&\wedge (u \vee v)\\
&\wedge (u \vee w)\\
&\wedge (\neg v \vee \neg w \vee \neg y)\\
&\wedge (c \vee x \vee y)\\
&\wedge (c \vee \neg x \vee \neg y)\\
&\wedge (c \vee \neg x \vee y)
\end{aligned}
$$

Let us examine the behavior of Algorithm 2 (full DPLL) on this SAT instance, using the variable order $a, b, c, d, x, y, z, u, v, w$ for selecting the next variable to branch on. Initially, neither the unit propagation rule nor the pure literal elimination rule can be applied. Thus, the algorithm branches on the value of a, and initially tries $a = \textbf{false}$. Once $a = \textbf{false}$ has been set, the unit propagation rule extends the partial assignment $\{a \mapsto \textbf{false}\}$ to $\{a \mapsto \textbf{false}, b \mapsto \textbf{false}\}$ and subsequently to $\{a \mapsto \textbf{false}, b \mapsto \textbf{false}, c \mapsto \textbf{false}\}$. Now let us assume that the algorithm chooses $d \mapsto \textbf{false}$. The partial assignment extends to $\{a \mapsto \textbf{false}, b \mapsto \textbf{false}, c \mapsto \textbf{false}, d \mapsto \textbf{false}\}$, and by unit propagation extends to $\{a \mapsto \textbf{false}, b \mapsto \textbf{false}, c \mapsto \textbf{false}, d \mapsto \textbf{false}, z \mapsto \textbf{true}\}$.

At this point, the algorithm branches once more and selects $x \mapsto \textbf{false}$. Unit propagation leads to $y \mapsto \textbf{true}$ and $u \mapsto \textbf{false}$. Further unit propagation lets us deduce $u \mapsto \textbf{false}$, $v \mapsto \textbf{true}$, $w \mapsto \textbf{true}$, and finally $y \mapsto \textbf{false}$. Since unit propagation also yields $y \mapsto \textbf{true}$, we have reached a conflict. The DPLL algorithm would now *backtrack*, choosing **true** as the next value for x (which leads to a conflict again), and then backtrack to choosing $d \mapsto \textbf{true}$. But let us have a look at the setting at the time of the first conflict for the time being.

When the conflict occurs, we find out that the partial assignment $A = \{a \mapsto \textbf{false}, b \mapsto \textbf{false}, c \mapsto \textbf{false}, d \mapsto \textbf{false}, z \mapsto \textbf{true}, x \mapsto \textbf{false}, u \mapsto \textbf{false}, v \mapsto \textbf{true}, w \mapsto \textbf{true}, y \mapsto \textbf{true}\}$ cannot be part of a satisfying assignment to ψ (where we assume that the conflict is detected before $y \mapsto \textbf{false}$ is deduced by the unit propagation rule as well). Some parts of the assignment are implied by unit propagation, whereas others represent decisions. We can arrange both of them in an *conflict graph*, where we distinguish between implied literals and decisions made by the algorithm. For assignment A, Figure 2.14 shows this graph. The left part of the graph shows the *trail*[3] of the DPLL algorithm, i.e., the variable valuations due to *decisions* by the algorithm. The rest of the graph shows implied variable valuations. The incoming edges to a literal show the other literals that were necessary in order to deduce it as an implied literal. For example, the literal $\neg y$ in the graph has edges from $\neg c$ and $\neg x$, as the literal $\neg u$ was deduced by unit propagation from $\neg c$, $\neg x$, and the clause $c \vee x \vee \neg u$.

The conflict graph is a useful tool as it shows us the cause of a conflict. For example, we can immediately see that the decision to set d to **false** did not influence the conflict. Even more important, we can read off some clauses that are *implied* by the SAT instance. For finding such a clause, we make a *cut* through

[3]TODO: Double-check this term. Minisat uses it differently.

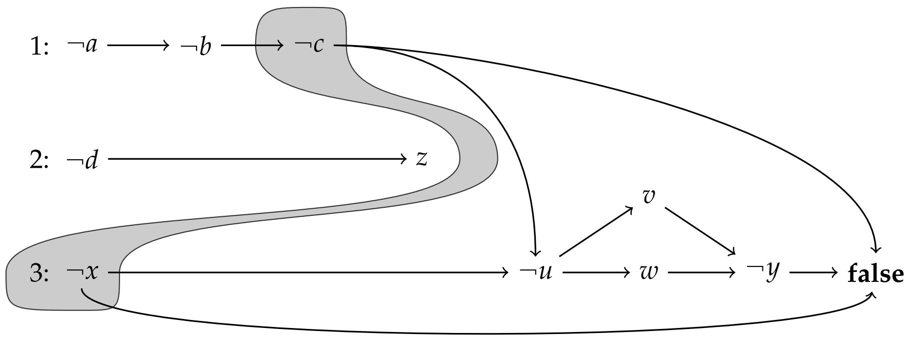


Figure 2.15: An cut though an conflict graph.

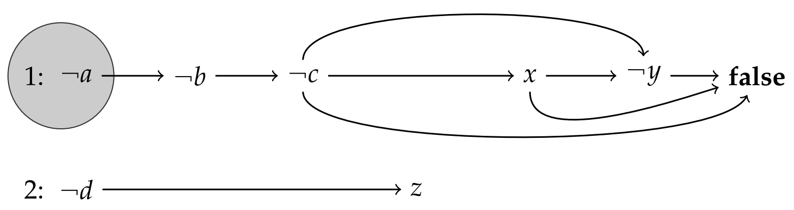


Figure 2.16: Another cut though a conflict graph.

the conflict graph that consists of some literals that separate the **false** fact from all decisions and such that no edge of the graph crosses the cut. Figure 2.15 shows an example of such a cut for the conflict graph from Figure 2.14.

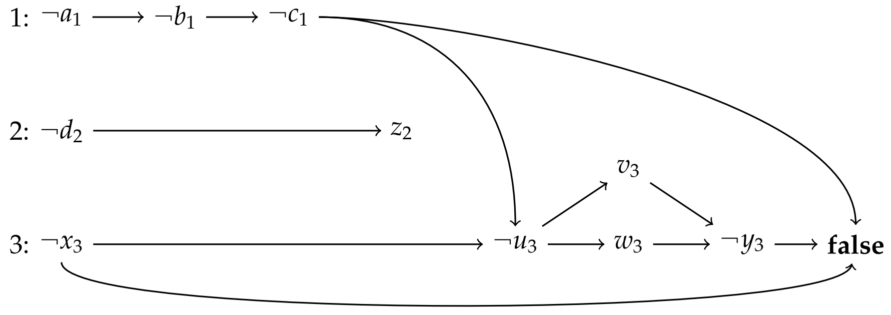
A cut in an conflict graph represents a set of variable values that together imply a conflict. Thus, if we have a set of literals $\{l_1, \ldots, l_n\}$ in a cut, we know that every solution to the satisfiability problem also satisfies the clause $\neg l_1 \vee \ldots \vee \neg l_n$, as this clause summarizes the way in which the conflict was derived. We can thus safely add such a *conflict clause* to a SAT instance without changing the set of its satisfying assignments. The benefit of adding a conflict clause is that it allows the SAT solver to (potentially) make better use of unit propagation in the future.

Let us continue with our running example in order to obtain a demonstration of this effect. After finding the cut from Figure 2.15, a DPLL-style algorithm augmented with clause learning would add the clause $c \vee x$ to the SAT instance ψ and undo the decision $x = \textbf{false}$ in order to try $x = \textbf{true}$. Setting x to **true** would however not be an active decision of the DPLL algorithm, but rather be the effect of performing unit propagation after c has been set to **false**, using the newly added clause $c \vee x$. When continuing with unit propagation, we eventually obtain another conflict, whose conflict graph is given is Figure 2.16 together with a cut.

This time, the cut decpited in the graph is very simple. It consists of a single literal, namely $\neg a$. This allows the algorithm to *learn* the clause a (which happens to be a *unit clause*).

Most conflict graphs allow many cuts. For example, the conflict graph in Figure 2.14 has a lot of cuts: there are 7 literals in the graph, and all subsets of literals are valid cuts that have $\neg x$ in them and at least one of $\neg a$, $\neg b$, and $\neg c$ are contained in the subset as well. This leaves us with $(1024 - 512) \cdot \frac{7}{8} = 448$ possible cuts. Even if we only consider the set of *Pareto-optimal cuts*, which do not contain any literal that can be removed from the cut without making it a subset that is not a cut any more, we still have 3 possible cuts, namely $\{\neg a, \neg x\}$, $\{\neg b, \neg x\}$, and $\{\neg c, \neg x\}$, and it is not clear which one is the best in order to speed up SAT solving in the best possible way.

SAT solvers apply a heuristic to choose the cut that is transformed into a learned clause. Most of them choose cuts that lead to so-called *asserting clauses*. To define what this means, let us have a look at the conflict graph from Figure 2.14 again, but this time label the graph by the *decision level* of each literal, i.e., the number of decisions that had to be made before the literal was deduced (or chosen). The augmented graph is given in Figure 2.17.



$$\begin{tikzcd}
1: & \neg a_1 \arrow[r] & \neg b_1 \arrow[r] & \neg c_1 \arrow[dddrr, bend left] \arrow[dddrrrrr, bend left] & & & & & \\
2: & \neg d_2 \arrow[rrr] & & & z_2 & & & & \\
 & & & & & & v_3 \arrow[dr] & & \\
3: & \neg x_3 \arrow[rrrr] \arrow[rrrrrrr, bend right] & & & & \neg u_3 \arrow[r] \arrow[ur] & w_3 \arrow[r] & \neg y_3 \arrow[r] & \textbf{false}
\end{tikzcd}$$

Figure 2.17: A conflict graph labelled by the decision levels for each literal.

An asserting clause is a clause that has exactly one literal from the last decision level. For example, for the conflict graph in Figure 2.17, $y \vee b$ is such a clause, but $y \vee x \vee c$ is none. An asserting clause can be obtained by the following procedure:

- At every stage, maintain a cut C, starting with $C = \{\textbf{false}\}$ (while ignoring that although **false** is a deduced fact in a conflict graph, it is actually not a literal).
- While C either contains **false** or an element with the highest possible decision level in the graph that is not a decision (but an implied literal), replace the element by the literals in the conflict graph that have an edge to the element.

In the graph in Figure 2.17, this algorithm starts with $C = \{\textbf{false}\}$, followed by $C = \{\neg y, \neg c, \neg x\}$, $C = \{v, w, \neg c, \neg x\}$, $C = \{\neg u, w, \neg c, \neg x\}$, $C = \{\neg u, \neg c, \neg x\}$, and finally $C = \{\neg x, \neg c\}$. So we obtain the learned clause $x \vee c$. While for many conflict graphs, there are multiple asserting clauses, this procedure chooses one particular one. It also the one that is actually implemented in the `minisat` solver.

Augmenting clauses help the solver with performing backtracking in an efficient way, which is why they are normally preferred outcomes of conflict graph analysis. In particular, after the last decision has been undone, whenever an asserting clause has just been learned, it guarantees that it can deduce the negation of the literal that was the last decision before the conflict occurred from the earlier decisions. This simplifies and speeds up the search procedure.

Whenever a new clause has been learned, many SAT solvers also not simply backtrack to the previous decision, but rather undo all decisions since the decision on the second-highest decision level that has literals in the learned clauses. For the example in Figure 2.17, this means not only undoing the $\neg x$ decision, but also the $\neg d$ decision. Part of the motivation behind this process is that unit resolution should be performed as early as possible in the search in order to make as few decisions as possible.

Again, let us study this approach by means of an example. If a DPLL-style algorithm augmented with clause learning reaches the conflict in Figure 2.14 on the SAT instance ψ, it learns the clause $x \vee c$, and undos all decision since the first decision level. Then, it applies unit propagation again and reaches again a conflict. The conflict graph at this point is the same as in Figure 2.16, except that the $\neg d$ and z literals are missing. The search algorithm now learns the augmenting clause a, which is unit. The algorithm then backtracks to the first decision level, and applies unit resolution to the unit clause a. Afterwards, among others, the clause $(\neg a \vee d)$ leads to more unit propagation, and the algorithm also derives the literal d. So unit propagation yielded a value for a variable now that has previously been set by a decision. So by back-tracking to the second-highest decision level, a DPLL-style solver can sometimes reduce the number of decisions needed during search, which motivates this approach.

This example shows the power of clause learning: it allows to deduce new corollaries of a SAT instance step by step, and these corollaries allow the solver to sometimes *jump* over some *decision levels* when backtracking.

2.4.3 Completing the picture: augmenting techniques for SAT solving

We have discussed the core concepts of modern SAT solvers in the previous sections, namely *unit propagation*, *pure literal elimination*, *branching and backtracking*, and *conflict-driven clause learning*. In order to obtain an efficient SAT solver, these concepts are typically augmented by a couple of supporting techniques, for which it suffices for the SAT practitioner to know that they exist in order to apply SAT solvers successfully, without knowing their details.

Fast implementations: Most SAT solvers are written in programming languages that do not cause a significant runtime overhead, such as C (e.g., `picosat`), or C++ (e.g., `minisat`). This choice is motivated by the fact that the set of core concepts in SAT solving is relatively small, but efficiency is of uttermost importance. While SAT solvers have become a lot more efficient in the last 10-20 years, there are many application domains that deal with large problems that are still difficult to solve, even for modern solvers. For these, a few percent of reduction in the computation time of a SAT solver is helpful, so solvers are written in low-overhead programming languages.

It is also common for SAT solvers to apply some low-level technical tricks to increase efficiency, such as custom memory management functions (e.g., in `picosat`, Biere, 2008), and the injection of a variable-sized array into the memory block of an object in C++ (`minisat`).

Watched literals: A SAT solver often has to perform a lot of unit propagation during runtime, and it has to undo unit propagation after a conflict. This motivates storing the clauses in a way that allows unit propagation and undoing it to be performed efficiently.

Many SAT solvers use *lazy data structures* for storing clause sets efficiently. These allow the SAT solver to only look at a subset of the clauses in every unit propagation step. A common way in which this idea is applied is to let every clause only have *two* literals that are being *watched*. Only when the variable of one of these literals has been assigned a value, the SAT solver updates the clause and checks if it is already satisfied by the current partial assignment. If this is not the case, then the solver either (1) selects a new second literal in the clause to be watched, or (2) finds that all literals except for one have already been falsified. Then, the clause gives rise to an additional step of unit propagation.

Lazy data structures for SAT solving are a non-trivial topic as it has to be ensured that the SAT solver's *clause database* and its lists of watched literals are still in a clean state after backtracking. The interested reader is referred to Marques-Silva et al. (2009, Chapter 4.5.1) for more details.

Efficiency-driven variable selection heuristics: In order to select variables to branch on in the DPLL algorithm, SAT solvers typically keep track of some statistics that indicate which variables are most promising for branching.

By for example keeping track of in how many clauses a variable appears complemented and non-complemented, the SAT solver can find the variables whose value is most central for finding a satisfying assignment and branch on them first. Also, keeping track of variable usage statistics allows the solver to apply the pure literal rule without actually iterating over the clauses: if for some literal, there is no clause that is not already satisfied and that contains the literal, then deciding a safe value for the variable of the literal is easy.

However, spending much time on analyzing which is the best variable to branch on in DPLL-style solving (and which *polarity* for the variable to select first) can slow the SAT solver down substantially. Thus, current heuristics are highly optimized for speed and try to avoid gathering data that takes a lot of time to compute. Especially in combination with the lazy data structures, this is of relevance.

Some SAT solvers (such as `picosat`) also allow to specify a *default phase* for variables whose values are set by branching. Supplying this information to the SAT solver can help to speed up solving for applications for which this was observed to be beneficial, or can help to guide the SAT solver towards finding certain more desirable solutions to the SAT problem, whenever this makes sense in an application.

Restarts and deletion of learned clauses Many SAT solvers *restart* after a number of conflicts, meaning that they scrap the current partial assignment and start afresh with the search process.

The intuition behind doing so is that SAT solvers can take much longer to solve an instance if wrong decisions have been performed early in the search. Restarts are executed to get the solver out of a part of the search space that does not contain a solution to the SAT problem (Biere et al., 2009, Chapter 3.6.4.5).

When a restart occurs, the solver keeps learned clauses from before the restart, and also keeps the information gathered for branching variable selection, which helps with selecting more suitable variables to branch on after the restart.

Most modern SAT solvers also delete learned clauses after a while if they do not turn out to be beneficial for speeding up the search process. To find out to which clauses this applies, they keep track of how often a learned clause triggered unit propagation.

2.5 Strengths and limits of SAT solving

Almost nothing is known about why the heuristics in modern solvers work as well as they do, or why they fail when they do. Most of the techniques that have turned out to be important were originally introduced for the wrong reasons!

Donald Knuth in an interview (Knuth et al., 2014)

While SAT solvers have been introduced as a computational engine that solves many problems of practical interest and we have discussed a few reasons for *why* SAT solvers are able to deal with many problem instances, the insights gained on these topics do not actually tell us how it comes that SAT solving scales so well in practice and where the limits of (CDCL-style) SAT solving lie. It is often claimed that SAT solvers can nowadays deal with six-digit numbers of variables and millions of clauses easily. However, it is not clear why SAT solvers can cope with such large instances to an NP-complete problem.

Unfortunately, theory lags behind practice on this topic, and there are no satisfactory answer to what makes the solution of a SAT instances easy. However, we can discuss some results that provide us with some insights into the limits and strengths of SAT solving, which we want to do in this section.

2.5.1 Random 3SAT

Literature: *Achlioptas, 2009*

The NP-completeness of the SAT problem in combination with the observation that most problem instances in the field seem to be "suffiently easy" has led to some interest in the *average case complexity* of the SAT problem, i.e., how fast the problem can be solved on average over some distribution of the input instances.

Let the expression $F_k(n, m)$ denote a distribution over SAT instances with n variables and m clauses, where each clause has k literals, and the literals are chosen uniformly at random, without replacement.

In this model, there are $2^k\binom{n}{k}$ possible different clauses, and the values k, n, and m have to be fixed in order to obtain a concrete distribution. While the study of *random satisfiability* has been carried out by many different researchers (Achlioptas, 2009), a commonly used concretization is to set $k = 3$. This is the minimum value of k that makes the problem of deciding the satisfiability of an instance from the distribution NP-complete, as there exists a polynomial-time algorithm for 2-SAT, i.e., the satisfiability problem over SAT instances for which each clause has at most two literals. It is also customary to express m by means of a *density value* r such that $m = r \cdot n$.

Let us have a look how a SAT solver behaves for such a distribution $F_3(n, rn)$ in an experiment in which we fix $n = 250$ and in which we let r slide between values from $[3, 6]$ in steps of $\frac{3}{100}$. So we consider 101

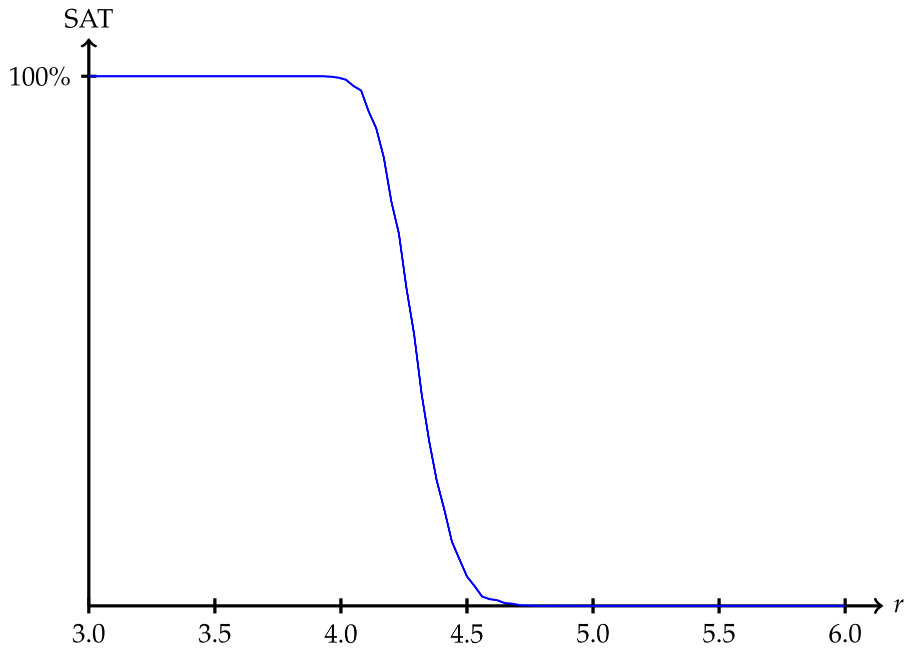


Figure 2.18: Percentages of 3SAT instances that were found to be satisfiable.

different points for r, and we generate 2270 random SAT instances from the distribution $F_3(250, r \cdot 250)$ for each of these values of r.

Figure 2.18 shows how many of these instances were found to be satisfiable. It can be seen that for low values of r, all of them were satisfiable, whereas for high values, almost none of them were. This suggests that the probability of the satisfiability of a SAT instance in the distributions depends strongly on the value of r, with the threshold between the mostly satisfiable and mostly unsatisfiable instances being somewhere close to $r = 4.3$.

Figure 2.19 shows computation times of the solver `picosat` on the generated instances. The two lines in the graph show the mean computation times on a AMD Opteron(tm) Processor 2.8GHz machine for satisfiable and unsatisfiable instances separately (where 1490 minutes of overall wall clock time was needed for the experiment). It can be seen that close to the threshold, the SAT instances are quite difficult to solve, but far from the threshold, they are quite easy to solve.

Unsatisfiable instances near the threshold appear to be more difficult to solve than satisfiable ones near the threshold. This is not a surprise: whenever the SAT solver finds a satisfying assignment, it can immediately stop with the search. To prove unsatisfiability, the SAT solver has to perform a complete sweep of the search space, which takes longer. Only when unsatisfiability becomes very easy to show (for high values of r), SAT solver running times become very short again.

The study of random 3SAT instances indicates that SAT solving seems to work well on instances whose satisfiability or unsatisfiability can be shown easily. Only for values of r where the satisfiability of an instance is relatively uncertain, the solver needs some more computation time. This observation was already made by Mitchell et al. (1992) and suggests that a similar effect may hold for SAT instances from practice. However, it should be noted that there is no reason to believe that the distribution of SAT instances that we find in the field is in any sense similar to $F_3(n, rn)$. Thus, the study or random SAT instances can only provide us with some ideas for why SAT solving works, rather than giving us the actual answer. Some techniques that are used in today's SAT solvers have however been motivated by effects that were observed during the solution of random SAT instances and have subsequently been proven themselves to be useful for SAT solving on industrial instances as well.

Note that we did not perform a full statistical analysis in this section: we neither stated conjectures upfront, nor computed confidence values for our results. Also, in order to obtain the random SAT instances, the default pseudo-random number generator of the `Python` interpreter was used, which

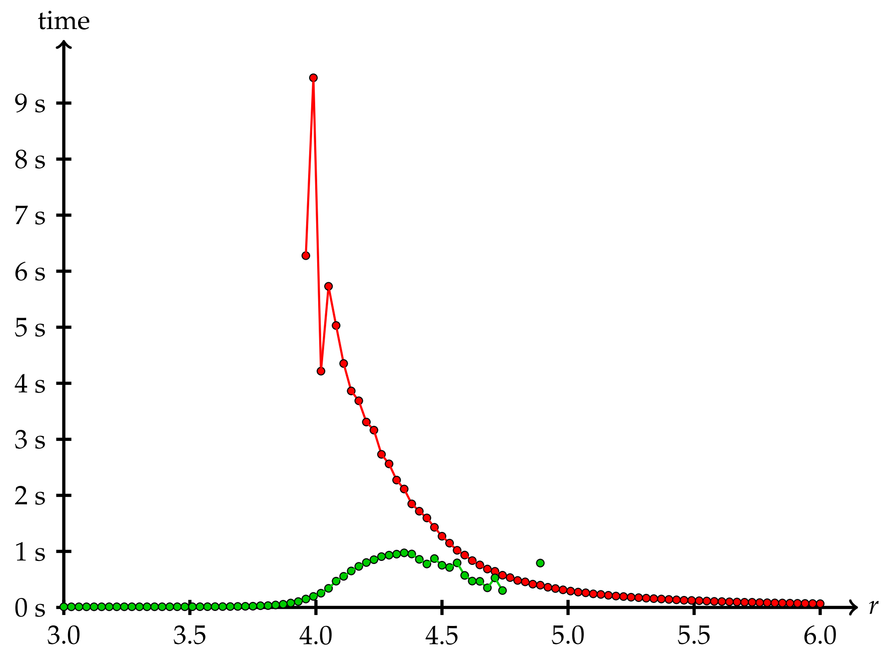


Figure 2.19: Computation times of `picosat` on random 3SAT instances.

may or may not skew the results into some direction. We shall also note that for all values $r \in [3,6]$, there exist satisfiable and unsatisfiable instances that $F_3(n, rn)$ generates with strictly positive probability (for $n \geqslant 3$). For example, we only need eight distinct clauses over some selection of three variables such that all instances that contain these clauses are unsatisfiable. Thus, unsatisfiable instances have a positive probability of being selected for all $n \geqslant 3$ and $r \geqslant 3$. Likewise, there is always a strictly positive probability that we get a SAT instance in which no variable ever appears in both positive and negative polarities. Such a SAT instance is trivially satisfiable. In both cases, due to the small probability of such instances appearing during experimentation, we have not observed them.

To conclude this section, it should be mentioned that some recent work in the study of random SAT instances aims at making progress towards proving the following conjecture.

Conjecture 1 (Satisfiability Threshold Conjecture)**.** *For all $k \in \mathbb{N}$ with $k \geqslant 3$, there exists some value $t_k \in \mathbb{R}_{>0}$ such that for all $r \in \mathbb{R}_{>0}$, we have*

$$\lim_{n\to\infty} \text{Probability}(\textit{SAT instance in } F_k(n, rn) \textit{ is satisfiable}) = \begin{cases} 1 & \textit{if } r < t_k \\ 0 & \textit{if } r > t_k \end{cases}$$

So the conjecture states that the curve in Figure 2.18 in fact has a sharp saddle when we let n range to infinity. For large values of k, the conjecture has already been proved (Ding et al., 2015). The conjecture is mainly of interest for theoretical reasons and because of its connection to statistical physics. The interested reader is referred to Achlioptas (2009) for more details.

2.5.2 The pidgeon-hole principle

So far, we only applied DPLL/CDCL-style SAT solvers to problems for which they performed quite well. Let us now consider a problem that is very difficult to deal with for modern SAT solvers, as this shows the limits of current solvers. For simplicity, we will use the DIMACS encoding of SAT instances in this section.

Assume that we have n pidgeons that we want to place into m pidgeon holes such that in no pidgeon hole, there is more than one pidgeon. It is relatively easy to see that this can only work if we have

$m \geqslant n$. Yet, we can ask a SAT solver to check for some fixed values of n and m with $m < n$ if there exists an assignment of n pidgeons to m pidgeon holes such that no two pidgeons are assigned to the same hole.

For this, we can use $n \cdot m$ variables in a SAT instance, so that we have one variable for every combination of pidgeon and pidgeon hole. We then add clauses for

1. requiring that every pidgeon is assigned to a hole, and
2. requiring that for no two pidgeons and no hole, the two pidgeons are assigned to this hole.

The number of clauses is $n + n \cdot (n-1) \cdot m$, with a mean number of $\frac{nm+2n(n-1)m}{n+n(n-1)m}$ literals per clause.

Let us consider the run of `minisat` 2.2.0 (Eén and Sörensson, 2010) on a SAT instance for $n = 11$ and $m = n - 1$, where the mean number of literals per clause is 2.09. Obviously, the SAT instance should be unsatisfiable. Figure 2.20 shows the output of `minisat` on the instance when running it on a Intel Core i5-4200U 1.60GHz machine with an x64 version of Linux.

It can first of all be seen that it takes `minisat` quite long to determine the unsatisfiablity of the instance. In total, approximately 16.8 million conflicts occurred during the search.

But also the statistics about learned clauses are interesting, which `minisat` gives in columns five to seven of the statistics table that it prints during execution.

To interpret them, it needs to be known that `minisat` enforces a limit on the number of learned clauses. Whenever some clause is added and the limit is exceeded, the solver removes some learned clauses from its database and increases the limit. During the solution process, the current limit can be seen in column five of the table. In column six, we see the number of learned clauses at the time of printing the line to the table. The seventh column finally reports the mean number of literals per learned clause. It can be seen that it is quite high for the pidgeon hole principle problem instance, namely between 31 and 38 literals. Such learned clauses only prune the state space very little during search, which is part of the reason why it takes `minisat` takes so long to deduce the unsatisfiability of the SAT instance.

But how comes that the learned clauses are so long? To see this, let us look at the very first conflict in the search process. Both of the types of clauses stated above need to be involved in a conflict, as individually, unit propagation would ensure that no conflict can arise. The clauses that require that every pidgeon is assigned to a hole have length m whereas the other clauses only have length 2. Learning an asserting clause, as done in modern SAT solvers now means that the cut in the conflict graph to determine a learnt lause is performed left of all unit propagation on the highest decision level. Since a clause of length m must be involved in the cut, all such clauses range over different variables, and all other unit propagation is performed on two-literal clauses, this means that the cut must range over at least m variables.

But actually, the learned clauses in the SAT solver run reported on in Figure 2.20 are even longer, which makes one wonder how this comes. Let us consider the pidgeon hole principle SAT instance for $n = 5$ and $m = 4$ to study this issue more closely. On this instance, `minisat` can almost immediately deduce the unsatisfiability, and needs only 32 conflicts for this. Unfortunately, `minisat` does not print any information about learned clauses in this case.

So let us use a modified version of `minisat` 2.2.0 (in the core solver version) that prints the learned clauses during the SAT solving process. Figure 2.21 shows the code that can be added to the end of `minisat`'s `Solver::attachClause(CRef cr)` procedure that adds clauses to a SAT instance in order to enable learned clause printing.

Running our modified `minisat` version on a SAT instance for the $n = 5$ and $m = 4$ case yields `-11 6 5 1 2` as the first learned clause. In the encoding, the variables are grouped by pidgeons. So for example, the variables 1, 2, 3, and 4 state whether the first pidgeon is put into holes 1, 2, 3, or 4. The learned clause now states that if pidgeon 1 is not put into holes 1 or 2 (literals `1` and `2`), and pidgeon 2 is not put into these holes either (literals `5` and `6`), then pidgeon 3 must not be put into hole 3. To see this why this clause has been learned, consider the case that pidgeon 3 is indeed put into hole three while the solver already chose `-1 -2 -5 -6` as partial variable valuation. In this case, the solver can immediately deduce that pidgeon 1 must be put into hole 4, as holes 1 and 2 are excluded by the partial valuation,

```
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:           110                                         |
|  Number of clauses:             561                                         |
|  Parse time:                   0.00 s                                       |
|  Eliminated clauses:           0.00 Mb                                      |
|  Simplification time:          0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|       100 |      99      550     1980 |      201      100     31 |  0.008 % |
|       250 |      99      550     1980 |      221      250     39 |  0.008 % |
|       475 |      99      550     1980 |      244      205     38 |  0.008 % |
|       812 |      99      550     1980 |      268      246     35 |  0.008 % |
|      1318 |      99      550     1980 |      295      275     39 |  0.008 % |
|      2077 |      99      550     1980 |      324      354     38 |  0.008 % |
|      3216 |      99      550     1980 |      357      209     37 |  0.008 % |
|      4924 |      99      550     1980 |      392      313     38 |  0.008 % |
|      7486 |      99      550     1980 |      432      269     34 |  0.008 % |
|     11330 |      99      550     1980 |      475      312     37 |  0.008 % |
|     17096 |      99      550     1980 |      523      435     34 |  0.008 % |
|     25745 |      99      550     1980 |      575      349     33 |  0.008 % |
|     38719 |      99      550     1980 |      632      603     35 |  0.008 % |
|     58180 |      99      550     1980 |      696      428     34 |  0.008 % |
|     87372 |      99      550     1980 |      765      741     36 |  0.008 % |
|    131161 |      99      550     1980 |      842      710     38 |  0.008 % |
|    196845 |      99      550     1980 |      926      834     31 |  0.008 % |
|    295371 |      99      550     1980 |     1019      718     37 |  0.008 % |
|    443160 |      99      550     1980 |     1121      701     36 |  0.008 % |
|    664843 |      99      550     1980 |     1233      597     36 |  0.008 % |
|    997368 |      99      550     1980 |     1356     1139     34 |  0.008 % |
|   1496156 |      99      550     1980 |     1492      759     34 |  0.008 % |
|   2244338 |      99      550     1980 |     1641     1507     38 |  0.008 % |
|   3366612 |      99      550     1980 |     1805     1590     38 |  0.008 % |
|   5050023 |      97      531     1926 |     1986     1390     32 |  1.827 % |
|   7575139 |      97      531     1926 |     2184     1444     33 |  1.827 % |
|  11362814 |      95      512     1888 |     2403     1383     33 |  3.645 % |
===============================================================================
restarts              : 24574
conflicts             : 16769952       (61251 /sec)
decisions             : 19982879       (0.00 % random) (72986 /sec)
propagations          : 219936114      (803304 /sec)
conflict literals     : 583080092      (9.07 % deleted)
Memory used           : 21.00 MB
CPU time              : 273.79 s

UNSATISFIABLE
```

Figure 2.20: Output of `minisat` 2.2.0 on a SAT instance for the pidgeon hole principle with $n = 11$ and $m = n - 1$.

and hole 3 is now blocked (which leads to the deduction of the literal `-3` by unit propagation using the clause `-3 -11 0`). But now unit propagation also deduces that `-7` and `-8` have to hold, so that pidgeon 2 does not have a hole, and thus the clause `5 6 7 8` is violated.

So `-11 6 5 1 2` is a correct learned clause. However, it is also a pretty long clause: it has five literals although it only considers three pidgeons (out of five). This is rooted in the fact that when the clause was built, the SAT solver already made the decisions `-1 -2 -5 -6`, which lead to no unit propagations at all. This causes relatively large learned clauses, and hence little to no speed-up of the solver by clause learning.

The pidgeon hole principle SAT instances show that the reasoning rules applied by SAT solvers do not always lead to high performance. In fact, regardless of how the solver deletes learned clauses or how variables to branch on are selected, exponential running time cannot be avoided by a CDCL-style solver on a pidgeon hole principle SAT instance if no additional measures are taken. Formally, this can be

```
// Modification for the A.C.E lecture: Print learned clause
if (c.learnt()) {
    std::cerr << "Learnt clause:";
    for (int i=0;i<c.size();i++) {
        if (c[i].x & 1) {
            std::cerr << " " << -1*(c[i].x/2)-1;
        } else {
            std::cerr << " " << (c[i].x/2)+1;
        }
    }
    std::cerr << std::endl;
}
```

Figure 2.21: Code that is added to the end of the `Solver::attachClause(CRef cr)` procedure in `core/Solver.cc` of `minisat` 2.2.0 to let the solver print learned clauses. To make this modification work, the line `#include <iostream>` had to be added to the beginning of the file. Because the solver uses least significant bits to store the polarity of literals instead of the signs of the integers, the literals have to be translated before they are printed.

proven as follows: CDCL-style SAT solvers *simulate* general *Boolean resolution*. This means that from every run of a SAT solver that leads to a satisfiability or unsatisfiability result, a proof of satisfiability or unsatisfiability of the respective SAT instance that only consists so-called resolution steps can be constructed (Pipatsrisawat and Darwiche, 2011). The proof is of size proportional to the number of computation steps performed by the solver. Since it has been shown that all resolution proofs for the pidgeon hole principle (in which there are fewer holes than pidgeons) need size exponential in the number of holes (Urquhart, 1987), this shows that classical CDCL-style SAT solvers cannot solve such instances quickly.

The study of pidgeon hole principle instances is relevant to the practice of SAT solving as many practical problems can include them as sub-problems when the current partial assignment by the solver is not a good one. In such a case, the solver can spend a lot of time on solving an impossible-to-solve subproblem, which is one of the reasons why modern solver use random restarts.

2.5.3 Backdoors

Literature: *Williams et al., 2003*

The SAT instances originating from the pidgeon hole principle show that modern SAT solvers are not always able to solve SAT instances in a reasonable timespan. Our study of random SAT instances shows that from a probabilistic point of view, most SAT instances are easy, any only those at the borderline between satisfiability and unsatisfiability seem to be difficult to solve. But what exactly makes a SAT instance easy or difficult to solve?

There has been some research that tries to answer this question (Williams et al., 2003), and the results that have been obtained so far give a partial answer. A central notion in this context is the one of a *backdoor*.

Definition 3 (Polynomial-time sub-solver). *We say that a function f that maps a Boolean formula φ over a set of variables $\mathcal{V}$ to* {**false**, **true**, *unknown*} *is a* polynomial-time sub-solver *if:*

- *all boolean formulas that f maps to* **true** *are satisfiable,*
- *all boolean formulas that f maps to* **false** *are unsatisfiable,*
- *f terminates after time polynomial in the length of φ,*
- *whenever f maps φ to* **false**, *then it also maps $\varphi \wedge l$ to* **false** *for some literal l, and*
- *whenever f maps φ to* **true** *and whenever we have some assignment $a : \mathcal{V} \to \mathbb{B}$ that is satisfying for φ, then for all literals l, if $a \models l$, then f maps $\varphi \wedge l$ to* **true**.

The length of a boolean formula is defined to be the number of literals that occur in it. Given a partial assignment $g : \mathcal{V} \rightharpoonup \mathbb{B}$ to a set of variables $\mathcal{V}$, we define $\Theta(g)$ to be a boolean formula in conjunctive normal form that encodes g. More concretely, given a partial assignment $g = \{v_{i_1} \mapsto b_1, \ldots, v_{i_n} \mapsto b_n\}$, we define:

$$\Theta(g) = \left(\bigwedge_{j \in \{1,\ldots,n\}, g(v_{i_j}) = \mathbf{true}} v_{i_j} \right) \wedge \left(\bigwedge_{j \in \{1,\ldots,n\}, g(v_{i_j}) = \mathbf{false}} \neg v_{i_j} \right)$$

We are now ready to define *backdoors*, which come in two variants:

Definition 4 (Weak backdoor)**.** *Given a Boolean formula φ over a set of variables $\mathcal{V}$ and a partial valuation g to $\mathcal{V}$, we say that g is a* weak backdoor *for φ for some polynomial time sub-solver f if f returns* **true** *for $\varphi \wedge \Theta(g)$.*

Definition 5 (Strong backdoor)**.** *Given a Boolean formula φ over a set of variables $\mathcal{V}$ and a subset of variables $V \subseteq \mathcal{V}$, we say that V is a* strong backdoor *for φ for some polynomial time sub-solver f if for every partial variable valuation g with domain V, we have that f returns either* **true** *or* **false** *for $\varphi \wedge \Theta(g)$.*

The definitions of weak and strong backdoors are independent of the concrete choice of a polynomial-time sub-solver. In order to reason about the performance of CDCL-based SAT solvers, it however makes sense to fix the sub-solver to refer to a procedure that performs unit propagation and checks if after saturating the partial assignment according to these concepts, a (partial) valuation emerges that satisfies all clauses.

Note that combining unit propagation with the pure literal rule does not lead to a polynomial-time sub-solver according to Definition 3, as it does not satisfy the last constraint of the definition. In particular, such a sub-solver would map the boolean formula

$$\varphi = (a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$$

to **true**, but would map $\varphi \wedge \neg a$ to *unknown*.

A weak backdoor g is essentially a recipe for deriving a satisfying assignment for a satisfiable SAT formula. If a SAT solver can derive an assignment by only applying unit propagation and the pure literal rule, then the assignment g is in some sense an "almost-solution" to the SAT problem - all of the important decisions have already been made when deriving g, and g only needs to be completed.

A strong backdoor V is – in a sense – even more. It pinpoints the important variables that have to be filled with values in order for the SAT instance to "unravel" itself. For solving the SAT instance, one then only needs to try all assignments to the variables in V and check for all of them if the sub-solver finds an assignment.[4]

Every satisfiable SAT instance has a trivial weak backdoor and a trivial strong backdoor, and every unsatisfiable SAT instance has a trivial strong backdoor. In the case of strong backdoors, only those of size $O(poly(\log_2(|\mathcal{V}|)))$ give rise to a polynomial-time algorithm for finding a satisfying assignment when the backdoor is already known. Consequently, the interest in backdoors lies in the observation that many SAT instances in fact have *small* backdoors.

As an example, Williams et al. (2003) list five practical SAT instances that have weak backdoors consisting of less than 1.12% of the instances' variables (when using unit propagation as a sub-solver). But also Kilby et al. (2005) find small weak and strong backdoors in many practical SAT instances. They note that:

> *Our experiments showed that there is very little overlap between backbones and backdoors. In addition, they demonstrated that problem hardness appears to be correlated with the size of strong backdoors, and weakly correlated with the size of the backbone, but does not appear to be correlated to the size of weak backdoors nor their number.*

[4]Note that we did not require the sub-solver to actually find a satisfying assignment. However, even if it does not, by repeated calls to the sub-solver the assignment can be computed. At most $2|\mathcal{V}|$ many calls to the sub-solver are necessary after it found the SAT instance to be satisfiable.

In this context, a backbone is a partial valuation that is a subset of every satisfying valuation to a SAT instance. It should be added, though, that both Williams et al. (2003) and Kilby et al. (2005) only provide a limited view onto the properties of practical SAT instances as only few benchmarks were considered.

Some problems have additional upper bounds on the sizes of smallest backdoors. For example when checking combinational circuits for equivalence, as in Section 2.2.3, we have a trivial upper bound of the sizes of both smallest weak and smallest strong backdoors. When the values of the input signals to the circuits are fixed, all of the other variable values can be deduced by unit propagation. Thus, the number of input signals to the circuits that are to be checked for equivalence provides us with the size of a strong backdoor, and the size of a weak backdoor if the SAT instance is satisfiable.

The fact that many SAT instances have small backdoors (for unit propagation as sub-solver) does not mean that CDCL-based SAT solvers are able to solve these instances quickly. In particular, the variable selection heuristics employed in modern CDCL solvers are not guaranteed to branch on these variables first, which is needed for making direct use of the backdoor. However, these heuristics are geared towards choosing variables that are involved in many conflicts in the search process first, and these are more likely to be backdoor variables. Furthermore, even if the solver branches on a few non-backdoor variables first, unit propagation is still able to determine the extensibility of the current set of decisions to a satisfying assignment after the backdoor variables' values have been fixed. So making use of a backdoor in practice does not strictly require the backdoor variables to be branched on first. Thus, small backdoors intuitively help with the solution of a SAT instance, but the literature on the subject does not contain a satisfactory formalization and proof of this intuition.

2.6 SAT encodings in practice

> There are usually many ways to model a given problem in CNF, and few guidelines are known for choosing among them. [...] In short, CNF modelling is an art and we must often proceed by intuition and experimentation.
>
> *Prestwich, 2009, Chapter 2.5*

SAT solvers can show their reasoning efficiency only after a (practical) problem has been encoded into a satisfiability (SAT) instance. Naive encodings can lead to huge SAT instances that are already difficult to store. Furthermore, even if such blow-up is avoided, the encoding can have a huge impact on the computation times of the solver.

In the Sudoku and graph coloring applications in Section 2.2, we use, for instance, a pretty *wasteful* encoding. In particular, in order to encode a $n = 3$ Sudoku (with 9×9 cells), we used $9 \cdot 9 \cdot 9$ variables. It is relatively easy to save many variables. In particular, there is no need to use 9 variables per cell in the Sudoku. Rather, we could binary-encode the cell contents in order to only need 4 bits per cell, saving more than half of the overall variables. Similarly, when performing graph coloring, we can binary-encode the colors used.

However, saving variables does not mean that SAT solving commences faster, so the question which encoding is better is fair. Similarly, for problems that do not lend themselves to a straight-forward CNF encoding, there are multiple ways in which the problem can be broken down into CNF Form, and the choice made can positively or negatively influence the performance of a SAT solver.

So from a practical perspective, the correct choice of CNF encoding is of relevance, which is why we want to discuss them in this section. In some cases, we can also make use of application insight when applying a SAT solver, which increases the performance of solving the resulting instances as well.

2.6.1 Using problem insight & symmetry breaking

In many cases, insight into the problem encoded into a SAT instance can be used in order to make the SAT instance easier to solve.

As an example, consider the graph coloring problem from Section 2.2.1. We encoded the coloring problem for some graph $G = (V, E)$ into a SAT instance φ over a set of variables $\{x_{v,c}\}_{v \in V, c \in \{1,\ldots,n\}}$, where n is the number of colors. A variable $x_{v,c}$ is assigned a value of **true** in an assignment if and only if vertex v is assignment color c.

Let the vertices be numbered $V = \{v_1, v_2, \ldots, v_m\}$. Without loss of generality, we can add the clause $x_{v_1,1}$ to φ. The resulting formula is equi-satisfiable to the original version of φ – as none of the colors is special in any way, a coloring remains valid when permuting the color numbers. Also, as we are only adding a clause when modifying the SAT instance, a solution to the modified SAT instance remains to be a valid coloring.

The argument can be further generalized. For example, let $k = \min\{1, \ldots, m \mid (v_1, v_k) \in E\}$. If k is defined, then we can also add the clause $x_{v_k,2}$ to φ. As we know that v_k cannot have the same color as v_1, and all colors have the same roles, it does not hurt to assign color 2 to v_k.

We can even further generalize the argument. Let $K \subseteq \{1, \ldots, m\}$ be the vertex indices of a *clique* in G. A clique is a subset of vertices $V' \subseteq V$ such that for every $v, v' \in V'$, if $v \neq v'$, then we have $(v, v') \in E$. We know that all the vertices with indexes in K must have different colors, so instead of adding $x_{v_1,1}$ and $x_{v_k,2}$ as clauses to φ, we can instead add

$$\bigwedge_{j \in \{1,\ldots,|K|\}} x_{v_{K_j},j}$$

as clauses, where K_j is the jth-smallest element of K. The added unit clauses fix the values of $|K|$ literals immediately and thus allow the SAT solver to apply unit propagation on a substantial number of variables (and clauses). Naturally, we would want to find a largest clique K to make use of this effect as well as possible. Unfortunately, finding a largest clique in a graph is an NP-complete problem as well, which suggests that finding a largest clique first is probably overkill.

As an alternative, we can apply a slightly more complex solution space reduction scheme that however also generalizes to other settings. We base our presentation on the following formal definition:

Definition 6 (Solution transformer)**.** *Let a set of variables $\mathcal{V}$, a boolean formula φ and its set of satisfying assignments P be given.*

We call a function $f : (\mathcal{V} \to \mathbb{B}) \to (\mathcal{V} \to \mathbb{B})$ a solution transformer *if for every assignment $p \in P$, we also have $f(p) \in P$ and $f(f(p)) = f(p)$.*

Note that a solution transformer may map elements that are not in P to arbitrary other assignments.

Example 6. Let φ be a SAT instance encoding a graph coloring problem over a set of $\mathcal{V} = \{x_{v,c}\}_{v \in V, c \in \{1,\ldots,n\}}$, where n is the number of colors and $V = \{v_1, \ldots, v_m\}$ is the set of vertices in the graph. The function $f : (\mathcal{V} \to \mathbb{B}) \to (\mathcal{V} \to \mathbb{B})$ with

$$f(p)(x_{v_i,j}) = \begin{cases} p(x_{v_i,j}) & \text{if } p(x_{v_1,1}) = \textbf{true} \\ \textbf{true} & \text{if } i = 1, j = 1, p(x_{v_1,1}) = \textbf{false} \\ \textbf{false} & \text{if } i = 1, j \neq 1, p(x_{v_1,1}) = \textbf{false} \\ p(x_{v_i,j}) & \text{if } i > 1, j \neq 1, p(x_{v_1,j}) = \textbf{false}, p(x_{v_1,1}) = \textbf{false} \\ p(x_{v_i,1}) & \text{if } i > 1, j \neq 1, p(x_{v_1,j}) = \textbf{true}, p(x_{v_1,1}) = \textbf{false} \\ p(x_{v_i,j'}) & \text{if } i > 1, j = 1, p(x_{v_1,1}) = \textbf{false} \text{ for the unique value of } j' \\ & \qquad \text{such that } p(x_{v_1,j'}) = \textbf{true} \end{cases}$$

is a solution transformer for φ and $\mathcal{V}$. ★

The solution transformer from Example 6 intuitively does the following: if in the variable valuation given to the transformer, vertex v_1 has color 1, it does not alter the valuation. Otherwise, the colors of vertex v_1 and color 1 are swapped. The transformer satisfies the constraint that valid solutions to the SAT problem are mapped to other valid solutions to the SAT problem. Also, applying the transformer twice is equivalent to applying the transformer once, so it satisfies the requirements from Definition 6.

Solution transformers give rise to a simple approach for reducing the number of solutions to a SAT instance. Given a function f that we know to be a valid transformer for a SAT instance without knowing the set of solutions P, we can add clauses to φ that exclude some or all candidate solutions $p \in (\mathcal{V} \to \mathbb{B})$ for which $f(p) \neq p$ holds. As we know in such a case that if p is a solution to the SAT problem for φ, then $f(p)$ is a solution to the SAT problem for the modification of φ, excluding such candidate solutions is sound. Of course, if f is the identity function (which is a valid solution transformer), no solutions are excluded, so in order to benefit from this effect, a non-trivial transformer must be used. By excluding solutions in a SAT instance (without excluding all of them), we allow the SAT solver to prune the search space more quickly, which is beneficial for unsatisfiable SAT instances.

A solution transformer that applies to many problems is *lexicographical minimality*, which maps a candidate solution to an equivalent one whose binary string representation is lexicographically least among the equivalent solutions.

In the case of graph coloring, we call two colorings equivalent if they can be translated to each other by applying an isomorphism between the vertex colors. We can then implement the lexicographical minimality solution transformer by ordering the nodes into a list $v_1, \ldots, v_m$ and requiring that some vertex $j \in \{1, \ldots, m\}$ can only have a color of c if color $c-1$ has already been used for one of the vertices $\{v_1, \ldots, v_{j-1}\}$. Violating this constraint would imply "jumping over" color $c-1$, proving that a lexicographically smaller solution could be obtained by permuting the colors $\geqslant c-1$. We can add the following clauses to the SAT instance to implement this constraint:

$$\bigwedge_{j \in \{1,\ldots,m\}, c \in \{2,\ldots,n\}} \neg x_{v_j,c} \vee \bigvee_{j' < j, c' \geqslant c-1} x_{v_{j'},c'}$$

Note that for $j = 1$ and $c = \{2, \ldots, n\}$, the clauses generated by the above constraint are unit. Thus, without stating it explicitly, we obtain the constraints that the first vertex should not have any of the colors $2, \ldots, n$ in addition to other constraints. By unit propagation and the clause that every vertex must have one color, the SAT solver can thus deduce that the first vertex must have color 1.

The solution transformer and the corresponding clausal implementation of the lexicographical minimality constraints can be further refined if a graph isomorphism is known. While in general, a minimal coloring does not necessarily assign the same colors to vertices that are isomorphic to each other, we can require the labeling not to become lexicographically smaller by applying the graph isomorphism to the coloring.

Adding clauses to restrict the space of possible solutions without excluding all solutions is also called *symmetry breaking* in the literature, as it breaks the symmetry in the solution space of a SAT instance. This is particularly helpful in settings in which there is a lot of symmetry (such as in our graph coloring application). CDCL-style SAT solvers typically do not have symmetry detection integrated into them. Thus, they often traverse parts of the search space that are isomorphic to other parts that have already been explored, and they similarly often learn clauses that represent constraints that could also be instantiated for many other permutations of the solution variables. Breaking the symmetry to some extent also helps the variable selection heuristic in the SAT solver to make informed choices by not offering many essentially equivalent possible variables to branch on.

Having said that, the number of constraints added by Equation 2.6.1 can be huge, which can slow down the SAT solving process (Sakallah, 2009, p. 318). As a remedy, it often makes sense to only add some symmetry breaking clauses (e.g., those up to a certain length).

While the formalization of a solution transformer is often not needed in order to apply symmetry breaking (as we have seen above for the case of graph coloring), it is still helpful to keep that concept in mind. Especially when for some application, there are multiple ways in which symmetry can be broken, theory helps with determining whether they can be combined. If for some application, two solution transformers f and g exist, then adding clauses that rule out solutions p such that $p \neq f(p)$ and clauses that rule out solutions p such that $p \neq g(p)$ may lead to an unsatisfiable SAT instance. However, we can combine f and g without introducing *spurios unsatisfiability* if $f \circ g$ is a solution transformer as well and for every p such that $p = (f \circ g)(p)$, we have $p = g(p)$ and $p = f(p)$ as well.

Note that we could also apply symmetry breaking in the SAT instance obtained from our pidgeon hole scenario, and it would actually become trivial in the process (i.e., unit propagation alone would solve it). However, symmetry breaking does not make our study of the pidgeon hole problem SAT instances obsolete, as a SAT instance can contain the pidgeon hole problem as a *sub-problem* after a few variable values have been fixed, leading to the same fruitless search process that we have seen in Section 2.5.2. Modifying a SAT instance to eliminate the problem on all sub-problems can be very difficult and can lead to substantial blow-up of the SAT instance.

As a final remark, the graph coloring example is also great for showing another problem-specific simplification: there is in fact no need to have clauses in the instances that encode that a vertex must only have at most one color assigned: if a vertex has two or more colors assigned, then our encoding guarantees that picking *any* of the possible colors leads to a valid coloring. Thus, we can obtain the effect of these clauses by *post-processing* a solution. Whether such a simplification helps with SAT solving is however problem-dependent.

2.6.2 Encoding of arbitrary boolean formulas

Literature: *Prestwich, 2009, Section 2.2.1–2.2.2*

Assume that we are given a boolean formula φ over a set of variables $\mathcal{V}$. If φ is not in conjunctive normal form, we cannot immediately apply a DPLL-style SAT solver as it expects its input to be in this form. The canonical way of solving this problem is to translate φ into CNF using De Morgan's law and the distributivity law. This process can however lead to an exponential blow-up in the length of the formula, giving this approach little attractivity.

The problem is commonly solved by applying the *Tseitin transformation*. Essentially, the process operates in precisely the same way as our encoding of the gates for the circuit equivalence problem in Section 2.2.3: we allocate one additional variable for every sub-formula in a Boolean formula and constrain it such that the new variable is **true** if and only if the sub-formula is satisfied. The new variable is then added to the set of variables.

The resulting SAT formula is then *equisatisfiable* to the original formula: it has a satisfying assignment if and only if the original formula has a satisfying assignment, and restricting a satisfying assignment of the modified formula to the variables that already appear in the original formula leads to a satisfying assignment for the original formula.

Example 7. Let $\varphi = (a \vee (b \wedge c) \vee (c \wedge \neg(d \vee e))) \wedge f$ for the set of variables $\mathcal{V} = \{a, b, c, d, e, f\}$. For some valuation h to $\mathcal{V}$, φ is satisfied if and only if h can be extended to a valuation over $\mathcal{V}' = \{a, b, c, d, e, f, t_1, t_2, t_3, t_4, t_5\}$ that satisfies the following clauses:

$$\begin{array}{rrrrr}
t_1 & \neg t_2 \vee a \vee t_3 \vee t_4 & \neg t_3 \vee b & t_4 \vee \neg c \vee t_5 & t_5 \vee \neg d \\
\neg t_1 \vee f & t_2 \vee \neg a & \neg t_3 \vee c & \neg t_4 \vee \neg t_5 & t_5 \vee \neg e \\
\neg t_1 \vee t_2 & t_2 \vee \neg t_3 & t_3 \vee \neg b \vee \neg c & \neg t_4 \vee c & \neg t_5 \vee d \vee e \\
t_1 \vee \neg f \vee \neg t_2 & t_2 \vee \neg t_4 & & &
\end{array}$$

Here, t_1 represent the value of the outermost conjunction in φ, t_2 represents the left conjunct in φ, and so on. The unit clause t_1 has the special aim of enforcing φ to hold on the SAT instance in CNF form. ★

In a Tseitin encoding, we start with a boolean formula φ over a set of variables $\mathcal{V}$ and translate it to a CNF formula φ' over the variables $\mathcal{V} \cup \{t_1, \ldots, t_n\}$ such that

1. φ' is equi-satisfiable to $t_1 \leftrightarrow \varphi$,
2. every valuation $p : \mathcal{V} \to \mathbb{B}$ that satisfies φ can be extended to a satisfying valuation p' for φ' with $p'(t_1) = \textbf{true}$, and

3. restricting a satisfying valuation p to φ' with $p(t_1) = \textbf{true}$ to $\mathcal{V}$ yields a satisfying valuation to φ.

For every m-ary disjunction, conjunction, or negation operator in the formula, φ' contains $m+1$ clauses. In Example 7, we required φ to hold on a satisfying valuation by adding the unit clause t_1.

The CNF formula φ' obtained by the Tseitin transformation can be used as part of a larger SAT instance. For instance, assume that we want to encode $\psi = \varphi_1 \oplus \varphi_2$ to a CNF-form SAT instance, where $\oplus$ denotes the exclusive or operator. Let both φ_1 and φ_2 range over some set of variables $\mathcal{V}$. We can translate φ_1 and φ_2 separately to CNF formulas φ'_1 and φ'_2 that are equivalent to $t_1 \leftrightarrow \varphi_1$ and $t'_1 \leftrightarrow \varphi_2$, respectively, and use them to build the overall SAT instance

$$\psi' = \varphi'_1 \wedge \varphi'_2 \wedge (t_1 \vee t'_1) \wedge (\neg t_1 \vee \neg t'_1). \tag{2.4}$$

The last two clauses implement the exclusive or constraint. The SAT instance ψ' ranges over $\mathcal{V}$ and the variables encoded by the Tseitin transformations of φ_1 and φ_2.

In many cases, some clauses can be left out when performing a Tseitin encoding. In Example 7, we included the clause $t_1 \vee \neg f \vee \neg t_2$ despite the fact that there is also a t_1 clause. This is because we have treated the Tseitin encoding as a function that computes a CNF formula for a constraint that ensures that $t_1 = \varphi(a,b,c,d)$, and the fact that t_1 should hold was not part of the transformation.

But there are also some other clauses whose superfluousness is less obvious. For instance, we can remove the clause $t_2 \vee \neg a$. While leaving out the clause extends the set of models of the SAT instance, no new valuations to $\{a,b,c,d,e,f\}$ that can be extended to models of $\{a,b,c,d,e,f,t_1,t_2,t_3,t_4,t_5\}$ are added in the process. This is because t_2 represents a sub-formula that appears in φ without any complementation, and outside of the clauses added by the Tseiting encoding, the variable that represents the value of the overall formula φ appears only with positive polarity in the SAT formula. Thus, having a value of $t_2 = \textbf{false}$ only makes it harder for the SAT solver to find a solution and never makes it easier, for no valuation of the other variables. Thus, we do not need clauses that enforce that the SAT solver does not "accidentally" set t_2 to **false** when t_2 could be set to **true**. In Equation 2.4, we use the *anchor variables* of φ and φ' both complemented and non-complemented. Thus, we do not have a similar effect and must use a complete Tseitin encoding.

We can formalize the idea as follows. Given a SAT-formula φ, we assign to every sub-formula a polarity of $\{-1,0,1\}$. We say that φ has a polarity of 1 if in the overall SAT formula that includes clauses that encode $\psi = t_1 \leftrightarrow \varphi$, except for the clauses that encode ψ, t_1 only occurs as literal t_1, and say that it has polarity -1 if it only occurs as literal $\neg t_1$ in the overall SAT instance except for the clauses that encoding ψ. In all other cases, it has polarity 0. For all sub-formulas of φ, we define the polarity recursively over the structure of φ. Let $\tilde{\varphi}$ be a sub-formula of φ (or $\varphi = \tilde{\varphi}$). We define:

- if $p = \mathit{polarity}(\tilde{\varphi})$ and $\tilde{\varphi} = \tilde{\varphi}_1 \vee \tilde{\varphi}_2$, then $\mathit{polarity}(\tilde{\varphi}_1) = p$ and $\mathit{polarity}(\tilde{\varphi}_2) = p$.
- if $p = \mathit{polarity}(\tilde{\varphi})$ and $\tilde{\varphi} = \tilde{\varphi}_1 \wedge \tilde{\varphi}_2$, then $\mathit{polarity}(\tilde{\varphi}_1) = p$ and $\mathit{polarity}(\tilde{\varphi}_2) = p$.
- if $p = \mathit{polarity}(\tilde{\varphi})$ and $\tilde{\varphi} = \neg\tilde{\varphi}_1$, then $\mathit{polarity}(\tilde{\varphi}_1) = -1 \cdot p$.

If a sub-formula occurs multiple times in φ, and the above rules would set it to more than one possible value at the same time, this conflict is resolved by setting its polarity to 0 instead.

Now whenever some sub-formula φ with its associated variable t_i introduced during the Tseitin transformation has polarity 1, we can leave out the clauses for the top-level operation in φ with literal t_i. Likewise, whenever some sub-formula φ with its associated variable t_1 has polarity -1, we can leave out the clauses for the top-level operation in φ with literal $\neg t_1$.

Example 8. Reconsider Example 7 with the non-CNF formula φ. The encoded formula is equivalent to $\phi = t_1 \wedge (t_1 \leftrightarrow \varphi)$, where the last conjunct is encoded by a Tseitin transformation. As t_1 only occurs positively in the computed CNF except for the Tseitin clauses, we have $\mathit{polarity}(\varphi) = 1$. Applying the recursive definition of the *polarity* function, we obtain:

- $\mathit{polarity}(a \vee (b \wedge c) \vee (c \wedge \neg(d \vee e))) = 1$, with the associated Tseitin variable t_2,
- $\mathit{polarity}(b \wedge c) = 1$, with the associated Tseitin variable t_3,
- $\mathit{polarity}(c \wedge \neg(d \vee e)) = 1$, with the associated Tseitin variable t_4, and

- *polarity*$(d \vee e) = -1$, with the associated Tseitin variable t_5.

By leaving out clauses that we found to be superfluous by the polarity argument, we arrive that the following (overall) CNF SAT instance:

$$\begin{array}{rrrrr}
t_1 & \neg t_2 \vee a \vee t_3 \vee t_4 & \neg t_3 \vee b & t_4 \vee \neg c \vee t_5 & t_5 \vee \neg d \\
\neg t_1 \vee f & t_2 \vee \neg a & \neg t_3 \vee c & \neg t_4 \vee \neg t_5 & t_5 \vee \neg e \\
\neg t_1 \vee t_2 & t_2 \vee \neg t_3 & t_3 \vee \neg b \vee \neg c & \neg t_4 \vee c & \neg t_5 \vee d \vee e \\
t_1 \vee \neg f \vee \neg t_2 & t_2 \vee \neg t_4 & & &
\end{array}$$

Thus, we were able to reduce the overall number of clauses from 17 to 10. ★

2.6.3 Encoding of numbers

Literature: *Prestwich, 2009, Section 2.2.4*

After we have discussed how to use problem insight when encoding a problem into a SAT formula and after discussing how to encode arbitrary boolean functions with the Tseitin encoding, we will next cover some *encoding patterns* that are useful when a problem does not have a natural boolean encoding, i.e., when some variables have a non-boolean finite domain. In many problems that can be solved by applying a SAT solver, we have to perform such an encoding before the solver can be used. Without loss of generality, we consider some subset $\{a, a+1, \ldots, b-1, b\}$ of the natural numbers as this domain here.

In the solution to the graph coloring problem in Section 2.2.1, we discussed a first example for such an encoding. Formally, a coloring for some graph $\mathcal{G} = (V, E)$ into a set of colors $\{1, \ldots, n\}$ is given by a function $c : V \to \{1, \ldots, n\}$. In Section 2.2.1, we encoded c by first of all allocating n variables for each vertex $v \in V$. For each vertex, we then added a clause that ensures that at least one of these n variables has to have a value of **true**. We finally added $\frac{1}{2} \cdot n \cdot (n-1)$ clauses for each vertex to prevent that two of these variables to have a value of **true** at the same time. This is also called the *direct encoding*, the *pairwise encoding*, or the *one-hot encoding* of the variable domain $\{1, \ldots, n\}$. Its advantages are its simplicity and the fact that the many two-literal clauses help the SAT solver with traversing the search space in an efficient way by providing ample opportunity for unit propagation. The downside of the approach is that we need $O(n^2)$ many clauses and $O(n)$ many variables to encode a single number.

As an alternative that is more compact regarding the clauses, but requires more variables, we can use a *ladder encoding*. Assume that we want to encode a variable in the domain $\{0, \ldots, n-1\}$. As in the direct encoding, we allocate a number of variables $\mathcal{V}' = \{x_0, \ldots, x_{n-1}\}$ for which we want to enforce that precisely one of the variables in $\mathcal{V}'$ is set to **true**. Instead of using $O(n^2)$ many clauses, we introduce an additional set of *ladder variables* $\{y_0, \ldots, y_{n-1}\}$ for which for every $i \in \mathbb{N}$, y_i shall be **true** if for some $j \leqslant i$, x_j is true. We can enforce this by first of all adding the constraints

$$(x_0 \vee \neg y_0) \wedge (\neg x_0 \vee y_0) \wedge \bigwedge_{0 \leqslant i \leqslant n-2} \left((y_{i+1} \vee \neg x_{i+1}) \wedge (y_{i+1} \vee \neg y_i) \wedge (\neg y_{i+1} \vee x_{i+1} \vee y_i) \right).$$

By furthermore adding the unit constraint y_{n-1}, it can then be enforced that for some $i \in \mathbb{N}$, x_i must be true. In order to enforce that for at most one $i \in \{0, \ldots, n-1\}$, we have that x_i is **true**, we furthermore add the constraints

$$\bigwedge_{0 \leqslant i \leqslant n-2} (\neg y_i \vee \neg x_{i+1}).$$

We can visualize the idea of a ladder encoding in a graph. Figure 2.22 shows an example for $n = 7$. The values of the variables $x_0, \ldots, x_6$ and $y_0, \ldots, y_6$ are shown along the different steps of the x-axis of the figure, where we use the usual encoding of **false** and **true** by 0 and 1. It can easily be seen that the ladder, which is encoded in the y_i variables, has a step at index 4. The ladder encoding ensures that the steps of the ladder can only "go up", so that at most one of the x_i variables has a **true** value.

The ladder encoding requires $2 \cdot n$ many variables, but in contrast to the direct encoding requires only $4n - 1$ many clauses.

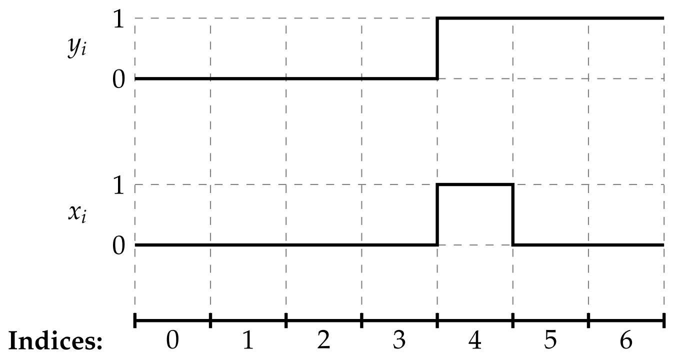


Figure 2.22: Visualization of the ladder encoding.

The ladder encoding can be further refined to a *phase transition encoding* that requires only $n-1$ variables and $n-2$ many clauses. In this encoding, we allocate a set of variables $\{y_0, \ldots, y_{n-2}\}$ and add the following constraints to the SAT instance:

$$\psi = \bigwedge_{0 \leqslant i \leqslant n-3} (\neg y_i \vee y_{i+1})$$

The clauses have the effect that if for some $i \in \mathbb{N}$, we have that y_i is **true**, then also for all $j > i$, we have that y_j is **true**. Likewise, if for some $i \in \mathbb{N}$, we have that y_i is **false**, then for all $j < i$, we have that y_j is **false**. Thus, these constraints enforce that there is at most one *phase transition* from **false** to **true** in the sequence $y_0, \ldots, y_{n-2}$. We can then use this phase transition to encode that the variables $y_0, \ldots, y_{n-2}$ encode some value in $\{0, \ldots, n-1\}$: if the phase transition is between variables $h-1$ and h, i.e., we have $y_{h-1} =$ **false** and $y_h =$ **true**, then the valuation of the variables $y_0, \ldots, y_{n-2}$ encodes the value h. The values $h = 0$ and $h = n-1$ are special cases, as they are represented by the all-**false** and all-**true** valuations of the variables $y_0, \ldots, y_{n-2}$. Due to the monotonicity of the y_i variable values enforced by the constraints ψ, it suffices to check if $y_0 =$ **true** or if $y_{n-2} =$ **false** to test for the encoded values 0 and $n-1$.

As in the phase transition encoding, we need to look at two variables in the SAT instance in order to determine if value $k \in \{0, \ldots, n-1\}$ is encoded, it is a bit more difficult to use in practice than the direct encoding. Whenever in the direct encoding, we would have a clause of the form $\neg x_i \vee \psi$ (where ψ does not contain another occurrence of the variables $x_0, \ldots, x_{n-1}$), we would replace this constraint by $y_{i-1} \vee \neg y_i \vee \psi$ if $i \in \{1, \ldots, n-2\}$, by the constraint $y_{n-2} \vee \psi$ if $i = n-1$, or by the constraint $\neg y_0 \vee \psi$ if $i = 0$. Thus, the overall number of literals in a SAT instance can grow when applying a phase transition encoding. Even more, whenever in the direct encoding, we would have a clause of the form $x_i \vee \psi$ (where ψ does not contain another occurrence of the variables $x_0, \ldots, x_{n-1}$), we would replace this constraint by the **two** constraints $\neg y_{i-1} \vee \psi$ and $y_i \vee \psi$ if $i \in \{1, \ldots, n-2\}$, by the single constraint $\neg y_{n-2} \vee \psi$ if $i = n-1$, or $y_0 \vee \psi$ if $i = 0$. Thus, also the overall number of clauses can grow with the phase transition encoding.

Thus, whether it makes sense to apply a phase transition encoding depends heavily on the problem at hand: if there are many constraints that refer to an encoded value, then the phase transition encoding may be the wrong choice. If there are only few constraints that refer to an encoded non-boolean variable, then it may be worth a try.

A third possibility to encode an integer value in $\{0, \ldots, n-1\}$ is to *binary-encode* it. The resulting encoding is also called a *bitwise encoding* or a *log-encoding* as it requires $k = \lceil \log_2(n) \rceil$ many boolean variables. If n is a power of 2, there are no additional constraints needed to enforce that the boolean variables $y_0, \ldots, y_{k-1}$ encode a value that is not $\geqslant n$. In the general case, at most $\lceil \log_2(n) \rceil$ clauses are needed.

When log-encoding a value, we only need $O(\log_2 n)$ many clauses to enforce that a valid value is encoded and $O(\log_2 n)$ many variables. However, as in the phase transition encoding, the clauses that actually use the encoded value from $\{0, \ldots, n-1\}$ become more complex, and their number may also need to increase. Furthermore, the log-encoding reduces the value of unit propagation, as oftentimes, conflicts during search can only be detected if all variables in $y_0, \ldots, y_{k-1}$ have been assigned a value.

To get rid of this drawback, we can mix the direct encoding and the log-encoding (Frisch et al., 2005). In order to encode a value $u \in \{0, \ldots, n-1\}$, we then have the variables $\{x_0, \ldots, x_{n-1}\}$ as well as the variables $\{y_0, \ldots, y_{k-1}\}$ for $k = \lceil \log_2(n) \rceil$. In addition to the constraints for the log-encoding, we need constraints that require that the variables $\{y_0, \ldots, y_{k-1}\}$ binary-encode some value h if and only if x_h is set to **true**. As $\{y_0, \ldots, y_{k-1}\}$ can only encode a single value in the domain $\{0, \ldots, n-1\}$, this enforces that precisely one proposition in $\{x_0, \ldots, x_{n-1}\}$ is set to **true**. In this mixed encoding, we need $n + \lceil \log_2(n) \rceil$ variables overall, and $O(n \log n)$ many clauses.

Now that we have discussed four different encodings for more complex variable domains into boolean variables, the question is fair which one is the best. Unfortunately, there are only very rough guidelines when to apply which encoding. While the direct encoding makes good use of unit propagation, the large number of clauses that are necessary for a direct encoding has a negative impact on the SAT solver's overall performance. On the other hand, the bitwise encoding does not allow to make good use of unit propagation, but it is significantly more compact, giving it an advantage on instances that are believed to be relatively easy to solve. We will continue the discussion of this effect in Section 2.6.5.

2.6.4 Encoding of k-out-of-n constraints

Sometimes, we may want to encode a *selection of objects* in a problem that we want to solve with the support of a SAT solver. In such a case, we have a set of elements $X \subseteq \{0, \ldots, n-1\}$ given, and we want to encode that some subset $K \subseteq X$ has $|K| \leqslant k$ or $|K| = k$ for some value $k \in \mathbb{N}$. Common applications for such encodings are *selection problems*, where a set of objects is to be picked such that this set satisfies certain requirements. The *maximum clique* or *maximum independent set* problems are examples (where the size of the largest clique is approximated step-wise).

Given the number of different approaches to the encoding of a single element from a domain $\{0, \ldots, n-1\}$ that we have seen in the previous subsection, all having different properties, it makes sense to ask the question how such a set K can be encoded in an efficient way. We will discuss two ways of doing so here.

First of all, we can generalize the ladder encoding for this purpose. We start with the variables $\{x_0, \ldots, x_{n-1}\}$ that encode whether some element is in the set K. For the ladder constraints, we need the set of additional variables $\{y_{i,j}\}_{0 \leqslant i \leqslant n, 0 \leqslant j \leqslant k}$. A variable value of $y_{i,j} = \textbf{true}$ should denote that from the first i elements of the set X, precisely j many have been selected to be in K. More formally, we need the following constraints:

$$y_{0,0} \wedge \bigwedge_{1 \leqslant j \leqslant k} \neg y_{0,j} \wedge \bigwedge_{0 \leqslant i < n, 0 \leqslant j < k} (\neg y_{i,j} \vee \neg x_i \vee y_{i+1,j+1}) \wedge (\neg y_{i,j} \vee x_i \vee \neg y_{i+1,j+1})$$
$$\wedge \bigwedge_{0 \leqslant i < n, 0 \leqslant j < k} (y_{i,j} \vee \neg y_{i+1,j+1}) \wedge \bigwedge_{0 \leqslant i < n, 0 \leqslant j \leqslant k} (\neg y_{i,j} \vee y_{i+1,j}) \wedge \bigwedge_{0 \leqslant i < n} (\neg y_{i,k} \vee \neg x_i)$$

By adding the unit clause $y_{n,k}$ to the SAT instance, we can then enforce that K has precisely k elements. Overall, we need $O(n \cdot k)$ variables and $O(n \cdot k)$ many clauses.

For large values of n and k, the generalization of the ladder encoding to the k-out-of-n case is quite wasteful. As an alternative, we can use *sorting networks*. Figure 2.23 shows such a network. It describes a schedule of pairwise compare-and-swap operations that, if executed from left to right, ensure that the resulting values are sorted. A compare-and-swap operation is also called a *comparator module*, and the lines are called *signal lines* in the following.

Applied to the boolean case, a sorting network sorts the **false** and **true** values of the variables $\{x_0, \ldots, x_{n-1}\}$ such that the values appear consecutively in the sequence $\{x'_0, \ldots, x'_{n-1}\}$. This allows to add unit clauses over $\{x'_0, \ldots, x'_{n-1}\}$ to enforce that K consists of at most or at least k elements.

To encode the functionality of a sorting network into a SAT instance, we add variables for all intermediate values along the signal lines in the sorting network. For the sorting network from Figure 2.23, we need

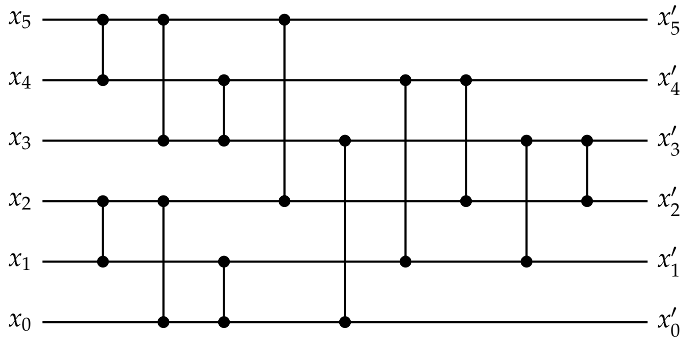


Figure 2.23: A sorting network for 6 elements

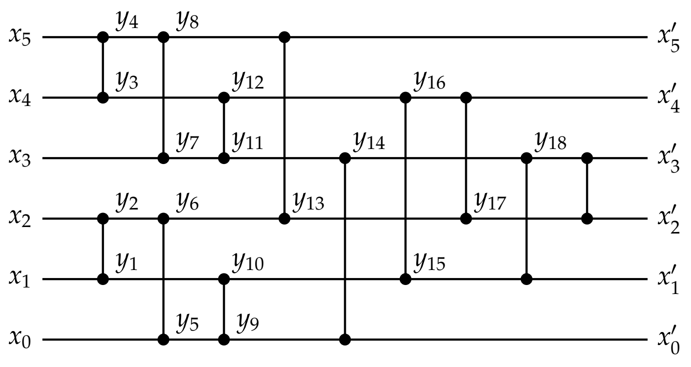


Figure 2.24: A sorting network for 6 elements with variable name for the intermediate values

18 such variables, as depicted in Figure 2.24. All of the constraints that we add then relate the input and output signals of a compare-and-swap step. As there exist sorting networks of size $O(n \log n)$ for every value of n, a sorting-network-based encoding of a k-out-of-n constraint needs $O(n \log n)$ many variables and $O(n \log n)$ many clauses. Note that the size of this encoding is independent of k.

2.6.5 Desirable properties of CNF Encodings

Literature: *Prestwich, 2009, Section 2.4; Bjork, 2009*

In this section, we have seen a couple of encodings for some problems and stated that the best encoding depends mostly on the application. But what makes an encoding good or bad (in the scope of CDCL-style SAT solving) for some application?

In a nutshell, SAT encodings are good if they allow the CDCL-style solver to make best use of unit propagation, as this causes the solver to find conflicts early without wasting time on parts of the search space for which it could easily be proven that they do not contain a solution. However, not all applications give rise to a lot of unit propagation for some encoded variable, and in such a case a compact encoding can be better, as it reduces the administrative workload of the solver. So we need to balance the suitability of a problem encoding for unit propagation against the size of the encoding.

Encoding of numbers In Section 2.6.3, we have seen a couple of ways to encode non-boolean number domains into boolean variables. To study the effect of different value encodings on how well unit propagation works, let us consider a problem with a numeric variable $v \in \{0, \ldots, n-1\}$ that is encoded into the SAT variables $\{x_0, \ldots, x_{l-1}\}$. Assume that the solver learned clauses that rule out all values for v except for value u. If $\{x_0, \ldots, x_{l-1}\}$ constitutes a direct encoding of v, this means that we have learned the clauses $\neg x_0$, $\neg x_1$, ..., $\neg x_{u-1}$, $\neg x_{u+1}$, ..., $\neg x_{l-1}$. In a direct encoding, there also exists a clause that enforces one of these variables to hold, and all of its literals are falsified except for one. Thus, a new literal is added to the current partial valuation (by unit propagation) that represents that the SAT solver derived that the value of v has to be u. The derivation of the literal x_u may then lead to more unit propagation since $\neg x_u$ can be a literal in other clauses.

Essentially, what we have seen by means of this example is that with a direct encoding, unit propagation suffices for the SAT solver to perform the "slightly different line of reasoning" that we have seen in the Sudoku example from the beginning of this chapter.

Now let us compare the direct encoding with the bitwise encoding. So we assume that $\{x_0, \ldots, x_{l-1}\}$ constitutes an bitwise encoding of v and the solver has learned clauses that rule out all values for v except for u. These clauses have between 1 and l literals. In the worst case, the solver has learned $n - 1$ clauses with l literals each. Since none of them are unit, the solver cannot perform any unit propagation and thus cannot deduce that v needs to have a value of u immediately. If the solver branches over all of variables $x_0, \ldots, x_{l-1}$ next, it can pretty quickly compute the only possible valuation for v. However, there is no guarantee that it does branch over these variables next. Thus, the solver will not detect any consequences that arise from v having a value of u at that point and that can be detected by unit propagation, making the search less efficient. An example for such a consequence is if the problem has a clause that states that either v does not have a value of u or some boolean variable p must be **true**. In such a case, unit propagation could set p to **true** at this point. Note that both the ladder encoding and the phase shift encoding do not have this problem.

Half-adder The effect of unit propagation can also be studied without assuming a non-boolean domain of some variable in the problem encoded into a SAT instance. Let us do so by considering the case of a *half-adder* for some input bits a and b and with some output bits *sum* and *carry* whose behavior is encoded into a SAT instance over a set of variables $\mathcal{V} \supseteq \{a, b, sum, carry\}$.

The behavior of the half-adder can be described by the expressions $sum \leftrightarrow (a \oplus b)$ and $carry \leftrightarrow (a \wedge b)$, where $\oplus$ denotes the exclusive or function. By translating each of these expressions into CNF in the most straight-forward way (e.g., by translating them to truth tables over three variables each and translating the truth table to CNF), we obtain the CNF clauses:

$$\begin{aligned}\varphi' = &(\neg sum \vee a \vee b)\\ &\wedge(\neg sum \vee \neg a \vee \neg b)\\ &\wedge(sum \vee \neg a \vee b)\\ &\wedge(sum \vee a \vee \neg b)\\ &\wedge(\neg carry \vee a)\\ &\wedge(\neg carry \vee b)\\ &\wedge(carry \vee \neg a \vee \neg b)\end{aligned}$$

The clause set φ' can be shown to not be *optimally propagating*, meaning that there exists a partial valuation to the variables $\{a, b, sum, carry\}$ for which there exists an assignment to a variable not in the partial assignment such that all of the completions of the partial assignment that satisfy φ' also contain this additional assignment, but this additional assignment is not found by unit propagation.

We can check if there exists such a *non-optimally propagating* partial assignment by taking a look at all possible partial assignments. Table 2.1 contains the result. In the first line of the table, we start with the empty assignment, and unit propagation does not derive any literal. This is optimal, as all variables can have any value in a satisfying assignment to φ'. In the second and third row, unit propagation can deduce from $\neg a$ and $\neg b$, respectively, that *carry* must be false. This is caused by the two clauses $\neg carry \vee a$ and $\neg carry \vee b$. Again, this is optimal, as without knowing both a and b, we do not know whether *sum* should be **true** or **false**. The partial valuation $\{carry \mapsto \textbf{true}\}$ is another interesting case. By the clauses $\neg carry \vee a$ and $\neg carry \vee b$, unit propagation can immediately derive $a \mapsto \textbf{true}$ and $b \mapsto \textbf{true}$. As φ' uniquely fixes the value of *sum* from the values of a and b, also the value of the *sum* variable follows.

Note that we left out partial valuations in Table 2.1 for which subsets thereof already give rise to enough unit propagation to obtain a full valuation for all four variables. For example, we did not consider any partial valuation in which three variables are fixed, as fixing two of the values of $\{a, b, sum\}$ gives rise to

a	b	sum	$carry$	**Derived**	**Optimal**
–	–	–	–		✓
false	–	–	–	$carry =$ **false**	✓
–	**false**	–	–	$carry =$ **false**	✓
–	–	**false**	–		✓
–	–	–	**false**		✓
true	–	–	–		✓
–	**true**	–	–		✓
–	–	**true**	–		✓
–	–	–	**true**	$a = b =$ **true**, $sum =$ **false**	✓
$\star$	$\star$	–	–	$sum = \star, carry = \star$	✓
$\star$	–	$\star$	–	$b = \star, carry = \star$	✓
–	$\star$	$\star$	–	$a = \star, carry = \star$	✓
true	–	–	$\star$	$b = \star, sum = \star$	✓
–	**true**	–	$\star$	$a = \star, sum = \star$	✓
–	–	**false**	**false**		✗
–	–	**true**	**false**		✓

Table 2.1: Systematic study of partial assignments for the half-adder from Section 2.6.5. The **Derived** column lists variables whose values can be derived from unit propagation by the fixed other values. The values **false** and **true** in the first four columns represent that the partial assignment assigns **false** or **true** to the respective variable. The value – represents that the partial assignment leaves the value open. Value $\star$ means that the table column holds for arbitrary values in $\{$**false**, **true**$\}$ of the variable in the partial assignment.

a complete valuation, and any partial valuation with three fixed variable values contains values for at least two of $\{a, b, sum\}$, so that rows 10-12 already cover these cases.

It can be seen from Table 2.1 that there is one case in which the encoding is not optimally propagating, namely the partial assignment $\{sum \mapsto$ **false**, $carry \mapsto$ **false**$\}$. Let us have a look at the impact of this partial encoding on φ'. After applying the partial assignment, the instance looks as follows:

$$\begin{aligned}\varphi' =& (\neg sum \vee a \vee b)\\ &\wedge (\neg sum \vee \neg a \vee \neg b)\\ &\wedge (sum \vee \neg a \vee b)\\ &\wedge (sum \vee a \vee \neg b)\\ &\wedge (\neg carry \vee a)\\ &\wedge (\neg carry \vee b)\\ &\wedge (carry \vee \neg a \vee \neg b)\end{aligned}$$

In this formula, we have marked clauses that are already satisfied and literals that are already falsified. As we can see, the solver cannot perform unit propagation in this case, and thus it cannot detect by unit propagation that the only way in which both $carry$ and sum can be **false** is that if both a and b are false. To counter this problem, we can add the clause $carry \vee sum \vee \neg a$ to φ'. Then, unit propagation can deduce that a has to be **false**, and by unit propagation on an already existing clause in φ', we would also get that b has to be **false**.

In the literature, encodings whose components give rise to unit propagation as quickly as possible (i.e., whenever for some component in the encoding, enough variable values have been fixed to deduce the value of some other variable) are also called *propagation complete* (see, e.g., Brain et al., 2016). So by adding the clause $carry \vee sum \vee \neg a$ to φ', we make the encoding of the half-adder propagation complete.

2.6.6 Summary

In this section, we discussed various topics on the encoding of practical problems into SAT. We gave insight into components of successful SAT encodings and dealt with the question what makes an encoding good or bad.

In particular, good encodings should make good use of unit propagation. However, whenever a SAT instance would become prohibitively large with an encoding other than a bitwise encoding, the latter may make more sense to be applied. Prestwich (2009, Section 2.2.5, p.80) formulate this as the following rule of thumb:

> *The pairwise encoding is acceptable for fairly small n, while for large n the ladder encoding gives good results with DPLL, and the bitwise encoding gives good results with local search algorithms.*

They also give the following summary of how important it is to reduce the overall number of literals or clauses in an encoding (Prestwich, 2009, Section 2.4.1, last paragraph):

> *In practice, reducing the size of an encoding is no guarantee of performance, no matter how it is measured. Nevertheless, small encodings are worth aiming for because computational results can be very unpredictable, and all else being equal a smaller encoding is preferred.*

Despite these rules of thumbs, an encoding must ultimately help the solver with finding solutions and the only thing that really counts are good results. Depending on the application, some orthogonal approaches can help the solver. For example, adding a few implied clauses can speed up SAT solving dramatically for some problems. It is often not easy to guess in advance which change to an encoding helps the SAT solver, as modern solvers apply multiple heuristics in combination, whose behavior is difficult to predict. A statement by Prestwich (2009, Chapter 2.5) provides a nice conclusion to this line of thought:

> *In short, CNF modelling is an art and we must often proceed by intuition and experimentation.*

Bjork (2009) summarize the difficulty of deriving a good encoding with more emphasis on the craft rather than the art:

> *Improving an encoding is often a process of trial and error.*

What we have not touched in this section is how to encode arithmetic operations on numbers encoded into a SAT instance. Doing so is typically not difficult. For example, for a direct encoding, we would simply encode the value table for $c = a + b$ for some integer variables a, b, and c into a set of three-literal clauses. For the bitwise encoding, we would encode an addition combinational circuit into the SAT instance. However, except for small value domains, such encodings lead to difficult SAT instances, as *bit-blasting* arithmetic operations into boolean formulas removes a lot of structure from the problem. As an alternative, we will discuss computational engines that work with numbers and arithmetic operations as first-class citizens and thus that can make use of the structure of arithmetic operations and number domains in the next chapters.

2.7 More sophisticated applications of SAT solving

To round off the discussion on SAT solvers, we consider a few more applications of SAT solving.

2.7.1 Bounded model checking

Engineering software or hardware is difficult: subtle bugs can hide deep in a design, so code reviews or extensive testing cannot always find them.

In *model checking*, a piece of software or hardware is rigorously tested against a specification: the process considers all possible traces of the system and verifies that the specification holds on all of them. Due to the computational difficulty of this problem in its variations, *bounded model checking* has emerged as a more tractable approximate version of model checking.

In a nutshell, the idea is to fix a number of steps $k \in \mathbb{N}$ and to check whether a piece of software or hardware can behave incorrectly during the first k steps of its execution. To check this, we encode a k-long *trace* of the system as a SAT instance and add clauses that require the trace to represent an illegal computation.

For the sake of simplicity, let us consider a sequential boolean circuit as system model in the following. It is defined by a set of state propositions P, a set of input propositions I, a transition relation $\delta \subseteq 2^P \times 2^I \to 2^P$, and an initial state $S_{init} : P \to \mathbb{B}$. Without loss of generality, we assume that its output signals O are part of the state propositions P.

We can represent δ by a set of CNF clauses $E(S, I, T, S')$ over four sets of variables:

- S and S' are two copies of the state variables, i.e., we have $|S| = |S'| = |P|$
- I is a set of input propositions, and
- T is a set of additional propositions used for the Tseitin encoding of the circuit's behavior.

To represent a k-step long computation of the system, we use multiple copies of E in a SAT instance:

$$\begin{aligned}\varphi_k = {} & \mathit{Init}(S_0) \wedge E(S_0, I_0, T_0, S_1) \wedge \ldots \wedge E(S_{k-2}, I_{k-2}, T_{k-2}, S_{k-1}) \\ & \wedge E(S_{k-1}, I_{k-1}, T_{k-1}, S_k) \wedge \mathit{Error}(S_k)\end{aligned}$$

In this formula, $S_0, \ldots, S_k$ are all copies of P and $I_0, \ldots, I_{k-1}$ are copies of I. Furthermore, $T_0, \ldots, T_{k-1}$ are copies of T. In addition to the copies of the $E(\ldots)$ clauses, $\mathit{Init}(S_0)$ refers to a set of clauses that encode that S_0 represents the initial state S_{init}, and $\mathit{Error}(S_k)$ encodes that the state represented by the values of the variables S_k is an *error state*, i.e., a state that we know to be reachable only by a computation that violates the system's specification.[5]

If a SAT solver finds a satisfying assignment to φ_k, then we can read off a trace of the system that violates the specification from the values to the variable sets S_0, I_0, S_1, I_1, ..., S_k. In fact, the values of $I_0, I_1, \ldots$ represent the stream of input signal valuations that can be fed to the circuit to trigger its erroneous behavior. Likewise, if the SAT solver does not find a satisfying assignment, then we know that there does not exist a system trace of length k that violates the specification.

Bounded model checking is particularly helpful to find subtle bugs that can be reached after few steps. The unsatisfiability of φ_k for some value of k does not imply that there does not exist an erroneous trace of length $k + 1$, however. Thus, in the way that we presented here, bounded model checking cannot prove the absence of errors in a design. Nevertheless, there exist techniques to use bounded model checking for proving the *unreachability* of error states of a circuit, which are however beyond the scope of this course.

Note that when performing bounded model checking, it makes sense to optimize $E(S, I, T, S')$ prior to SAT solving, as this set of clauses is instantiated for many copies of S, I, T, and S'.

[5]For the sake of simplicity, we assumed here that the system's specification is very simple and expressible by a set of *forbidden states*. However, more complex properties can also be encoded in bounded model checking, but this is beyond the scope of this course.

2.7.2 Learning boolean functions

Literature: *Knuth, 2015*

An interesting application of SAT solving is to *learn a boolean function* (not to be confused with *clause learning*, which is used by a SAT solver to speed up the solution process). The problem comes in many variants that have in common that the task is to compute a boolean function $f : \mathbb{B}^n \to \mathbb{B}$ that is consistent with some relatively unstructured input. Knuth (2015) discusses the SAT encoding of a variant of the problem where the input consists of a set of *positive examples* and *negative examples*.

Definition 7. *Let $n \in \mathbb{N}$ be a number of dimensions, $P \subseteq \mathbb{B}^n$ be a list of positive examples, $N \subseteq \mathbb{B}^n$ be a list of negative examples, and k be a number of minterms.*

*The k-*minterm DNF learning problem *for n, P, and N is to compute a boolean formula in disjunctive normal form with k minterms that represents a boolean function $f : \mathbb{B}^n \to \mathbb{B}$ such that for all $\vec{x} \in P$, we have $f(\vec{x}) =$* **true**, *and for all $\vec{x} \in N$, we have $f(\vec{x}) =$* **false**, *or to determine that no such boolean formula exists.*

Learning allows us to uncover the hidden structure of a *black box*, i.e., a memoryless process that obtains n input bits, produces a single-bit outcome, and whose actual definition is unknown to us. By probing the black box with some input vectors, we obtain a partial model of the black box's behavior. If we conjecture that the box has a small DNF representation, we can compute a candidate DNF and compare it against our expectation of how the box operates.

Let $x_1, \ldots, x_n$ be boolean variables that encode an input vector to f. In order to encode the k-minterm DNF learning problem to SAT, we use $2kn + |P|k$ variables in the SAT instance. In particular, we have:

- the variables $\{p_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k}$ to encode that minterm j contains the literal x_i.
- the variables $\{q_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k}$ to encode that minterm j contains the literal $\neg x_i$. and
- the variables $\{z_{j,l}\}_{1 \leqslant j \leqslant k, 1 \leqslant l \leqslant |P|}$ to encode that the lth positive example is covered by minterm j.

As clauses, we use

- the constraints
$$\left(\bigvee_{1 \leqslant i \leqslant n, v_i = \textbf{false}} p_{i,j} \right) \vee \left(\bigvee_{1 \leqslant i \leqslant n, v_i = \textbf{true}} q_{i,j} \right)$$
for every $(v_1, \ldots, v_n) \in N$ and $j \in \{1, \ldots, k\}$ that enforce that none of the minterms in the the DNF formula to be computed captures some negative example,
- the constraints
$$z_{1,l} \vee \ldots \vee z_{k,l}$$
for every $1 \leqslant l \leqslant |P|$ that encode that all positive examples must have some minterm that captures them, and
- the constraints
$$\bigwedge_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k, 1 \leqslant l \leqslant |P|, v_{i,l} = \textbf{true}} (\neg z_{j,l} \vee \neg q_{i,j})$$
and
$$\bigwedge_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k, 1 \leqslant l \leqslant |P|, v_{i,l} = \textbf{false}} (\neg z_{j,l} \vee \neg p_{i,j})$$
for $P = \{(v_{1,1}, \ldots, v_{n,1}), \ldots, (v_{1,|P|}, \ldots, v_{n,|P|})\}$ for encoding that the variables $\{z_{j,l}\}_{1 \leqslant j \leqslant k, 1 \leqslant l \leqslant |P|}$ represent which minterms in the DNF formula capture the positive examples.

Note that there are no clauses that require that in no minterm, some variable x_i occurs with both polarities. There is no need for such clauses as if the SAT solver finds such a solution, then the respective minterm is equivalent to **false** and can be removed from the solution. Yet, it may make sense to add such clauses as they only have two literals each so that they make the search more efficient by providing opportunities for unit propagation.

2.7.3 Planning

Planning is a classical problem in artificial intelligence. In a nutshell, the problem is concerned with transforming the state of a world into another state in a step-by-step fashion. A sequence of operations that does so is called a *plan*, where each operation modifies a part of the state.

Planning comes in a variety of forms that share their main ideas. We consider the following variant here.

Definition 8. *Let A be a set of boolean atomic propositions. We call a pair $\langle p, e\rangle$ an* operator *if p is a boolean formula over A, and e is a list of effects. The latter are pairs (f, d), where f is a boolean formula over A and d is a literal over A.*

Given a valuation $V : A \to \mathbb{B}$, we say that $\langle p, e\rangle$ can be applied in state V if $A \models p$. We say that $\langle p, e\rangle$ transforms state V into another state $V' : A \to \mathbb{B}$ if

1. *$\langle p, e\rangle$ can be applied in state V,*
2. *for all $(f, d) \in e$, if $V \models f$, then $V' \models d$, and*
3. *for all variables $v \in A$, if for no $(f, d) \in e$, we have that $V \models f$ and d is a literal of v, we have that $V(v) = V'(v)$.*

A planning problem *is a tuple $\langle A, I, O, G\rangle$, where A is a set of propositions, $I : A \to \mathbb{B}$ is an initial state, O is a set of operators, and G is a boolean formula over A. In a planning problem, we search for a* plan, *which is a sequence of elements in O that step-wise transform I into a state V' for which $V' \models G$ holds. The length of a plan is the number of operations in the sequence.*

One of the most interesting applications of planning is probably in logistics.

Example 9. Assume that we have a set of goods X, a set of depots Y, and a set of trucks Z, where the trucks operate on fixed routes. The initial locations of the goods and the trucks are given by functions $I_G : X \to Y$ and $I_T : Z \to Y$. We want to find a plan that relocates the goods in X to target depots defined by some function $T_G : X \to Y$.

We can model the scenario by three sets of boolean variables. A variable $v_{x,y}$ is meant to be **true** if good $x \in X$ is in depot $y \in Y$. Likewise, a variable $v'_{z,y}$ is **true** if truck z is at depot y, and a variable $v''_{x,z}$ is **true** if good x is on truck z. For all $x \in X$, $y \in Y$, and $z \in Z$, we have the following operations:

- *Loading a good x onto truck z:* These operations have as preconditions that for some $y \in Y$, both $v_{x,y}$ and $v'_{z,y}$ are **true**, and for no $z' \in Z$, we have that $v''_{x,z'}$ is **true**. As an effect, this action sets $v''_{x,z}$ to **true**.
- *Unloading of a good x for a truck z:* Similar to the previous case
- *Truck z proceeding from depot y to another depot y':* Here, the only condition is that $v'_{z,y}$ must hold. Unconditional effects are that after the transition, $v'_{z,y}$ is **false** and $v'_{z,y'}$ is **true**. For every good $x \in X$, there are conditional effects $(v''_{x,z}, \neg v_{x,y})$ and $(v''_{x,z}, v_{x,y'})$.

Shorter plans are preferred solutions to this planning problem as they require fewer loading/unloading operations and fewer truck trips in order to be executed. ★

Planning and bounded model checking are very similar problems. In both cases, a solution to the problem at hand (a plan or a trace) consists of several steps, and the encoding into SAT is very similar: we restrict our search to plans or traces of a fixed maximum length, allocate sets of variables for each state in the plan or the trace, and replicate the same constraints that encode that a step is valid many times. It is not surprising that planning can be seen as a model checking problem by defining the goal of planning to reach an "error". Determining a trace to the error then corresponds to finding a plan to a goal state.

Consequently, both problems also have a similar complexity. The planning problem defined above is PSPACE-complete. The question whether there exists an error trace for the problem from Section 2.7.1 for some arbitrary $k \in \mathbb{N}$ is also PSPACE-complete. The introduction on a bound on the trace or plan length thus allows us to tackle a PSPACE-complete problem using a decision engine for an NP-complete problem.

There is however also a major difference between model checking (of hardware) and planning: in the latter case, it is quite common that each operator only modifies a small part of the state. This observation gives rise to the following idea: we can analyze which operators do not influence each other and can thus be executed in parallel. This allows us to encode a modified planning problem in which certain operators may be executed in parallel, which in turn allows to find shorter plans. If two operators are concerned with disjoint sets of variables in their conditions and effects, then they do not interfere with each other and can be executed in any order. Then, a plan with parallel operator applications can be transformed into a sequential plan by locally ordering the operations that can be executed in parallel in an arbitrary way.

It has been observed that parallelizing operations is beneficial for the performance of planning. The performance of SAT solving for planning is connected to the overall number of steps needed, and hence minimizing this number helps to obtain results quickly.

It should be noted that there are relaxed notions of parallelizability of operators, which often allow the plans have an even smaller number of states. The interested reader is referred to Rintanen (2009) for more details.

2.8 Debugging SAT encodings

Deferring the solution of a computational problem to a SAT solver requires an encoding of the problem as a SAT instance. As always in computing, it is easy to introduce errors in the process. In classical programming, one can execute a program in a step-by-step fashion in order to find the program line at which things go wrong. As a SAT instance is not executed, doing the same on a SAT encoding is not possible. Yet, there are other ways to debug a SAT instance, which we want to discuss in this section by means of an example. The example is based on `Python` code, but should also be understandable by readers with knowledge of other programming languages.

In the example application, we want to plan a *Halloween party* with 10 guests. There are a couple of aspects of the party that need to be planned. We have to pick one item of food to be served, one style of music to be played, one type of drink to be served (in addition to non-alcoholic drinks), and their favorite Halloween decoration. For each of the guests, we know their favorite food, their favorite style of music, their favorite drink, and their favorite decoration. Since we want them to all feel at home at the party, they should all get one of their favorites, but it does not matter which one. Since only one type of food can be served, one type of drink can be served, one style of music can be played, and different decoration styles do not mix well, we have to plan well in order to get everyone at least one of their favorites.

We start with some Python code that declares and initializes the data structures for the lists of guests and their favorites.

```
#!/usr/bin/env python2
#================================================================
# The halloween party planner
#================================================================
import os,sys

guests = ["Carol","Carl","Marc","Josephine","Max","Cary","William",
     "Lindsey","Frank","Judy"]

aspects = [
   ("Food",{
     "Carol":     "Lasagne",
     "Carl":      "Pizza",
     "Marc":      "Stir Fry",
     "Josephine": "Turkey",
     "Max":       "Spaghetti",
     "Cary":      "Pizza",
     "William":   "Spaghetti",
     "Frank":     "Lasagne",
     "Judy":      "Scrambled Eggs"
   }),

   ("Decoration",{
     "Carol":     "Skeletons",
     "Carl":      "Ghosts",
     "Marc":      "Spiders",
     "Josephine": "Ghosts",
     "Max":       "Zombies",
     "Cary":      "Ghosts",
     "William":   "Spiders",
     "Lindsey":   "Skeletons",
     "Frank":     "Vampires",
     "Judy":      "Werewolves"
   }),

   ("Drinks",{
     "Carol":     "Mojitos",
     "Carl":      "Long Island Ice Tea",
     "Marc":      "Bloody Mary",
     "Josephine": "White Russian",
     "Max":       "Mojitos",
     "Cary":      "Mojitos",
     "William":   "Tequila Sunrise",
     "Lindsey":   "Bloody Mary",
     "Frank":     "Long Island Ice Tea",
     "Judy":      "White Russian"
   }),

   ("Music",{
     "Carol":     "Country",
     "Carl":      "Alternative",
     "Marc":      "Classic Rock",
     "Josephine": "Hip-Hop",
     "Max":       "Jazz",
     "Cary":      "Alternative",
     "William":   "Jazz",
     "Lindsey":   "Irish Folk Music",
     "Frank":     "Reggae",
     "Judy":      "Reggae"
   })
 ]
```

The variable `aspects` contains a list of aspects, each consisting of a party aspect name, and a mapping from guests to their favorite choice for the aspect. It can be seen that there is some overlap. For example, both Carol's and Frank's favorite food is Lasagne.

The program should write the SAT instance to disk, including the correct DIMACS header. During the run of the program, we want to successively write the SAT instance to disk, but still want to write the header last. In order to make sure that we reserve enough space in the file, we apply a trick: we write a first line to the SAT instance file that is actually too long. The rest of the line then later becomes a comment. We append the following block to the program to open the SAT instance file and to initialize variables that keep track of the number of variables and clauses so far:

```
#==========================================================
# Open instance file. Leave some space for the first line.
#==========================================================
outFile = open("planner.cnf","w")
outFile.write("-----------------------\n")
varsSoFar = 0
clausesSoFar = 0
```

The following code is then put at the end of the program to finish the CNF instance by writing the header.

```
#==========================================================
# Finalize encoding
#==========================================================
assert varsSoFar <= 10000
assert clausesSoFar <= 10000
outFile.seek(0)
outFile.write("p cnf "+str(varsSoFar)+" "+str(clausesSoFar)+"\nc ")
outFile.close()
```

The `assert` statements ensure that the number of variables or clauses does not become too large for the space reserved at the beginning of the file to be sufficient. Executing the program without adding any further code in between these blocks gives us the SAT instance

```
p cnf 0 0
c -----------
```

The SAT instance is satisfiable, as there is an assignment to all 0 variables that satisfies all clauses, namely the empty assignment.

To add functionality to the program, we start by reserving variables for encoding which party aspects have been chosen in which way. We allocate one variable for every possible favorite of all guests. It does not make sense to choose the food, drinks, decoration, or music in a way that nobody likes it, as we can then just as well pick one choice that at least someone likes. So we can assume without loss of generality that only food, drinks, decoration, or music that appears somewhere in `aspects` is selected. The following code computes a mapping between choices and SAT variables:

```
#==========================================================
# Define encoding for the favorites
#==========================================================
favouriteToVarMapper = {}
for guest in guests:
    for (aspect,favourites) in aspects:
        for (a,b) in favourites.iteritems():
            if not b in favouriteToVarMapper:
                varsSoFar += 1
                favouriteToVarMapper[b] = varsSoFar
                print >>sys.stderr, "Encoding",b,"into variable",\
                                    varsSoFar
```

The code stores the variable assignments for all aspects into a single map, which is called `favouriteToVarMapper`. This is a little bit unclean, as it disallows having possible choices in different aspects with the same name. So we could, for example, not have `Gothic` both as a decoration choice and as music style. This decision however makes the code to follow easier, which is why it has been made anyway. The code also prints additional output (to the *error stream*), namely which variable encodes which object selection. This information is needed to interpret the SAT solver's output (which for reasons of conciceness we do manually). Using this the variable assignment, we can now add clauses that enforce that all party guests get one of their favorites:

```
#==========================================================
# Define that everybody gets one of their favorites
#==========================================================
for guest in guests:
    for (aspect,favourites) in aspects:
        print >>outFile,favouriteToVarMapper[favourites[guest]],
    print >>outFile,"0"
    clausesSoFar += 1
```

A little Python trick is used in this code: a `print` statement ending with a `,` does not generate a new line, so that for every execution of the outer loop of that code block, only a single clause is added to the output file.

We can already execute the code at this point to see if the generated SAT instance looks as expected. Running the program yields the following output:

```
Encoding Spaghetti into variable 1
Encoding Scrambled Eggs into variable 2
Encoding Pizza into variable 3
Encoding General Tso's Chicken into variable 4
Encoding Stir Fry into variable 5
Encoding Turkey into variable 6
Encoding Lasagne into variable 7
Encoding Spiders into variable 8
Encoding Werewolves into variable 9
Encoding Ghosts into variable 10
Encoding Skeletons into variable 11
Encoding Zombies into variable 12
Encoding Vampires into variable 13
Encoding Tequila Sunrise into variable 14
Encoding White Russian into variable 15
Encoding Mojitos into variable 16
Encoding Bloody Mary into variable 17
Encoding Long Island Ice Tea into variable 18
Encoding Jazz into variable 19
Encoding Reggae into variable 20
Encoding Alternative into variable 21
Encoding Irish Folk Music into variable 22
Encoding Classic Rock into variable 23
Encoding Hip-Hop into variable 24
Encoding Country into variable 25
```

The generated SAT instance looks as follows:

```
p cnf 25 10
c ---------
7 11 16 25 0
3 10 18 21 0
5 8 17 23 0
6 10 15 24 0
1 12 16 19 0
3 10 16 21 0
1 8 14 19 0
4 11 17 22 0
7 13 18 20 0
2 9 15 20 0
```

The instance looks as expected: we have 10 clauses, one for each guest, and they span all 25 variables. Running the SAT solver `picosat` on the generated SAT instance yields the following assignment:

```
s SATISFIABLE
v 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 0
```

So the SAT solver just sets the values of all variables to **true**, which is correct, as we did not encode yet that there can only be one food item selected, one music style, one type of drinks, and one decoration. To do so, we have to select one of the encoding options from Section 2.6.3, which deal with this exact question. We want to use a variant of the *ladder encoding* here. For every aspect, we have to allocate a set of ladder variables and then add the clauses that perform the ladder encoding. We add the following chunk of code:

```
#==========================================================
# For every aspect, there can only be one choice
# ---> Do the ladder encoding!
#==========================================================
for (aspect,favourites) in aspects:

    # Get a *List* of all possible choices
    choices = set()
    for (a,b) in favourites.iteritems():
        choices.add(b)
    choices = list(choices)

    ladderVars = [varsSoFar+i+1 for i in xrange(0,len(choices))]
    varsSoFar += len(choices)
    print >>sys.stderr, "Ladder variables for aspect",\
                        aspect,":",choices,":",ladderVars

    # Ladder clauses, part 1: The first ladder element must
    # reflect the choice of the first choice element
    print >>outFile,-1*favouriteToVarMapper[choices[0]],\
                    ladderVars[0],0
    clausesSoFar += 1

    # Ladder clauses, part 2: The ladder never goes down
    for i in xrange(0,len(ladderVars)-1):
        print >>outFile,-1*ladderVars[i],ladderVars[i+1],0
        clausesSoFar += 1

    # Ladder clauses, part 3: The ladder has a step if an object has
    # been selected
    for i in xrange(0,len(ladderVars)-1):
        print >>outFile,ladderVars[i],-1*favouriteToVarMapper[\
                        choices[i]],ladderVars[i+1],0
        clausesSoFar += 1

    # Ladder clauses, part 4: The ladder has a step if an
    # object has not been selected
    for i in xrange(0,len(ladderVars)-1):
        print >>outFile,ladderVars[i],favouriteToVarMapper[\
                        choices[i]],-1*ladderVars[i+1],0
        clausesSoFar += 1

    # Ladder clauses, part 5: At the end, the ladder is up!
    print >>outFile,ladderVars[-1],0
    clausesSoFar += 1
```

The code that is executed first for each aspect needs some explanation. It collects all possible favorites for the aspect and stores them in a set in order to remove duplicates, and then translates the set into a list in order to give the elements an order. Then, it allocates ladder variables starting from the current value of `varsSoFar` and prints the ladder variables together with the order of the choices for the aspect. Running the program again yields the additional variable information:

```
Ladder variables for aspect Food : ['Turkey', 'Lasagne', 'Scrambled
    Eggs', 'Stir Fry', "General Tso's Chicken", 'Spaghetti',
    'Pizza'] : [26, 27, 28, 29, 30, 31, 32]
Ladder variables for aspect Decoration : ['Vampires', 'Werewolves',
    'Spiders', 'Zombies', 'Ghosts', 'Skeletons'] : [33, 34, 35, 36,
    37, 38]
Ladder variables for aspect Drinks : ['Mojitos', 'White Russian',
    'Tequila Sunrise', 'Bloody Mary', 'Long Island Ice Tea'] : [39,
    40, 41, 42, 43]
Ladder variables for aspect Music : ['Reggae', 'Country', 'Jazz',
    'Alternative', 'Irish Folk Music', 'Hip-Hop', 'Classic Rock'] :
    [44, 45, 46, 47, 48, 49, 50]
```

Since a first runnable version of the program that should have all constraints in place is now ready, we can start the testing and debugging process.

2.8.1 Debugging: Act I

Running the SAT solver `picosat` on the SAT instance generated by the program yields:

```
s SATISFIABLE
v 1 2 3 4 5 -6 7 8 9 10 11 12 -13 14 15 -16 17 18 19 -20 21 22 23
v 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
v 45 46 47 48 49 50 0
```

We can see that the SAT instance is buggy, as out of the variables 1-7, which all represent food items, all of them except for one have **true** values. So too much food has been selected. Looking at the ladder for food (variables 26-32) yields that the ladder "goes up" at its first element, i.e., the ladder counts the number of food items correctly. However, there seems to be nothing that stops the SAT solving from selecting more than one item for an aspect.

Looking at the code again yields the answer: there are actually no constraints that enforce that for every i, if the ladder variable i has a **true** value, then the items from position $i+1$ onwards of the list are not selected. Using the fact that in the ladder encoding, a ladder always "stays up" once it "went up", we can encoding this by adding a few lines to the program's loop in which the code iterates over the aspects for generating the ladder encoding clauses:

```
    # Ladder clauses, part 6: If the ladder is up, no more choices!
    for i in xrange(0,len(ladderVars)-1):
        print >>outFile,-1*ladderVars[i],-1*favouriteToVarMapper[\
                        choices[i+1]],0
        clausesSoFar += 1
```

2.8.2 Debugging: Act II

Running the program again and then also the SAT solver, we get the following variable valuation:

```
s SATISFIABLE
v -1 -2 -3 -4 -5 -6 7 8 -9 -10 -11 12 -13 -14 15 -16 -17 -18 -19
v -20 21 22 -23 -24 -25 -26 -27 28 29 30 31 32 -33 -34 -35 36 37
v 38 -39 -40 41 42 43 -44 -45 -46 -47 48 49 50 0
```

We see that both the variables 8 and 12, which stand for spiders and zombies, have **true** values. Spiders and zombies do not mix well, so we have to check what the problem could be. We need to check the values of the ladder variables for this. The ladder variables 33-38 that are responsible for the decoration look fine at first: the phase transition happens between variables 35 and 36. So either the phase transition happens at the wrong point, or the clauses generated by the code that we added above are not correct. Variable 35 is for spiders, while variable 36 is for zombies.

Now the question is if the phase transition is really at the correct point. We want that the ith ladder variable represents whether one of the first i choices has been made. This means that when spiders are selected, variable 35 having a **false** value is incorrect. Looking at the ladder clauses, we see that the problem lies in part 3 of the clauses: in order to evaluate whether the $(i+1)$th ladder variable needs to be **true**, these clauses are concerned with whether the ith item in the list of aspect alternatives has been made. However, the $(i+1)$th choice should be looked at instead.

Fixing this issue requires changing this part of the code as follows:

```
# Ladder clauses, part 3: The ladder has a step if an object has
# been selected
for i in xrange(0,len(ladderVars)-1):
    print >>outFile,ladderVars[i],-1*favouriteToVarMapper[\
                    choices[i+1]],ladderVars[i+1],0
    clausesSoFar += 1
```

2.8.3 Debugging: Act III

Again, we run the program and then the SAT solver. This time, running `picosat` yields "`s UNSATISFIABLE`" as answer. While it could be correct that there is no way to satisfy all guests at the same time, we want to look into the unrealizability result closer to check if we find remaining bugs in the code.

To do so, we can compute a *minimal unsatisfiable core* of the SAT instance. This is a subset of all clauses that is still unsatisfiable such that removing any single clause from the subset yields a satisfiable SAT instance. In a sense, an unsatisfiable core represents a concise reason for the unsatisfiability of a SAT instance. The SAT solver `picosat` comes with the tool `picomus` to compute unsatisfiable cores. We run "`picomus planner.cnf output.cnf`" in order to let `picomus` compute the minimal unsatisfiable core and dump it into the file `output.cnf`.

The generated unsatisfiable core file looks as follows:

```
p cnf 50 23
1 12 16 19 0
1 8 14 19 0
-29 30 0
29 -4 30 0
30 -1 31 0
30 4 -31 0
-30 -1 0
-33 34 0
33 -9 34 0
```

```
34 -8 35 0
35 -12 36 0
34 9 -35 0
35 8 -36 0
-34 -8 0
-35 -12 0
-16 39 0
-39 40 0
-40 -14 0
-44 45 0
44 -25 45 0
45 -19 46 0
45 25 -46 0
-45 -19 0
```

Most of the clauses in the minimal unsatisfiable core are part of the ladder encoding. There are actually only two clauses that represent that two guests must get one of their favorites, and both of them have `1` (which stands for choosing spaghetti) as a literal. This is strange, as choosing Spaghetti as food would satisfy both of these guests at the same time, which should be possible.

To investigate the problem further, we want to check if there is really no way in which spaghetti could be part of a solution. To do that, we remove all the clauses from `planner.cnf` that encode that everyone gets one of their favorites, and insert the clause `1 0` instead that only asks for spaghetti to be chosen as food items. After reducing the number of clauses in the file header by 9 to 93, we save the file and run `picosat` again. The instance stays unsatisfiable, and computing a minimal unsatisfiable core with `picomus` yields:

```
p cnf 50 6
1 0
-29 30 0
29 -4 30 0
30 -1 31 0
30 4 -31 0
-30 -1 0
```

So this time, the minimal unsatisfiable core is even smaller. It shows the clauses that enforce spaghetti not to be taken together with the clause that enforces spaghetti to be selected. None of the clauses individually force spaghetti not to be taken - otherwise the unsatisfiable core would not be minimal. This makes it difficult to *blame* one of the clauses.

However, what we can do is to write down an assignment to the variables that we *think* should allow to satisfy the SAT instances. We can then see which clause or clauses prevents or prevent this assignment from being correct. Here, this assignment consists of the literals `1 -26 -27 -28 -29 -30 31 32`, as 1 makes spaghetti selected, and the other literals represent the ladder that has a step at exactly the position of the spaghetti (recall that we can see from the output of the program that spaghetti are the second to last item in the list used for the ordering the elements in the ladder encoding).

When comparing this (partial) assignment with the clauses, we see that it satisfies all clauses except for `30 -1 31`, so this must be the culprit. The clause has been produced by the code for part 4 of the ladder encoding (which we could have determined by letting the program write comment lines into the SAT instance that label the clauses by the line in which they are generated, running the program, and then searching for the labelling of this clause). In the previous debugging step, we had to add 1 to the indices of the choices. It looks like that we forgot to do the same as well for the clauses produced in part 4 of the ladder encoding program code. So we change this code to:

```
    # Ladder clauses, part 4: The ladder does not have a step if an
    # object has not been selected
    for i in xrange(0,len(ladderVars)-1):
        print >>outFile,ladderVars[i],favouriteToVarMapper[\
                        choices[i+1]],-1*ladderVars[i+1],0
        clausesSoFar += 1
```

2.8.4 Debugging: Final act

Running the program and `picosat` again yields the result that the SAT instance is still unsatisfiable. As before, we do not know if this is the result of a bug in the program or the actual result to our problem. So we compute the unsatifiable core again.

This time, the unsatisfiable core has 57 clauses and looks as follows:

```
p cnf 50 57
7 11 16 25 0
6 10 15 24 0
1 12 16 19 0
3 10 16 21 0
1 8 14 19 0
4 11 17 22 0
7 13 18 20 0
2 9 15 20 0
-6 26 0
-26 27 0
-27 28 0
-28 29 0
-29 30 0
-30 31 0
26 -7 27 0
[...]
```

So eight clauses that require guests to get one of their favorites are necessary in order to get an unsatisfiable SAT instance. This means that any subset of seven of the eight guests represented by the clauses can be accommodated, provided that the encoding is correct. Note that it may be possible to accommodate nine of the ten guests, as there may be multiple minimal unsatisfiable cores, and the unsat core generator tool may have chosen to give preference to removing ladder clauses from the SAT instance instead of removing the clauses that enforce that the guests get one of their favorites.

What we can do now to debug the SAT instance is to remove some party guests from our consideration such that we get a satisfiable instance and check if the assignment that we get from the SAT solver makes sense.

We take the first seven clauses of the minimal unsatisfiable core and replace the 10 guest favorite definition clauses in the original SAT instance by them. Running `picosat` again yields the following variable assignment:

```
s SATISFIABLE
v -1 -2 -3 -4 -5 -6 7 -8 -9 10 -11 -12 -13 -14 -15 -16 17 -18 19
v -20 -21 -22 -23 -24 -25 -26 27 28 29 30 31 32 -33 -34 -35 -36
v 37 38 -39 -40 -41 42 43 -44 -45 46 47 48 49 50 0
```

Inspecting it manually yields that there is only one food item, one decoration, one drink, and one music type selected. These are:

- Lasagne, as variable `7` has a **true** value,
- Ghosts, as variable `10` has a **true** value,

- Bloody Mary, as variable 17 has a **true** value, and
- Jazz, as variable 19 has a **true** value.

Looking at the guests' preferences, we see that lasagne is Carol's and Frank's favorite food, ghosts are Carl's, Josephine's and Cary's favorite decoration, Bloody Mary is Marc's and Lindsay's favourite drink, and Jazz is liked by Max and William. So we even found a solution that makes 9 out of the 10 guests happy (while enforcing only for seven guests that they need to get one of the favorites).

Note that this does not contradict the fact that in the last unsatisfiable core, we only had clauses for eight guests, as a minimal unsatisfiable core is not necessarily the smallest one (in terms of the number of clauses it contains), especially when counting only a subset of the clauses.

Since the result makes sense and we did not find one more error in the encoding, it is fair to stop debugging at this point. While there may of course still be bugs in the program that we did not find, we at least tested that for a slightly relaxed version of the problem, the computed result is correct.

2.9 Conclusion

In this chapter, we have learned about a first computational engine: SAT solvers. We have seen how they operate (by the DPLL and CDCL principles) and what makes them work well in practice (early conflict detection by unit propagation and clause learning). We have also discussed their limits by dealing with random SAT instances, the pidgeon hole principle and backdoors. The example applications considered show how SAT solvers can be used in practice, and our treatment of encodings for practical problems into SAT sheds light on the answer to the question of how to use them most effectively.

We want to conclude with a few final notes to allow the interested reader a smoother transition into reading the other literature on SAT solving.

We used the Sudoku problem at the beginning of this chapter to motivate the usage of simple rules to augment guessing as a reasoning engine. Most likely, Sudoku and SAT solving evolved seperately, so that there is no historical connection between the two. However, Sudoku solving provided us with a smooth introduction into the topic, which is why it was used here anyway.

As the aim of this lecture is to give a good overview of various computational engines, we covered a couple of sub-topics only very briefly without going into the details. This required to adapt the formalization of some topics. For example, in Section 2.6.1, we gave a very short introduction into symmetry breaking on the basis of solution transformers. Literature on the topic typically uses *symmetry groups* as basis for discussing the topic. However, a treatment of the topic based on symmetry groups is not only more comprehensive, but also much more lengthy, which is why it was not done here. To the best of the author's knowledge, solution transformers are not used in the primary literature on symmetry breaking so far.

It should also be mentioned that there are other approaches to SAT solving that do not use DPLL and CDCL. *Local search* has already been mentioned in a quotation on page 49, but has not been explained so far. In a nutshell, in local search, a candidate assignment to the SAT variables is iteratively refined in order to find a satisfying assignment. In the past, local search was an important concept and observations made on local search algorithms gave rise to new techniques that are also useful for CDCL-style solving.

An even earlier approach to SAT solving is to apply *resolution*. Here, variables are eliminated from the SAT instance step by step. Except for a few counter-examples, resolution is considered to not be competitive with CDCL-style SAT solving, however, and for most classes of problems, local search is also considered to be less suitable than CDCL-style solvers.

Chapter 3

The Ecosystem around SAT

The success of SAT solving has motivated the construction of a few computational engines that use a SAT solver under the hood, but actually solve a problem that is different to SAT. We discuss a few of these engines in this chapter as they are well-suited for some practical applications.

3.1 Pseudo-boolean constraint solving

In Section 2.6.3, we discussed many ways for encoding a non-boolean but finite variable domain into SAT. Typically, this domain is $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$. We refrained from discussing thorougly how we can perform arithmetic operations (such as additions) with these numbers, as this is relatively cumbersome and there are special computational engines that do that themselves. Pseudo-boolean constraint solvers fall into this category of solvers.

Definition 9. *Let $\mathcal{V}$ be a set of variables with domain $\{0,1\}$. We call a set of linear inequalities over $\mathcal{V}$ a* pseudo-boolean constraint system*.*

Example 10. Let $\mathcal{V} = \{a, b, c\}$. The following constraints form a pseudo-boolean constraint system:

$$\begin{aligned} 3 \cdot a + 4 \cdot b - 2 \cdot c &\geqslant 5 \\ -1 \cdot a + 2 \cdot b &\geqslant 1 \\ 5 \cdot c + 5 \cdot b &\leqslant 6 \end{aligned}$$

★

An assignment of $\mathcal{V}$ to the values $\{0,1\}$ that satisfies all constraints in a constraint system is called a *solution* of the system. The process of checking if there exists such as assignment is called *solving* the system.

Definition 9 only allows inequalities in constraint systems. However, every equality can also be expressed as a set of two inequalities. For example,

$$3 \cdot a + 4 \cdot b - 2 \cdot c = 5$$

holds if and only if all of the following constraints are satisfied:

$$\begin{aligned} 3 \cdot a + 4 \cdot b - 2 \cdot c &\geqslant 5 \\ -3 \cdot a - 4 \cdot b + 2 \cdot c &\geqslant -5 \end{aligned}$$

Thus, without loss of generality, we can also use the equality operator in pseudo-boolean constraint systems despite the fact that equality is not covered by Definition 9.

Note that pseudo-boolean constraint solving includes SAT solving as a special case. Every SAT instance in CNF form can be easily translated easily to a Pseudo-boolean constraint system: we use the same set of variables, and for every clause $l_1 \vee l_2 \vee \ldots \vee l_n$, we use a constraint $f(l_1) + f(l_2) + f(l_3) \ldots f(l_n) \geqslant 1$ for $f(\neg v) = (1 - v)$ and $f(v) = v$ for all $v \in \mathcal{V}$. A $\{0,1\}$-assignment to $\mathcal{V}$ that satisfies the constraint system is then a solution to the SAT problem. The constraints built by this translation step can easily be brought into a form in which the constants appear only right of the comparison operator.

Unlike in the SAT case, in pseudo-boolean constraint solving, we can also have constraints that are equivalent to **true** or **false** without this being completely obvious. For example, the clause $4 \cdot a + 2 \cdot b \geqslant 7$

is not satisfiable as a and b have a domain of $\{0,1\}$. Similarly, the clause $-4 \cdot a - 2 \cdot b \geqslant -7$ is trivially satisfied.

While all variables in pseudo-boolean constraint systems are still boolean (if we ignore the semantic difference between $\{\mathbf{false}, \mathbf{true}\}$ and $\{0,1\}$), they allow us to concisely represent constraints over numbers. Assume for example that we have the variables $\mathcal{V} = \{a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_n\}$, where each of $\{a_1, \ldots, a_n\}$, $\{b_1, \ldots, b_n\}$ and $\{c_1, \ldots, c_n\}$ denote a one-hot/direct encoding of a number $\vec{a}$, $\vec{b}$, or $\vec{c}$ (between 1 and n). We can require an assignment to $\mathcal{V}$ satisfy $\vec{a} + \vec{b} = \vec{c}$ and to be a valid direct encoding by the following constraints:

$$\begin{aligned}
1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + \ldots + n \cdot a_n + 1 \cdot b_1 + \ldots + n \cdot b_n - 1 \cdot c_1 - \ldots - n \cdot c_n &= 0 \\
a_1 + a_2 + \ldots + a_n &= 1 \\
b_1 + b_2 + \ldots + b_n &= 1 \\
c_1 + c_2 + \ldots + c_n &= 1
\end{aligned}$$

The last three constraints enforce that the direct encoding is valid, whereas first constraint encodes that $\vec{a} + \vec{b} = \vec{c}$, using the semantics of the direct encoding. It can be seen that we only need four constraints overall. Recall that in SAT, we need $O(n^2)$ many constraints/clauses just for encoding that only one variable in $a_1, \ldots, a_n$ may have a value of **true**. So pseudo-boolean constraints allow us to state some problems in a very compact way.

We can also use a binary encoding for the same purpose. So let $\{a_1, \ldots, a_n\}$, $\{b_1, \ldots, b_n\}$, and $\{c_1, \ldots, c_n\}$ denote binary values, where $\vec{a}$ and $\vec{b}$ have domains of $\{0, \ldots, n_a\}$ and $\{0, \ldots, n_b\}$, respectively. The following constraints then encode that $\vec{a} + \vec{b} = \vec{c}$ and that the values for $\vec{a}$ and $\vec{b}$ are valid:

$$\begin{aligned}
1 \cdot a_1 + 2 \cdot a_2 + 4 \cdot a_3 + \ldots + 2^{n-1} \cdot a_n + 1 \cdot b_1 + \ldots + 2^{n-1} \cdot b_n& \\
-1 \cdot c_1 - 2 \cdot c_2 - 4 \cdot c_3 - \ldots - 2^{n-1} \cdot c_n &= 0 \\
1 \cdot a_1 + 2 \cdot a_2 + \ldots + 2^{n-1} a_n &\leqslant n_a \\
1 \cdot b_1 + 2 \cdot b_2 + \ldots + 2^{n-1} b_n &\leqslant n_b
\end{aligned}$$

Again, we only need very few constraints, and the shape of the first constraint is actually the same as in the direct encoding case.

Let's consider an example application for pseudo-boolean constraint solving.

Example 11. Let a set of tasks $T = \{t_1, \ldots, t_k\}$ be given that are to be performed by a group of workers $W = \{w_1, \ldots, w_n\}$ every week. A function $t : T \to \mathbb{N}$ assigns to each task a duration in hours. We want to find an mapping of the tasks to the workers that ensures that no worker gets assigned work for more than 40 hours a week.

We can model the setting as a pseudo-boolean constraint system with the variables $\{x_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k}$, where x_{ij} is **true** whenever task j is assigned to worker i. The constraints are:

$$\begin{aligned}
\forall 1 \leqslant i \leqslant n : &\qquad \sum_{1 \leqslant j \leqslant k} t(j) x_{ij} \leqslant 40 \\
\forall 1 \leqslant j \leqslant k : &\qquad \sum_{1 \leqslant i \leqslant n} x_{ij} = 1
\end{aligned}$$

The assignment of the tasks to the workers can be read off from a valuation of the x_{ij} variables. ★

The literature on pseudo-boolean constraint system solving describes several methods to do so. Some of them encode the problem into SAT. Section 2.6.4 provided some encodings for n-out-of-k constraints. These can be adapted to encoding a pseudo-boolean constraint if the coefficients in the constraint are small. However, there are also other encodings that are more efficient for numbers with high coefficients. For brevity, we will not discuss them here. It should be noted however that by letting a pseudo-boolean constraint solver perform the encoding, we save the work of implementing them manually. Also, some pseudo-boolean solvers support multiple encodings that can be manually selected in order to check which one works best for an application.

One interesting variant of the pseudo-boolean constraint solving problem is *pseudo-boolean optimization*. Here, a *cost* function is to be given in addition to a pseudo-boolean constraint system. The cost function is a linear function over the variables of the problem, and we search for assignments to the variables that not only satisfy the constraint system, but also induce the minimal possible cost (among the assignments that satisfy all constraints). The following example shows how a cost function can be used.

Example 12. Reconsider Example 11. If some workers negotiated a bonus for performing certain tasks into their work contracts, then it makes sense to try to minimize the overall amount of bonus money spent when computing the assignment of the tasks to the workers. Let $w_{i,j}$ be the bonus of worker i for performing task j (for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant k$). Our cost function is then defined as:

$$c(x_{1,1}, \ldots, x_{n,k}) = w_{0,0} \cdot x_{0,0} + w_{0,1} \cdot x_{0,1} + \ldots + w_{n,k} \cdot x_{n,k}.$$

★

3.1.1 Pseudo-boolean constraint solving in practice

The development of Pseudo-boolean constraint solvers is driven by the competitions that take place regularly. The problem description file format (“pbs”) used therein is the de-facto standard for encoding pseudo-boolean constraint systems.

Files in this format always start with a line of the form

```
* #variable= <X> #constraint= <Y>
```

in which <X> and <Y> are to be replaced by the number of variables and the number of constraints in the problem instance. The header line is then followed by the pseudo-boolean constraints. Each constraint consists of a left-hand side (a list of weighted boolean terms), a comparison operator (only “greater than” (>=) and “equal” (=) operators are allowed, and a right-hand side, which is an integer constant. Lines beginning with a star (*) are comments and can appear anywhere in the files. Each non-empty non-comment line must end with a semicolon (;). Each boolean variable is named with a lowercase 'x' followed by a strictly positive integer number (e.g., x4). The integer number can be considered as a identifier of the variable. The integer numbers cannot be greater than $2^{32} - 1$. Each boolean variable name must be followed by an empty space.

The weight of a variable may contain an arbitrary number of digits. There must be no space between the sign of an integer and its digits (i.e. to weighten variable x4 by -5, we must write -5 x4).

As an example, consider the following constraint system with the three variables $\{x_1, x_2, x_3\}$:

$$\begin{aligned} 1 \cdot x_1 + 2 \cdot x_2 - 1 \cdot x_3 &\geqslant 2 \\ -2 \cdot x_1 + 6 \cdot x_3 &\geqslant 4 \end{aligned}$$

Its encoding in into a pbs file looks as follows:

```
* #variable= 3 #constraint= 2
* Example comment line
+1 x1 +2 x2 -1 x3 >= 2;
* There are 3 variables (x1, x2 and x3) and 2 constraints
-2 x1 +6 x3 >= 4;
```

Feeding the file to the solver clasp (Gebser et al., 2012a), which solves pseudo-boolean constraint systems (among some more sophisticated problems), yields the following result:

```
c clasp version 3.1.4
c Reading from test.pbs
c Solving...
c Answer: 1
v x1 x2 x3
s SATISFIABLE
c
c Models         : 1+
c Calls          : 1
c Time           : 0.000s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
c CPU Time       : 0.000s
```

A problem instance for pseudo-boolean optimization is structured in the same way, except that the first line after the header line of the pbs file contains the description of a linear function to be minimized. Such as line can look as follows:

```
min: 1 x2 -1 x3;
```

The linear function is described in the same way as the constraints. Running `clasp` on the example instance from before with this added line yields the following output:

```
c clasp version 3.1.4
c Reading from test.pbs
c Solving...
o 0
c Answer: 1
v x1 x2 x3
s OPTIMUM FOUND
c
c Models         : 1
c   Optimum      : yes
c Optimization   : 0
c Calls          : 1
c Time           : 0.000s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
c CPU Time       : 0.000s
```

The line starting with s represents what type of result the solver has obtained:

- *"s UNSUPPORTED":* this line is printed if the solver finds that a feature of the (extended) PBS file format is used that the solver does not support.
- *"s SATISFIABLE":* this line indicates that the solver has found a model of the formula.
- *"s OPTIMUM FOUND":* this line is printed when the solver has found a model and it can prove that no other solution is better.
- *"s UNSATISFIABLE":* this line is printed when the solver can prove that the formula has no solution.
- *"s UNKNOWN":* This line is printed if the solver abort without solving the formula, e.g., when a time bound is given to the solver and the bound has been exceeded before finding a model or proving unsatisfiability.

3.2 Alloy

All computational problems that we have considered so far (SAT and pseudo-boolean constraint solving) have a finite number of variables. Thus, there is an upper bound on the complexity of the problems as it suffices to enumerate all assignments to the variables and check for each of them whether they satisfy the constraints.

Sometimes, this simple view is not appropriate. Take for example the planning or bounded model checking problems from Section 2.7.1 and Section 2.7.3. In both cases, we did not know in advance how long the artifacts that we wanted to compute are. In order to encode these problems into SAT, we had to bound the length of a system trace to some number *k*. This allowed us to use a tool for solving an NP-complete problem to tackle an PSPACE-complete problem in an approximate way. In order to obtain completeness, we would need to run the process for all sensible values of *k*, normally starting with a low value of *k*, and gradually increasing it until a trace or plan is found or all sensible values of *k* have been tried.

This raises the question if there are more appropriate formalisms for computational problems whose results (also called *models*) have an a-priori unknown size. Such a formalism would allow us to describe relationship between objects in our model, the components that need to be in the model, and a corresponding computational engine could then unroll the search problem on its own.

The `Alloy` analyzer (Jackson, 2002) builds on these ideas. It is mainly intended to be used to verify communication protocols or software, but its input language is general enough to also capture many puzzles and constraint satisfaction problems. In this section, we discuss a few example `Alloy` specifications to give insight into the capabilities of the analyzer. Due to the richness of `Alloy`'s input language, a full description of the capabilities of the system would exceeds the scope of this course. Thus, we will only discuss the syntax and semantics by means of the examples to be given.

One of the key concepts of `Alloy` is that everything is a relation. Thus, in an `Alloy` specification, we describe relations between objects (which are 0-ary relations). Let us consider the following `Alloy` specification, which is a modification of one of `Alloy`'s example problems.

```
1  module myproblem
2
3  sig Person {
4    father: one Person,
5    mother: one Person
6  }
7
8  assert noSelfFather {
9     no m: Person | m = m.father
10 }
11
12 check noSelfFather
```

The first line describes a *module name*, which we can ignore for the time being. Lines 3 to 6 describe the *signature* of an object. Here, we declare an object type named "Person", and each person has a father and a mother. The `Alloy` syntax allows to treat them as fields as in object-oriented programming, but the two fields can also be seen as relations $\mathit{father} \subseteq \mathit{Person} \times \mathit{Person}$ and $\mathit{mother} \subseteq \mathit{Person} \times \mathit{Person}$. The occurrences of the keyword `one` in the `Alloy` code induce the constraints that every person should have precisely one mother and one father.

The `assert` statement in the code describes an assertion, i.e., a statement that should be true on all models of the specification. The command `check noSelfFather` afterwards instructs `Alloy` to test if the assertion actually holds. The "`|`" character in the command should be read as "such that".

If we run `Alloy` on this specification, we get a model that violates the assertion. It is shown in Figure 3.1. If can be seen that the model in Figure 3.1 violates the `assert` statement, but satisfies all other parts of the specification. The `assert` statement intuitively states that there is no person who is his/her own father. Apparently the specification does not exclude that a person is his own father, and the model shows an example for that.

So let us try something else now. Let us first of all exclude all models from consideration in which a person is his/her own father. We modify the specification from above to the following specification:

```
module myproblem
```

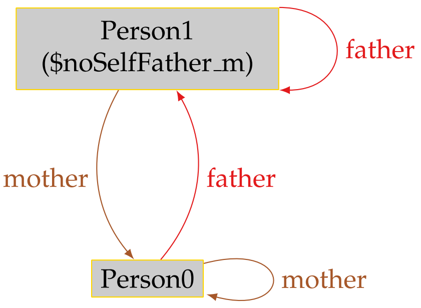


Figure 3.1: A first `Alloy` model.

```
sig Person {
  father: one Person,
  mother: one Person
}

fact noSelfFather {
   no m: Person | m = m.father
}
```

Note that we changed the `assert` statement to a `fact`. Now let us refine the model such that all persons are either women or men. `Alloy` has a special syntax for this:

```
sig Woman extends Person { }
sig Man extends Person { }
```

Again, the syntax is draws motivation from object-oriented programming, where `extends` often refers to inheritance. We now add two more facts that ensure that every person's father is a man, and every person's mother is a woman.

```
fact fatherhood {
   all m: Person | some m2: Man | m.father = m2
}

fact motherhood {
   all m: Person | some m2: Woman | m.mother = m2
}
```

So we state that for every person, there exists some man that is the person's father. Likewise for every person, there exists some woman that is the person's mother.[1]

Now we can ask `Alloy` whether it is possible for some person p to have some person p' both as mother and as father.

```
assert singleParent {
   no m: Person | m.father = m.mother
}

check singleParent
```

[1]For full disclosure, it should be added that there are more elegant ways to express this in `Alloy`. However, to provide a better exposition of the main ideas of `Allow`, we are not using them here.

This time, the solver states that no counter-example to the statement has been found, and it may thus be valid. Note that the "may thus be valid" statement comes from the fact that `Alloy` only tries to find *small and finite* models that conform to a specification. The developers of `Alloy` proposed the line of reasoning that most specifications (in practice) have small models, so it suffices to search for these. Consequently, the tool cannot prove that a specification does not have a model.

Note that we can ask `Alloy` to check for models of a certain size. This is done by replacing the `check` line of the model above by the following line:

```
check singleParent for <number>
```

Internally, `Alloy` makes use of the `Kodkod` model finder (Torlak and Jackson, 2007), which translates the problem of searching for a model of a certain size to SAT. The `Kodkod` model finder uses some advanced techniques such as *auto-compacting circuits* and a translation algorithm based on *sparse matrices*. Also, it features symmetry breaking based on *symmetry detection* in the specification.

For the `singleParent` check, `Alloy 4.2` produces SAT instances with, e.g., 228 variables (for bound 2), 489 variables (for bound 3), or 3415 variables (for bound 10). All of them are solved on a moderately modern 1.6 GHz machine with the Java-based SAT solver that is bundled with `Alloy 4.2` in under 0.2 seconds. The short computation time shows that `Alloy` and `Kodkod` encode the problem in a quite efficient way.

3.2.1 Alloy for verification

What makes Alloy so interesting for the purpose of verification is that it allows us to reason about state changes in systems. We can model the evolution of a system over time as a set of states that are connected by a transition relation. Consider the following example specification:

```
module myproblem

sig Data {}

sig State {
    currentData: one Data,
    nextState: lone State
}

fact linearOrder { no s:State | s in s.^nextState }

fact dataChange { no s: State |
    s.currentData = s.nextState.currentData }

assert noRepetition {
    no s : State | s.currentData in s.^nextState.currentData
}

check noRepetition
```

It models that at every point in time, some program is in a state in which `currentData` points to precisely one data item. Every state may be succeeded by a successor state, but does not need to have a successor (the keyword `lone` represents that a field may have zero or one element in relation).

We restrict the models to those in which the states are ordered linearly. The fact `linearOrder` ensures this using the ^ operator for non-reflexive transitive closure. The fact `dataChange` states that during the execution of the system, the `currentData` element of the state always changes during transitions of the system.

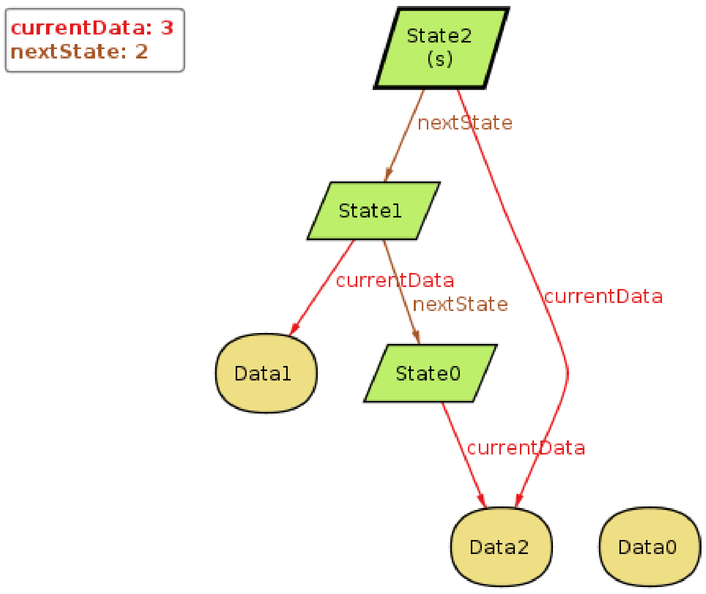


Figure 3.2: An `Alloy` model. This time, the figure is shown as drawn in the graphical user interface of `Alloy`.

Then we asked the question if it is possible that some data element occurs twice along an execution of the system. `Alloy` finds a model relatively quickly, which we show in Figure 3.2.

3.3 Constraint programming

In *constraint programming*, which is also called *constraint satisfaction problem (CSP) solving*, we are given a finite set of variables, their domains, and a set of constraints over these variables. A *constraint* solver then has the task of assigning values to the variables that satisfy all of the constraints.

The main difference of this problem formulation to SAT and pseudo-boolean constraint solving is that it is much more general: not only do we allow non-boolean domains for the variables, but we also did not state how the constraints look like. While this makes the problem definition quite incomplete, the reason for this choice is rather simple: there are many tools for solving constraint programming problems and they have different sets of supported variable types and operators, so that a unique formal and complete definition cannot be given.

There are problems that are more naturally modeled as constraint programming problems and for which the built-in decision procedures of constraint programming tools provide an advantage over plain SAT solving. Consider the following constraint satisfaction problem, which is in the *MiniZinc* input format (Marriott and Stuckey, 2014):

```
include "alldifferent.mzn";

var 0..9: a;
var 0..9: b;
var 0..9: c;
var 0..9: d;
var 0..9: e;
var 0..9: f;
var 0..9: g;
var 0..9: h;
var 0..9: i;
var 0..9: j;
var 0..9: k;

constraint alldifferent([ a,b,c,d,e,f,g,h,i,j,k ]) ;

solve maximize c;
```

In this problem instance, we introduce 11 variables with domains of size 10 each. A constraint states that all the 11 variables must have different values. Finally, we request the value of variable `c` to be maximized.

This input instance encodes the pidgeon hole principle that we studied for SAT solvers in Chapter 2.5.2. Obviously, the problem instance does not have a solution, but because of the fact that we did not encode the `alldifferent` constraint into SAT, the solver has the chance to perform the analysis in a more clever way.

The MiniZinc input format is normally not used directly, but rather translated to a different input format (called *FlatZinc*) that can then be used by solvers. The translation process is solver-dependent and in the process, *MiniZinc* features that are not supported by target solvers are compiled away.

Using for example the solver `JaCoP` (Kuchcinski and Szymanek, 2010), the resulting FlatZinc instance is found to be unsatisfiable in 9.5 seconds on the same computer as used for the run of `Minisat` in Figure 2.20 (on page 35). When we look at the FlatZinc file, we can observe that it has the following form:

```
array [1..2] of int: X_INTRODUCED_1 = [1,-1];
var 0..9: a:: output_var;
var 0..9: b:: output_var;
var 0..9: c:: output_var;
var 0..9: d:: output_var;
var 0..9: e:: output_var;
var 0..9: f:: output_var;
var 0..9: g:: output_var;
var 0..9: h:: output_var;
var 0..9: i:: output_var;
var 0..9: j:: output_var;
var 0..9: k:: output_var;
constraint int_lin_ne(X_INTRODUCED_1,[a,b],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,c],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,d],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,e],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,f],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,g],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,h],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,i],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,j],0);
constraint int_lin_ne(X_INTRODUCED_1,[a,k],0);
constraint int_lin_ne(X_INTRODUCED_1,[b,c],0);
constraint int_lin_ne(X_INTRODUCED_1,[b,d],0);
```

...many more lines...

```
constraint int_lin_ne(X_INTRODUCED_1,[i,k],0);
constraint int_lin_ne(X_INTRODUCED_1,[j,k],0);
solve  maximize c;
```

So the `alldifferent` constraint is actually not supported natively by `JaCoP` (at least when using MiniZinc/FlatZinc input files). Yet, it is much more efficient than `Minisat` on this example, as inequality of integer values is supported natively by the solver.

Due to the richness of the input formats of constraint programming tools, it makes little sense to discuss their capabilities in more detail here. The interested reader is rather referred to the MiniZinc tutorial (Marriott and Stuckey, 2014).

Constraint programming is commonly applied in *operations research*, as it can natively deal with optimization problems under constraints, which are the heart of that area of work. Also, it offers off-the-shelf support for many types of constraints, which make constraint programming tools a reasonable choice for such problems.

Note that not all constraint propagation solvers use concepts from SAT solving. Some are based on *integer programming* principles, which we will also get to know later in the course.

As a final note, constraint programming tools are often meant to be used as libraries from other programs. They often provide ways to tweak some parameters for an application domain and hence are seen less as *black boxes* than SAT solvers. While this makes their integration into domain-specific programs simple, it has the disadvantage that switching between libraries is difficult, so that trying out different constraint programming engines for an application domain is cumbersome.

3.3.1 Comparing constraint programming and SAT solving

Literature: *Petke, 2015*

Constraint satisfaction problem (CSP) solvers are the "big sisters" of SAT solvers – they support more operations and often use *propagation* and *branching* to perform the search for solutions in an efficient way, just as SAT solvers do. However, they are significantly more complex to engineer due to the many types of constraints that they typically support, and also do not enjoy the high level of optimization of modern SAT solvers. As a consequence, the authors of many research papers reduce their problems to SAT rather than using constraint programming. Encoding into SAT also gives more control over the encoding, which can be beneficial for some applications. But without knowing when CP solvers work well and when the principles applied in CDCL-style SAT solvers is better suited, we do not have a proper guideline for when to use CSP solvers and when to use SAT solvers.

Petke (2015) compared the basic reasoning engines of SAT solvers and CSP solvers to fill this gap. She focussed on reasoning efficiency rather than the simplicity of modelling a computational problem.

Firstly, she discusses the basic differences in the problem solving approaches: while clause learning is essential for SAT solving performance, it has been observed that lifting the idea to constraint programming did not yield big benefits. This observation is tightly connected to the way in which traditional CSP solvers operate: they *pre-process* the problem instance to be *locally consistent*. There are many different variations of local consistency, and we have already seen one for SAT solving in Section 2.6.5 during the discussion of the half adder example. In a nutshell, ensuring local consistency makes sure that best use of (unit) propagation is made. Constraint programming solvers often extend this concept to *k-path consistency*, where any assignment to $k-1$ variables should give rise to propagation if there is only one assignment to the kth variable that is consistent with all constraints and the $k-1$ other values. It seems that after a strong level of local consistency has been enforced, clause learning is less useful than in classical CDCL-style SAT solving. Petke proved that a purely randomly branching SAT solver approximates k-path consistency in a SAT instance in expected polynomial time for any fixed value of k. Hence, SAT solvers somewhat make use of the benefits of k-consistency, but are at the same time more powerful by being able to learn clauses that do not implement k-consistency and enforcing k-consistency on-the-fly during the solution process.

Constraint programming and SAT solving also have in common that in both cases, efficiently solvable fragments of the problem are known. For example, in SAT, problem instances that consist purely of Horn clauses or two-literal clauses can be solved in polynomial time (Franco and Martin, 2009, Chapter 1.19). Similar classes of easy CSP instances are known. Petke (2015) showed that for a couple of such classes, when encoding their CSP instances into SAT using the *phase transition encoding* (Section 2.6.3, p. 44), the resulting SAT instances fall into one of the categories that are easy to solve. Even more, she showed that modern CDCL-style SAT solvers can make use of the structure of the resulting SAT instances, so that they are solved quickly. Petke (2015, p.74) summarizes this observation as follows:

In practice, it seems that current clause-learning SAT solvers with highly tuned learning schemes, branching strategies and restart policies are often able to exploit structure in the Boolean encoding of a CSP instance even more effectively than local consistency techniques.

On the negative side, Petke (2015) showed that modern DPLL-style solvers have problems with Pidgeon-hole principle problem instances. While CSP solvers can make use of the structure of the problem if the problem is specified directly, for all standard encodings of objects/numbers given in Section 2.6.3 of these lecture notes, DPLL solvers need exponential time in the best case. This comes from the fact that so-called *extended resolution proofs* for the unsatisfiability of these instances need to be exponential-sized. As it has been shown that a DPLL-style SAT solver needs at least as many steps as there are components in such proofs, the exponential-time claim then follows. For the Pidgeon-hole principle problem, there is however also a specialized encoding that permits polynomial-sized such proofs. Her experiments show that modern DPLL/CDCL-style SAT solvers are not able to make use of this effect, so that even with this encoding, Pidgeon-hole principle instances remain difficult to solve for modern SAT solvers.

To sum it up, given that some time can be spent on performing the SAT encoding manually, using CSP solvers only seems to make sense if the problem to be encoded has constraints that are notoriously tricky for SAT solvers to deal with, such as the *all-different constraints* that underlie the Pidgeon-hole principle problem. While this problem may seem to be artificial at first, it actually appears as a *sub-problem* in many practical applications, which we want to discuss by means of an example.

Example 13. There are 20 actors available for a theatre play with 15 roles. We need to select 15 of 20 actors for filling all the available roles. To make the problem interesting, there are some constraints on which actors are suitable for which role. There are 13 actors that can fill any of the 15 roles, but there are 7 actors that are only suitable for roles 1 and 2. It such a case, it is clear that none of the 13 flexible actors may be assigned role 1 or role 2, as otherwise, there are not enough flexible actors left for 13 roles that are more difficult to fill.

The problem can be represented as a SAT instance with $15 \cdot 20$ variables without any unit clause. A CDCL-style SAT solver could, for instance, start the search process by assigning one of the flexible actors to role 1. If this happens, the the problem of competing this partial assignment to a satisfying one has no solution. To find this out, the SAT solver however needs to try to assign the 13 roles that are difficult to fill to 12 (flexible) actors, which is exactly an instance of the Pidgeon-hole principle problem. ★

The example also shows why modern CDCL-style SAT solvers use random restarts (see Section 2.4.3): they allow to get the solver out of parts of the solution space that contain no solution in case some bad decisions have been performed early in the search process.

3.4 Conclusion

In this section, we discussed the *periphery of SAT solving*, i.e., computational engines that are somewhat close to SAT solving, but extend it in non-trivial ways.

Pseudo-boolean constraints allow us to work with linear constraints over boolean variables, and they are especially well suited for problems in which numbers are only needed to define the constraints, but are not used in the model to be computed. However, pseudo-boolean constraints can also easily be used to perform computations over numbers that are encoded into the boolean variables.

`Alloy` and its model finder `Kodkod` allow to specify problems with non-fixed model size. They are well suited if we are searching for solutions whose objects form relations.

Finally, constraint satisfaction problem (CSP) solving is the big sister of SAT solving, as it allows more types of constraints and gives CSP solvers the possibility to perform reasoning on a higher level. However, they do not enjoy the sophisticated engineering that SAT solvers have received in the last two decades. We also have more control over the encoding when using SAT directly. If an encoding into SAT does not lead to good performance, we can easily change the encoding, which is more difficult to do with CSP.

Because we only treated these three approaches very briefly, we did not cover the questions what the limits of these approaches are and why they work well on many practical problems in a comprehensive way. However, all approaches are relatively close to SAT, so most of the results for SAT solving still hold. The three approaches add some extensions to the DPLL/CDCL way of reasoning to mitigate its limitations. Pseudo-boolean constraint solving comes with efficient built-in methods for encoding computations over numbers, `Alloy` uses some smart encoding techniques, and in constraint programming, optimizations that are adapted to the supported types of constraints are used. When problems are relatively unstructured and huge, all three computational engines considered in this chapter have scalability problems, however, just as SAT solving.

Chapter 4

QBF

In most of the computational problems that we have considered so far, it was easy to check a solution for correctness once it had been computed. Rather, the computational difficulty lay in actually find the solution.

There are many computational problems that do not have this property, however. Take for example the problem of computing a *sorting network*. We have already seen sorting networks in Chapter 2.6.4 as a tool for enforcing k-out-of-n constraints in SAT solving. Now we want to use a computational engine to automatically compute a correct sorting network for boolean values. There are many methods in the literature for constructing sorting networks, and they all have in common that the networks work for arbitrary domains. But perhaps the networks can be smaller if they only need to work on boolean values? So let us try to compute some smallest sorting networks for some values n of input bits.

The question that we are asking to a computational engine in this context is: "does there exist a sorting network of size m such that for all valuations to n input bits, they are always sorted correctly?". The reader may observe that we now have an alternation between the "does there exist" and "for all" quantifiers. To solve such problems, we cannot simply use a computational engine that can only deal with only one type of quantifiers (such as SAT solvers), but we rather have to consider computational engines that can deal with both of them.

The direct extension to SAT solving that can solve such problems is called *quantified boolean formula* (*QBF*) solving.

4.1 Definition

The QBF problem is very similar to the SAT problem. As it is the case for SAT solvers, most QBF solvers require the input to be a boolean formula in conjunctive normal form. However, in addition to the clauses, a so-called *prefix* that describes the quantification of the boolean variables is to be given.

Definition 10. *Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a set of boolean variables. A QBF instance in* prenex form *is defined by the grammar*

$$\Gamma ::= \quad \exists v.\Gamma \mid \forall v.\Gamma \mid \varphi,$$

where φ is a CNF formula over $\mathcal{V}$, $v \in \mathcal{V}$, and all variables in $\mathcal{V}$ appear immediately right of some $\exists$ or $\forall$ quantifier in the problem precisely once.

Note that a QBF instance can be seen as a *closed* boolean formula, i.e., one that is equivalent to **true** or **false**. The *QBF problem* is to check for some given QBF formula whether it is equivalent to **true** or **false**.

Example 14. Consider the following QBF instance:

$$\psi = \exists a.\forall b.\exists c.\exists d.(\neg a \vee b) \wedge (a \vee b \vee c) \wedge (a \vee b \vee d) \wedge (\neg b \vee \neg c)$$

The instance evaluates to **true** by the following reasoning: First of all note that in order to satisfy the formula, a must not be set to **true**, as otherwise the first clause is not satisfied if b is set to **false**. Since a must have a value such that for all values of b, the CNF formula can be satisfied, this means that a must be **false**.

Then, we consider the two possible values for b. If b is **false**, the second and third clauses imply that c and d have to be **true**. If b is **true**, the last clause implies that c must be **false**. In both cases, we find values for c and d that make all clauses satisfied. Overall, ψ thus evaluates to **true**. ★

In a QBF instance (in prenex form), we call the quantifier structure the *prefix* of the formula, whereas the CNF formula is called the *matrix*. If a formula is equivalent to **true**, the formula is also called *valid*. Borrowing terminology from SAT solving, a valid QBF instance is also called *satisfiable*.

4.2 Example: finding sorting networks

Let us now formulate the search for a sorting network that sorts boolean values correctly as a QBF problem. Without loss of generality, we can assume that on every level of the sorting network, only a single comparator module is located. We use three groups of variables:

- The outermost existentially quantified variables represent the sorting network.
- The middle universally quantified variables represent the arbitrary input for which the sorting network has to work correctly.
- The innermost existentially quantified variables represent the intermediate values of the network's lines when exposed to the input.

Let us assume that we want to check if n input bits can be sorted with m comparator modules in the network.

We use $n \cdot m$ many variables $x_{i,j}$ for encoding the network. A variable $x_{i,j}$ has a **true** value if comparator module j touches the ith line of the network. For every $1 \leqslant j \leqslant m$, exactly 2 variables of $x_{i,j}$ for $1 \leqslant i \leqslant n$ must be set to **true**. We use a variant of the direct encoding and need $O(m^3)$ many clauses to enforce that precisely two variables of $\{x_{i,1}, \ldots, x_{i,m}\}$ have a **true** value.

For encoding the values to be sorted, we use n many elements y_i (for $1 \leqslant i \leqslant n$).

Then, for encoding the behavior of the sorting network on the input signal valuation given in the variables $\{y_i\}_{1 \leqslant i \leqslant n}$, we use $n \cdot m$ many variables $z_{i,j}$. A variable $z_{i,j}$ should have a **true** value if (and only if) after comparator module j, there is a **true** value on signal line i of the sorter. Note that unlike in Section 2.6.4, this means that we have variables for the values on all lines after each of the comparator modules, rather than having variables only for whose lines whose values were potentially changed by the sorter. This is because at the time of encoding, the positions of the sorters are not yet fixed.

We need a couple of constraints to enforce that the variables $\{z_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ are filled with correct data. We start with the ones that ensure that the first comparator module operates on the input y_i in the correct way:

$$
\begin{aligned}
& \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee y_i \vee \neg z_{i,1}) \\
\wedge & \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee y_{i'} \vee \neg z_{i,1}) \\
\wedge & \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee \neg y_i \vee \neg y_{i'} \vee z_{i,1}) \\
\wedge & \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee y_i \vee y_{i'} \vee \neg z_{i',1}) \\
\wedge & \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee \neg y_i \vee z_{i',1}) \\
\wedge & \bigwedge_{1 \leqslant i \leqslant n, i < i' \leqslant n} (\neg x_{i,1} \vee \neg x_{i',1} \vee \neg y_{i'} \vee z_{i',1})
\end{aligned}
$$

These clauses have been obtained by translating a truth table for a comparator module over two input bits and two output bits into CNF form and implementing the truth table for all possible first comparator

n	m	Satisfiable	Computation time
3	2	✗	0.016s
3	3	✓	0.017s
4	4	✗	0.045s
4	5	✓	0.067s
5	5	✗	2.915s
5	6	✗	11m47.987s
5	7	?	$> 20h$
5	8	?	$> 20h$
5	9	✓	4.938s

Table 4.1: Computation times of `depqbf` on sorting network finder instances from Chapter 4.2.

modules and their respective signal line pairs. We also need a couple of clauses for enforcing that signal lines that are not touched by the first comparator module stay unchanged:

$$\begin{aligned} &\bigwedge_{1\leqslant i\leqslant n} (x_{i,1} \vee y_i \vee \neg z_{i,1}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n} (x_{i,1} \vee \neg y_i \vee z_{i,1}) \end{aligned}$$

For values of $j > 0$, the jth comparator module operates on the values $\{z_{i,j-1}\}_{1\leqslant i\leqslant n}$ and sorts a pair of them. The result needs to be encoded into $\{z_{i,j}\}_{1\leqslant i\leqslant n}$. Using the same constraint shape as before, we obtain the following constraints:

$$\begin{aligned} &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee z_{i,j-1} \vee \neg z_{i,j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee z_{i',j-1} \vee \neg z_{i,j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee \neg z_{i,j-1} \vee \neg z_{i',j-1} \vee z_{i,j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee z_{i,j-1} \vee z_{i',j-1} \vee \neg z_{i',j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee \neg z_{i,j-1} \vee z_{i',j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, i<i'\leqslant n, 2\leqslant j\leqslant m} (\neg x_{i,j} \vee \neg x_{i',j} \vee \neg z_{i',j-1} \vee z_{i',j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, 2\leqslant j\leqslant m} (x_{i,j} \vee z_{i,j-1} \vee \neg z_{i,j}) \\ \wedge &\bigwedge_{1\leqslant i\leqslant n, 2\leqslant j\leqslant m} (x_{i,j} \vee \neg z_{i,j-1} \vee z_{i,j}) \end{aligned}$$

Finally, we require that all values are sorted correctly by the sorting network. The following constraints require that they are ordered along $z_{1,m}, \ldots, z_{n,m}$ in ascending order.

$$\bigwedge_{1\leqslant i<n} (\neg z_{i,m} \vee z_{i+1,m})$$

We can now ask a QBF solver for some values of n and m whether there exists a sorting network that sorts n values correctly with m comparator modules. Table 4.1 lists some results obtained with the QBF solver `depqbf 2.0-1`. All computation times were obtained on an AMD Opteron 2220 SE 2.80GHz machine, running an x64 version of Linux. It can be seen that the computation times are very low for low values of n and m and grow rapidly with higher values of m and n.

n	m	**Satisfiable**	**Computation time**
3	2	✗	0.012s
3	3	✓	0.015s
4	4	✗	0.032s
4	5	✓	0.093s
5	5	✗	1.750s
5	6	✗	46.962s
5	7	✗	202m14.757s
5	8	?	>15h
5	9	✓	3.597s

Table 4.2: Computation times of `depqbf` on sorting network finder instances from Chapter 4.2 with rudimentary symmetry breaking.

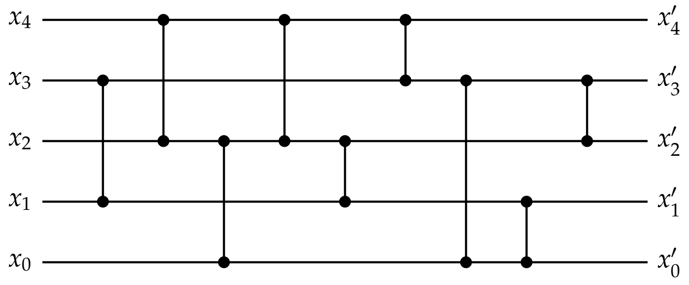


Figure 4.1: A smallest possible sorting network for 5 input values to be sorted (found using QBF solving)

In order to reduce them, let us try to apply symmetry breaking. Note that all values of $\{z_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ are fixed by the values of $\{x_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ and $\{y_i\}_{1 \leqslant i \leqslant n}$. So we would have to be very careful when adding symmetry breaking clauses over $\{z_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ to the QBF instance in order not to rule out possible solutions. For the values of $\{y_i\}_{1 \leqslant i \leqslant n}$, it is important that they can be arbitrary, so adding symmetry breaking constraints on them does not seem to be a good idea.

However, we can add a lot of constraints over $\{x_{i,j}\}_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ to reduce the search space. Whenever comparator modules that are adjacent in the order of modules do not have some signal line in common, i.e., whenever for some distinct values of a, b, c, and d with $a < b$ and $c < d$, we have that all of $x_{a,j}$, $x_{b,j}$, $x_{c,j+1}$, $x_{d,j+1}$ are all **true** for some $1 \leqslant j < m$, then we can assume without loss of generality that $a < c$. This is because we could swap the two comparator modules j and $j+1$ in such a case without affecting the overall behavior of the network. As comparator modules j and $j+1$ are concerned with different signal lines, their sorting operations can be performed in any order.

Adding these symmetry breaking constraints to the QBF instances gives us some improvement in computation time, as witnessed in Table 4.2. Still, the approach only works for small values of n and m. Figure 4.1 shows the sorting network found for the $(n, m) = (5, 9)$ case.

We can now check if for some n, we found a sorting network for n boolean values that is smaller than the smallest sorting network for an arbitrary value domain in the literature.

According to Knuth (1998, p. 226), the smallest possible networks have

- 1 comparator module for $n = 2$,
- 3 comparator modules for $n = 3$,
- 5 comparator modules for $n = 4$,
- 9 comparator modules for $n = 5$, and
- 12 comparator modules for $n = 6$.

We can see in Table 4.2 that we did not find sorting networks that are smaller than that. So our restriction to sorting networks that only need to sort boolean values correctly did not help to find smaller networks. There is actually a theorem (Knuth, 1998, Theorem Z, p.223) that states that if a sorting network sorts boolean values correctly, then it is a correct sorting network for an arbitrary value domain, which explains our inability to find smaller networks.

4.3 QBF as a game

The alternating quantifiers in a QBF formula give rise to the idea of modeling the validity of a QBF formula (i.e., whether it is equivalent to **true**) as a game between two players. They are called the *existential player* and the *universal player*. Given a QBF problem in prenex form, it is the aim of the existential player to satisfy the matrix of the problem, whereas the universal player wants to falsify the matrix. The game is played from the outer-most quantifiers to the innermost quantifiers, and for every existentially or universally quantified variable, the respective player can choose the value of the variable.

Whenever a player reaches her aim, we say that the player *wins* a play of the game. We say that a player *wins the game* if she has a *strategy* that ensures that all plays corresponding to the strategy are winning for her.

Example 15. Reconsider QBF formula ψ from Example 14. In the game that models the formula, in the first move, it is the existential player's turn to choose a value for a, then the universal player can choose a value for b, and finally the existential player can choose values for c and d.

The existential player can win the game with the following strategy: in the first round of the game, she sets a to **true**. Then, when it is her turn again, she sets c to $\neg b$, and d to **true**. All plays of the game that are consistent with this strategy are winning for the existential player, so she wins the game. ★

Note that for every QBF formula, one of the players has a winning strategy to win the game induced by the formula. This follows directly from the fact that a QBF formula is always equivalent to **true** or **false**.

Viewing a QBF formula as a game shows that QBF formulas are also well-suited to model games. For example, we could model the question whether for a configuration of a chess board, the black player has a strategy to win within k steps as a QBF formula. In this QBF formula, there would be $2k - 1$ *alternations* between the existential and universal quantifiers.

Games also give rise to an important observation: for a set of variables that are all universally quantified or all existentially quantified, if the variables appear next to each other in the prefix, they can be assigned values in an arbitrary order without affecting the outcome of the game.

For example, in the QBF formula ψ, for the variables c and d, the existential player can select values for them in any order. This is because when it is the existential player's turn to set of a value for c in the game built from the QBF formula, she knows that it will be her turn until the value for d has been set as well. Thus, she can plan ahead and choose values for all of the variables in a *level* of the prefix at a time. We say that some variables are on the same level of the QBF formula's prefix if they have the same quantifier and appear next to each other in the quantifiers' order. Levels are counted from the inner-most to outer-most quantifiers.

Example 16. The following QBF formulas in prenex form are annotated with the levels of their variables.

$$\underbrace{\exists a.}_{level\ 3}\ \underbrace{\forall b.}_{level\ 2}\ \underbrace{\exists c.\exists d.}_{level\ 1}\ (\neg a \vee b) \wedge (a \vee b \vee c) \wedge (a \vee b \vee d) \wedge (\neg b \vee \neg c) \tag{4.1}$$

$$\underbrace{\forall a.\forall b.\forall c.}_{level\ 4}\ \underbrace{\exists d.\exists e.}_{level\ 3}\ \underbrace{\forall f.}_{level\ 2}\ \underbrace{\exists g.\exists h.}_{level\ 1}\ (a \vee h) \wedge (b \vee g) \wedge (c \vee f) \wedge (d \vee e) \tag{4.2}$$

★

4.4 A DPLL-style algorithm for QBF

Many modern QBF solvers use a variant of the DPLL algorithm from Section 2.4 as basic reasoning engine and also apply clause learning, just like a SAT solver. However, due to the universal quantifiers, DPLL-style reasoning is embedded into a more complex framework.

Algorithm 2 on page 26 describes a DPLL algorithm for the satisfiability problem. In order to perform DPLL-style reasoning for the QBF problem, we can extend it as shown in Algorithm 3.

Algorithm 3 A DPLL-style algorithm for QBF solving. The parameter $\mathcal{V}$ represents the set of variables, ψ is the QBF formula (with prefix ψ_p and matrix ψ_m), and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment.

```
 1: function SEARCH($\mathcal{V}$,$\psi$,$A$)
 2:     for all assignments $v_k = b_k$ implied by $(\psi_m, A)$ by unit propagation do
 3:         if $v_k$ is universally quantified then
 4:             return false
 5:         else
 6:             $A := A \cup \{v_k \mapsto b_k\}$
 7:         end if
 8:     end for
 9:     for all assignments $v_k = b_k$ implied by $(\psi_m, A)$ by pure literal elimination do
10:         if $v_k$ is existentially quantified then
11:             $A := A \cup \{v_k \mapsto b_k\}$
12:         else
13:             $A := A \cup \{v_k \mapsto \neg b_k\}$
14:             goto 2
15:         end if
16:     end for
17:     if some clause in $\psi_m$ is falsified by $A$ then
18:         return false
19:     end if
20:     if $|A| = |\mathcal{V}|$ then
21:         return true
22:     end if
23:     Pick a variable $v$ that is not yet in the domain of $A$ from the highest level of the prefix that still
        has variables without an assignment in $A$.
24:     if $v$ is an existential variable then
25:         if SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ false$\}$)= true then return true
26:         if SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ true$\}$)= true then return true
27:         return false
28:     else
29:         if SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ false$\}$)= false then return false
30:         if SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ true$\}$)= false then return false
31:         return true
32:     end if
33: end function
```

Unlike Algorithm 2, the new QBF DPLL algorithm does not return a satisfying assignment (if one exists), but only **true** or **false**. The general idea of Algorithm 2 is however retained in the new DPLL-style algorithm for QBF solving. As before, the algorithm uses unit propagation and pure literal elimination to make *safe choices*. All branching in the search is done from line 23 of the algorithm onwards.

When branching, the algorithm always only picks variables from the outermost level of the QBF formula's prefix that does not have all values fixed yet. This resembles the order of moves of the two players in a QBF game. If the game reaches some quantifier level i, the same player stays in charge of making the next moves until all variables on level i have been assigned values.

If an existential variable is picked for branching, the algorithm checks (by recursion) if setting the chosen value to **true** can lead to a solution and if setting the value to **false** can lead to a solution. Whenever one of these cases holds, the algorithm returns **true** as the (sub-)problem has a solution. Otherwise, it returns **false**. For universally quantified variables, the algorithm checks if solutions can be found for both possible values of the variable. Taking both cases together, the algorithm thus checks if the existential player has a strategy to win the game if the choices in A have already been made by the two players.

The unit propagation and pure literal eliminations steps deserve a closer look, as they make their decisions *out of order*, meaning that they fix the values of some variables before all variables of the higher levels have been assigned values. In all cases, fixing the values ahead of time is sound, however, which we can deduce by a case-by-case analysis.

In the case of unit propagation, fixing an existential variable ahead of time is sound as the existential player loses if the literal does not hold on the *play*, i.e., the final set of decisions made by the two players. Thus, at the point of the play when unit propagation can be applied, both players know that the existential player must later ensure that the literal holds, as otherwise she loses the play. Thus, fixing the value ahead of time is sound.

If unit propagation finds a literal of a universally quantified variable, however, the procedure can immediately stop. We know that in such a case, the universal player can later in the play choose a value that falsifies the literal, which also falsifies the complete clause in which the literal is located. This makes the universal player win the play, regardless of the other moves that are still to be performed in the play.

Applying the pure literal rule is sound for existential variables: there is simply no incentive for the existential player to falsify the literal later in the play, regardless of the moves made by the universal player in the meantime. In case of a universally quantified variable, if a literal is pure, then the player has an incentive to falsify the literal, as this only makes winning the play *harder* for the existential player. Thus, we can assume for the search that the universal player will do so when it is her turn to assign the literal's variable a value. As this assignment causes some literals in some clauses to be falsified, the algorithm then jumps back to the unit propagation step to test if more unit propagation can be performed.

The following example shows the operation of Algorithm 3 on an actual QBF formula.

Example 17. Let the following QBF problem over $\mathcal{V} = \{a, b, c, d, e\}$ be given:

$$\psi = \underbrace{\exists a.\exists b.}_{level\ 3}\ \underbrace{\forall c.\forall d.}_{level\ 2}\ \underbrace{\exists e.\exists f.}_{level\ 1}\ (b \vee c) \wedge (\neg b \vee a) \wedge (\neg c \vee e) \wedge (\neg b \vee d \vee \neg f)$$
$$\wedge\ (\neg a \vee \neg b \vee \neg d \vee f) \wedge (\neg e \vee b)$$

When function Search from Algorithm 2 is called for the empty partial valuation, it cannot initially perform unit propagation or pure literal elimation. So it picks a variable from the highest level that has an unfixed variable value. Let us assume that it picks b. So it recursively calls itself on the partial valuation $\{b \mapsto \textbf{false}\}$. Now, unit propagation can be applied on the variable c for the first clause. Variable c is universally quantified, so the algorithm return **false** for this partial valuation in line 6. As alternative, the function tries to set b to **true** by recursing on the partial valuation $\{b \mapsto \textbf{true}\}$.

Now, unit propagation can deduce that $a \mapsto \textbf{true}$ (by the second clause) and the pure literal rule adds $c \mapsto \textbf{true}$ to the partial valuation, as the among the clauses not yet satisfied, the literal c does not occur any more, and c is a universally quantified variable. This gives rise to more unit propagation: by the third clause, we can now set e to **true**.

The algorithm now has to assign d a value, as all variables on level 3 have been assigned values at this time, and it is the only variable left at level 2 without a value. Choosing $d \mapsto \textbf{false}$ and recursing leads to unit propagation on the fourth clause and setting f to **false**. This completes the assignment. As the resulting assignment satisfies the matrix of ψ, the algorithm returns **true**. Coming back from the recursive call, the algorithm furthermore tries the case of $d \mapsto \textbf{true}$. During the recursive call on $\{b \mapsto \textbf{true}, a \mapsto \textbf{true}, c \mapsto \textbf{true}, d \mapsto \textbf{true}\}$, unit propagation on the fifth clause leads to $f \mapsto \textbf{true}$, again completing the partial assignment to one that already satisfies the matrix. Coming back from this recursive call, the function returns **true** in line 31. Finally, coming back from the call in line 26, the function also returns **true** on the highest level of recursion. All in all, the function thus finds ψ to be satisfiable. ★

Algorithm 2 is only a highly simplified version of the reasoning performed by modern search-based QBF solvers. For example, the type of unit propagation that they apply is more sophisticated than the standard unit propagation in SAT solving. In standard SAT solving, we apply unit propagation if all literals except for one have are falsified by the current partial assignment. For the treatment of QBF, we can extend the approach by letting it ignore all universal literals with a level that is lower than the level of literal that we want to propagate. This is best shown with an example.

Example 18. Let us consider the QBF instance

$$\underbrace{\exists a.\exists b.}_{level\ 3}\ \underbrace{\forall c.}_{level\ 2}\ \underbrace{\exists d.}_{level\ 1}\ (a \vee c \vee b) \wedge \psi',$$

where ψ' is some arbitrary set of clauses over $\mathcal{V} = \{a, b, c, d\}$. Using the game-based view onto the QBF satisfaction problem, we assume that the existential player chose a value of **false** for a.

After this decision, we can see that it is safe to set b to **true**. This is because clause $(a \vee c \vee b)$ has only three literals, the first literal has already been falsified, and the literal c refers to a universally quantified variable at a lower level. Thus, if the existential player would not set b to **true**, this would allow the universal player to select c = **false** later to falsify the clause. By allowing unit propagation of a literal if all other literals in the clause either (1) have been falsified by the partial assignment, or (2) are universally quantified variables at a lower level than the literal chosen for propagation, we can capture this case.

Note that the restriction to universally quantified variables with a *lower* level than the literal to be propagated is actually important for soundness, as the following formula shows:

$$\underbrace{\exists a.}_{level\ 3}\ \underbrace{\forall c.}_{level\ 2}\ \underbrace{\exists d.}_{level\ 1}\ (a \vee c \vee \neg d) \wedge (a \vee \neg c \vee \neg d) \wedge (\neg a \vee c)$$

In the first step, unit propagation yields $\neg a$, as the existential player has no control over c, and thus setting a to **true** would allow the universal player to win.

Note that the existential player can then win the game by setting d to the same value as c. When ignoring the condition for extended unit propagation that all remaining universal literals must belong to levels lower than the one of the literal to be propagated, we would incorrectly determine that $\neg d$ must hold (by the first clause) and that d must hold (by the second clause), leading to a conflict despite the validity of the QBF formula. ★

Search-based QBF solvers obtain a good part of their reasoning efficiency from the unit propagation and – if implemented – pure literal elimination steps, which both help with solving QBF problems in a more efficient way than by pure brute force. However, there are many more techniques that they apply in addition, such as clause learning, which is more complicated in the QBF case than it is in the SAT case. Also, many solvers apply *cube learning*, which is the dual concept of *clause learning*, i.e., where the solver learns that fixing the values of some variables to some given values always allows the existential player to win the QBF game.

In addition, modern QBF solvers *preprocess* their input instances in order to simplify them before staring the search process. In the preprocessing step, a variety of transformation rules simplify the instance but do not alter its satisfiability. Preprocessing is generally seen as more important for QBF solving than for SAT solving, as preprocessing is computationally somewhat expense. As QBF solvers tend to require more time for the search process, the payoff of preprocessing becomes larger in the SAT case.

4.5 QBF solving in practice

Search-based QBF solvers typically take their input in the so-called *QDIMACS* format[1], which is similar to the SAT solver input format described in Section 2.3.

[1]Available at `http://www.qbflib.org/qdimacs.html`, retrieved on the 17*th* May 2016.

A QDIMACS file always starts with a line of the form `p cnf <numberOfVariables> <numberOfClauses>`, which has the same meaning as for a SAT file. Immediately after the first line, the quantifier lines establish how and in which order the variables are quantified. These lines begin either with a `e` or `a`, followed by the `0`-terminated list of variables, separated by spaces. For example, the lines

```
a 2 5 0
e 4 3 0
```

establish the quantification order

$$\forall x_2, x_5 . \exists x_4, x_3 . \ldots,$$

where we assume $x_1, \ldots, x_n$ to represent the variables from the QBF instance. The quantifier lines are followed by the clause lines, which are written in the same way as in the DIMACS format for SAT solving. Comment lines can appear everywhere in the input file and start with a `c`.

As an example, the QDIMACS file for the QBF formula from Example 17 is given as follows (for `1` to `6` representing the variables a to f):

```
p cnf 6 6
e 1 2 0
a 3 4 0
e 5 6 0
2 3 0
-2 1 0
-3 5 0
-2 4 -6 0
-1 -2 -4 6 0
-5 2 0
```

The QDIMACS standard also imposes some additional constraints, whose satisfaction is needed by some solvers:

- There shall not be two "`e`" lines or two "`a`" lines in succession in the input file.
- Each variable used in the prefix must appear in at least one clause.

Variables that are used in the matrix but do not appear in any quantifier line are treated as *outermost existentially quantified*. When using a QBF solver in practice, this is a source for errors. For example, for variables introduced in the *Tseitin encoding* (see Section 2.6.2), treating them as outermost existential normally leads to wrong results. Such variables are normally safe to treat them as *innermost existential*, however, so that they need to be mentioned explicitly in the QBF prefix in the QDIMACS file.

There are many QBF solvers that operate on QDIMACS QBF instances. Using, for example, the solver `depqbf` 5.0 (Lonsing, 2015) with the input instance from above yields the following output:

```
SAT
```

The solver `depqbf` can also print an assignment to the outermost existential variables for satisfiable instances. The parameter `--qdo` does this. The resulting output of the solver is:

```
s cnf 1 6 6
V 1 0
V 2 0
```

The first line represents the answer (1 = satifiable, 0 = unsatisfiable, -1=unknown), the number of variables, and the number of clauses in the input file. The other lines contain the variable assignments for the outermost existentially quantified variables, where a negative literal means a **false** value for a variable, and a positive one denotes a **true** value. Here, the QBF solver found that setting both the first and the second variables to **true** leads to the validity of the overall QBF instance.

4.6 Counter-example guided inductive synthesis (CEGIS) for QBF

The QBF solving algorithm from the previous section is applicable to all QBF formulas, but this generality comes at the price of performance. For some types of problems, researchers have developed special decision procedures that are adapted to them.

For example, many QBF problems have the form of the one in Chapter 4.2, where we are searching for an implementation to some specification that works for all possible inputs. Written as a QBF formula, the shape of the problems looks as follows:

$$\exists implementation.\forall inputs.\exists helper\_variables.\psi(implementation, inputs, helper\_variables)$$

In such a problem, the values of the helper variables are often only dependent on the choices of the implementation and the input. This is the case when searching for sorting networks of small sizes, and we henceforth assume this to be the case for the problems of interest.

Counter-example guided inductive synthesis (Solar-Lezama et al., 2006) is a method to deal with such computational problems. It is based on the idea that in order to compute an implementation, it often suffices to only consider relatively few valuations of the input variables in order to restrict the search for implementations to correct ones.

Given for example three input variable valuations $\vec{i}_1$, $\vec{i}_2$, and $\vec{i}_3$, we can ask a SAT solver to find an implementation that works for all three of them by solving the following problem:

$$\begin{aligned} &\exists implementation, helper\_variables_1, helper\_variables_2, helper\_variables_3. \\ \psi(implementation, \vec{i}_1, helper\_variables_1) \wedge{}& \psi(implementation, \vec{i}_2, helper\_variables_2) \\ &\wedge \psi(implementation, \vec{i}_3, helper\_variables_3) \end{aligned}$$

However, we do not know in advance which valuations to the input variables are the right ones to force the solver to find implementations that work on all possible input variable valuations. For this purpose, we embed the computation of a suitable implementation into a *counter-example-guided refinement loop*.

Algorithm 4 describes the overall appproach. The CEGIS computation is started by calling the function Search, which performs the search for an implementation. It needs two copies of the specification ψ, namely ψ_p and ψ_n. The first of them is a set of clauses that ranges over a set of implementation variables, a set of input variables, and a set of helper variables and is satisfiable for some valuation of the implementation and helper variables if the implementation represented by the variables' values works correctly on input vector given in the input variable valuation, and helper variables have values that prove this. The clause set ψ_n has the same interface but encodes that the input variables' values encode an input vector to the implementation on which it works *incorrectly*.

Function Search returns an implementation if one can be found. It maintains a set of counter-examples C that is initially empty. In line 12 of the algorithm, it is checked if an implementation exists that works correctly for all input variable valuations in C. If one is found, the function Check is called to test if the found implementation actually works on all input variable valuations. If this is the case, the overall CEGIS loop terminates, as we have found an implementation then. Otherwise, the counter-example to the conjecture that the implementation is correct is returned to the Search function, where it is added to the set of input variable valuations for which the implementations to be subsequently found must be correct.

Note that with every added counter-example, the number of variables in the SAT instance computed in line 12 increases, as a new set of helper variables is needed for every counter-example. Also note that the check can never compute the same implementation twice, as adding the counter-example computed by Check to C prevents this.

Let us now apply the CEGIS approach to the sorting network finding problem. We start with the encoding from Chapter 4.2. The specifications ψ_p and ψ_n are almost the same. The only difference is that ψ_p contains clauses that state that the output of the sorting network is sorted correctly, whereas in ψ_n, we have clauses that require the output to be sorted incorrectly. In both cases, the helper variables

Algorithm 4 CEGIS algorithm.

```
1: function CHECK($\psi_n$,$\vec{t}$)
2:     if $\psi_n(\vec{t}, input, helper\_variables)$ is satistiable with model $m$ then
3:         return $m|_{input}$
4:     else
5:         return unsatisfiable
6:     end if
7: end function
8:
9: function SEARCH($\psi_p$,$\psi_n$)
10:     $C = \varnothing$
11:     while true do
12:         Check if $\bigwedge_{\vec{c} \in C} \psi_p(implementation, \vec{c}, helper\_variables_c)$ is satisfiable
13:         if sat then
14:             $\vec{t}$ := the valuation of the implementation variables in the assignment found
15:             if CHECK($\psi_n$,$\vec{t}$) yields a new assignment $\vec{d}$ then
16:                 $C := C \cup \{\vec{d}\}$
17:             else
18:                 return $\vec{t}$
19:             end if
20:         else
21:             return unrealizable
22:         end if
23:     end while
24: end function
```

n	m	Satisfiable	Computation time	# of counter-examples
3	2	✗	0.004s	4
3	3	✓	0.004s	2
4	4	✗	0.025s	9
4	5	✓	0.054s	11
5	5	✗	0.237s	12
5	6	✗	1.032s	15
5	7	✗	11.114s	22
5	8	✗	1m7.155s	26
5	9	✓	0m0.114s	16

Table 4.3: Computation times of a CEGIS implementation for finding sorting networks.

encode the behavior of the encoded sorting network on the input instance, and the clauses that differ between ψ_p and ψ_n only concern the final output of the sorting network.

Table 4.3 shows running times and numbers of counter-examples needed to compute sorting networks or to deduce that none of the specified size exists based on a simple implementation of CEGIS for this application. The implementation uses `picosat v. 959` as SAT solver, and performs *incremental solving* in the SEARCH function, i.e., rather than building the SAT instance from scratch in every iteration of the loop, the internal state of the SAT solver is retained between the iterations, so that in every step, only the clauses for the latest counter-example need to be added and clauses that have been learned by the SAT solver are kept between its executions.

It can be seen that the CEGIS-based approach is a lot faster than the general QBF solver for the problem of finding sorting networks. We can also see that the number of counter-examples needed varies a lot between the search problem for different values of n and m. It is actually a property of the CEGIS approach that we may be "lucky" early on in the search to find an implementation that works on all possible input variable valuations when the SAT instance built in the SEARCH function still has multiple solutions that do not all represent correct implementations.

The sorting network synthesis problem has the interesting property that certain input variable valuations can never be found as counter-examples. In particular, input variable valuations that are already sorted can never by "unsorted" by a sorting network. For $n = 5$, there are 6 such input variable valuations. For the case of $n = 5$ and $m = 8$, we can actually see that the CEGIS loop enumerated *all* other input variable valuations. This shows that the problem at hand is actually relatively difficult for CEGIS.

There are many problems in which the number of counter-examples needed for unrolling the universal quantifier before a solution is found is actually low. This motivated the application of CEGIS to also solve general QBF problems. Janota et al. (2012) employ several nested copies of the CEGIS procedure explained above. The number of nestings needed depends on the number of quantifier alternations for the prefix of the given QBF problem.

4.7 Further applications, discussion, and summary

In this chapter, we dealt with the problem of solving quantified boolean formulas. There is a good number of practical problems that can be concisely encoded into QBF. We have seen a synthesis problem as an example, but there are also others. For example, Cook et al. (2005) use QBF solving as a sub-step in the verification of asynchronous programs. While the main work is performed by a SAT solver in their setup, the QBF solver is applied to check if they reached a *fixpoint* of a function that computes states that are reachable from some other set of states. QBF solvers can also be used for *partial design verification*, where we want to find bugs in hardware of software even before the hardware or software has been fully designed. As a final example, the planning problem from Chapter 2.7.3 can be extended to support *non-deterministic operators*, which allow the world to evolve in multiple ways for some operators (Rintanen, 2009, Chapter 15.7). A plan in this case corresponds to a strategy to reach a goal state regardless of the behavior of the environment (within its limits defined by the non-determinism of the operators).

We discussed two approaches for solving QBF problems. In the first of these, we extended the DPLL algorithm for SAT solving to support universal quantifiers. Then, for the special case of so-called $\exists\forall$ problems (where the helper variables that have fixed values are ignored in this name), counter-example guided inductive synthesis (CEGIS) can speed up the solution process. It should be added, however, that the sorting network finding problem that we used as running example is illustrative, but not a good example for QBF solving. Bundala and Zavodny (2014) recently gave a pure SAT-solving based approach to finding small sorting networks, and they were able to compute all smallest sorting networks for $n \leqslant 10$ input lines. Since a pure SAT-based approach is feasible, using computational engines for problems with higher complexity (PSPACE-complete in the general QBF case, and Σ_2-complete for the CEGIS case) is probably not the way to go.

As a final note, it should be added that we skipped over a few fine details of encoding a problem into prenex QBF form. For example, we have assumed in this chapter that the problems naturally come in prenex form. Translating a problem in which the quantifiers are located deeper in the boolean formula to this form is however easy, and the interested reader can look up the details of this step in Giunchiglia et al. (2009).

Chapter 5

Linear Programming

In the previous chapters, we dealt with computational engines for problems that were at least NP-hard. These engines allow their users to make use of the reasoning procedures developed for dealing with NP-hard problems, without the need to worry about the details of these procedures.

In this chapter, we discuss the *linear programming* (LP) problem, which can be solved in polynomial time. Strictly speaking, the problem does consequently not fit into the scheme of problems considered so far. However, the *simplex algorithm*, which is one of the algorithms used to solve linear programs, actually has an exponential worst-case running time. Still, it is useful in practice, as it performs well on many practical problems from operations research or logistics that we can encode into LP problems. This connects the simplex procedure with, for instance, SAT solvers, which give quite weak performance guarantees, but are also observed to work well in practice.

Linear programming is however not only limited to logistics and operations research applications, but is also of interest to the computer scientist. Especially the extensions of linear programming that we will discuss, namely *integer linear programming* (ILP) and *mixed-integer linear programming* (MILP), are used in computer science applications such as *scheduling*. Both of these extensions are NP-hard again.

5.1 Definition

A linear programming problem is given by

1. a finite set of real-valued variables $\mathcal{V}$,
2. a finite set C of linear equations and linear inequalities over $\mathcal{V}$, and
3. a linear function $f : (\mathcal{V} \to \mathbb{R}) \to \mathbb{R}$ over $\mathcal{V}$

The elements of C are called the *constraints* of the LP problem, whereas f is called the *optimization function*. Together with $\mathcal{V}$, they form a *linear program*.

We call an assignment $\vec{x} : \mathcal{V} \to \mathbb{R}$ a *solution* of the LP problem $(\mathcal{V}, C, f)$ if x satisfies all constraints in C. We call $f(\vec{x})$ the *quality of the solution $\vec{x}$* and say that $\vec{x}$ is an *optimal solution* if there exists no other solution $\vec{y}$ such that $f(\vec{y}) > f(\vec{x})$.

The linear programming problem is to find such an optimal solution.

Example 19. Let $\mathcal{X} = \{x, y, z\}$ and C consist of the following constraints:

$$\begin{aligned} x &\geqslant 0 \\ y &\geqslant 0 \\ z &\geqslant -15 \\ x + y &= 4 \\ \frac{1}{2}x - \frac{3}{4}z &\leqslant 1 \\ x + y + z &\leqslant 24.5 \end{aligned}$$

Let the optimization function furthermore be $f(x, y, z) = \frac{1}{6}x + \frac{1}{3}y + \frac{1}{2}z$. As the constraints are all linear inequalities and linear equations, and the optimization function is also linear, this constitutes a valid linear programming problem. ★

Given a linear programming (LP) problem, we can use a LP solver to find out a best possible valuation of the variables, i.e., a valuation that maximizes the value of the optimization function. For Example 19, one such assignment is $(x, y, z) = (0, 4, 20\frac{1}{2})$. The value of the optimization function is $69\frac{1}{2}$ for this valuation.

It is essential for most LP solving algorithms that all constraints and the optimization function are actually linear. So constraints such that $x \cdot y \geqslant 30$ are disallowed if x and y are both variables in the problem. We will often multiply several variables when modelling practical problems as LP instances. If all variables in a constraint except for one have a problem-dependent fixed value, this is however allowed, as the constraints and optimization functions that we are getting are then still linear.

5.2 Example application: factory production

Assume that we are in charge of a factory that can manufacture 3 different kinds of goods. There are 4 machines in the factory that can be used for up to 100 hours per month, except for the fourth machine that needs 5 hours of maintenance per month, so that only 95 machine hours are available per month. Producing any of the three goods needs some time on the machines, and as all machines have different purposes, the time needed per machine is fixed and differs between the goods. In particular,

- the production of one item of good 1 needs 3 hours of time on machine 1, 1 hour of time on machine 2, and 1 hour of time on machine 4,
- the production of one item of good 2 needs 2 hours of time on machine 1, 10 hours of time on machine 2, and 4 hours of time on machine 3, and
- the production of one item of good 3 needs 5 hours of time on machine 1, 1 hour of time on machine 2, 2 hours of time on machine 3, and 9 hours of time on machine 4.

When producing one item of good 1, the factory makes 10 currency units of profit. Likewise, for goods 2 and 3, the production makes 15 and 25 currency units of profit, respectively.

We ask the question what goods the factory should produce over the month in order to maximize profit. We assume all fixed costs to be independent of the choice of produced goods, so that they do not have to be considered. Furthermore, we are fine with fractional quantities, as partially completed goods can be carried over to the factory's production in the next month.

We model the problem as a linear programming problem over the variables $\mathcal{V} = \{x_1, x_2, x_3\}$, where x_i (for $i \in \{1, \ldots, 3\}$) denotes the monthly production of a good i. We use the following constraints:

$$\begin{array}{rcrcrcr}
3x_1 & + & 2x_2 & + & 5x_3 & \leqslant & 100 \\
1x_1 & + & 10x_2 & + & 1x_3 & \leqslant & 100 \\
 & + & 4x_2 & + & 2x_3 & \leqslant & 100 \\
1x_1 & & & + & 9x_3 & \leqslant & 95
\end{array}$$

The constraints enforce that no machine can be utilized for more than the available time per month.

The optimization function is $10x_1 + 15x_2 + 25x_3$ and represents the profit made (per month) from the production of x_1, x_2, and x_3 items of goods 1, 2, and 3, respectively.

We can write the LP problem in the format needed for the LP solver `lp_solve` in order to automatically deduce an optimal production plan. The resulting formalization looks as follows:

```
max: 10*x1 + 15*x2 + 25*x3;
3*x1 + 2*x2 + 5*x3 <= 100;
1*x1 + 10*x2 + 1*x3 <= 100;
4*x2 + 2*x3 <= 100;
1*x1 + 9*x3 <= 95;
```

The input format is self-explanatory and the output of `lp_solve` on this input instance is the following:

```
Value of objective function: 474.265

Actual values of the variables:
x1                         12.9412
x2                         7.79412
x3                         9.11765
```

5.3 Translating an LP problem into canonical form

In order to talk about how to solve the linear programming problem in the next section, it is helpful to translate linear programming problems into a canonical form first. This form is defined as follows:

Definition 11. *A linear program in canonical form is given by an ordered list of variables $\mathcal{V}$ of length n, an $m \times n$ matrix A for some value of $m \in \mathbb{N}$, an m-element column vector $\vec{b}$, and an n-element column vector $\vec{c}$. We represent valuations to $\mathcal{V}$ by column vectors with n elements. A valuation $\vec{x}$ is valid for the LP problem if all elements of $\vec{x}$ are $\geqslant 0$ and*

$$A\vec{x} \leqslant \vec{b}$$

holds, where $\leqslant$ performs an element-wise comparison of two (column) vectors. The quality of the solution is defined to be $\vec{x} \cdot (\vec{c})^T$.

Apart from the different representation of the optimization function, there are four differences between the general LP problem and an LP problem in canonical form:

1. Equations are no longer supported in the canonical form.
2. The comparison operator $\geqslant$ is no longer allowed, and only $\leqslant$ remains as an allowed comparison operator.
3. Solutions need to be non-strictly positive in all dimensions.
4. All constraints C are encoded into A and b.

None of these differences affect the generality of the LP problem, as we will discuss in the remainder of this section.

Let us start with the equality comparison operator. Disallowing it is no problem, as all equations of the form $f = b$ for some linear expression f and some constant b can be rewritten to a conjunction of two inequations $f \geqslant b$ and $f \leqslant b$. Disallowing the $\leqslant$ operator is also no problem, as $f \geqslant b$ is equivalent to $-f \leqslant -b$.

For the non-strict positiveness, consider a variable x that can have negative values. We can replace every occurrence of x with in the constraints by the expression $(x' - x'')$, where x' and x'' are new variables which do not need to be positive. Positive values for x can be simulated by setting $x' = x$ and $x'' = 0$, whereas negative values of x can be simulated by setting $x' = 0$ and $x'' = -x$. If we have a solution for the modified LP problem (with x' and x''), we can translate it to a solution for the original LP problem by setting $x = x' - x''$.

Finally, After performing the substitutions above for all constraints in C, they are all of the form $f \leqslant b$ for some linear function f and some constant b. Arranging all constraints into a matrix A and a column vector $\vec{b}$ is now trivial.

5.4 Solving the LP problem – the simplex algorithm

As in the preceding chapters, let us have a look at the problem of solving a linear program and analyze the key properties of one of the most commonly used algorithms for solving LP: the *simplex algorithm*.

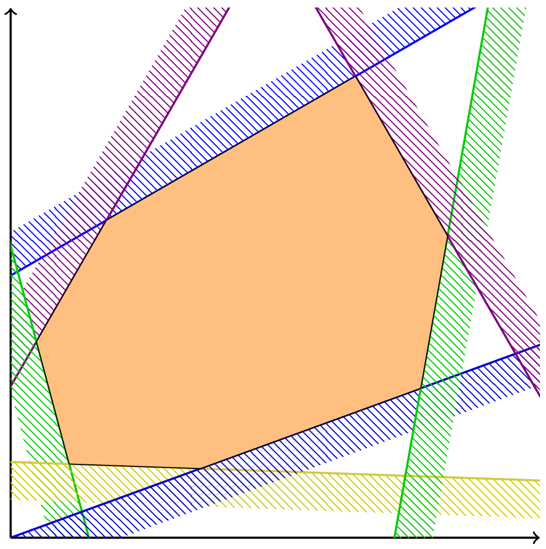


Figure 5.1: Graphical representation of set of constraints in linear programming.

5.4.1 Geometric interpretation of the LP problem

To make the description of the algorithm as intuitive as possible, we depict its main idea in figures, such as the one in Figure 5.1. We consider LP problems over two variables for this purpose, which allows to show the space of possible solutions as a two-dimensional drawing. We also assume that the LP problem is given in normal form. This implies that only solutions $\vec{x}$ in which all elements of $\vec{x}$ are non-strictly positive are admissible, which allows us to put the $\vec{0}$ point into the lower left corner of the figures.

Every constraint in an LP problem restricts the space of solutions to a half-space. As the constraints are linear, this means that for each constraint, there exists a plane in the space of solutions such that all solutions lie on one side of the plane (or on the plane itself). In Figure 5.1, which depicts a two-dimensional solution space, the planes are actually lines, and the sides of the planes that contain the points $\vec{x}$ that violate the constraint are marked by a hatched area.

When requiring all 7 constraints in Figure 5.1 to be satisfied, the area in the middle of the Figure represents the set of possible solutions to the LP problem instance. Any point in the set would be allowed as a solution, but since we are interested in optimal solutions, i.e., those that maximize $\vec{x} \cdot (\vec{c})^T$, we are interested in certain particular points. Figure 5.2 overlays the content of Figure 5.2 with arrows that all point into the same direction, i.e., the direction in the solution space that increases the quality of the solution.

As the arrows indicate the direction of optimization, all solution points that are on a line that is orthogonal to the arrows have the same quality. Figure 5.3 shows a couple of such lines. Lines that are further away from the origin represent better solutions. The figure also shows the optimal solution of the linear programming instance, which is the point in the top right of the solution space. Intuitively, we can obtain it by pushing the line as far into the direction of the arrow as possible and only stopping when it cannot be pushed any further without making it non-intersecting with the solution space.

The optimal point lies on the boundaries of the solution space in this example. We can prove that this is always the case (if $\vec{c} \neq \vec{0}$): if for some point $\vec{x}$, there is some $\epsilon > 0$ such that all points $\vec{y}$ that are not further away than ϵ from $\vec{x}$ (in terms of euclidean distance) are also solutions to the LP problem instance, then we know that $\vec{x}$ cannot be optimal. The optimization function is linear, so half of such points $\vec{y}$ have be better than $\vec{x}$, and half of them have to be worse. Since $\epsilon > 0$, at least one point has to be better than $\vec{x}$.

Without loss of generality, we can furthermore assume that if a set of constraints can be fulfilled (i.e., there exists a solution to an LP problem instance), the optimal points that we search for lie in *corners* of

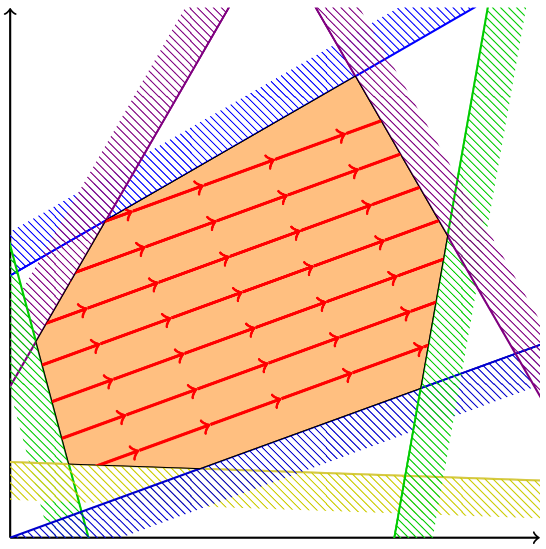


Figure 5.2: Graphical representation of a linear programming problem instance.

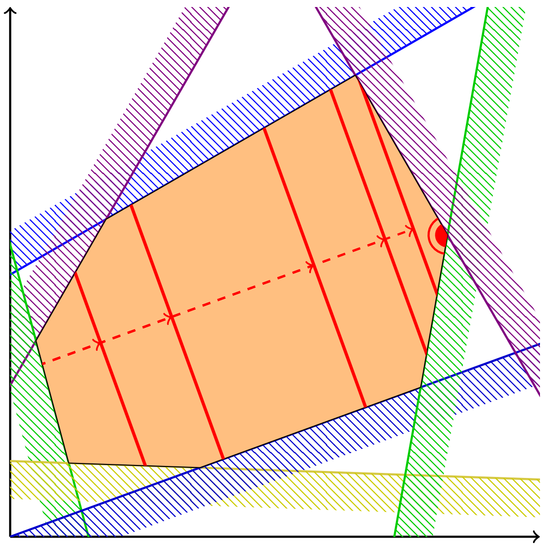


Figure 5.3: Graphical representation of solutions to a linear programming problem instance.

the solution space, i.e., are equivalent to one of the vertices of the *convex polytope* that has the constraints in the linear program as faces.

This fact can be derived from the linearity of the optimization function and the convexity of the polytope that represents the set of solutions. All optimal solutions lie on a plane, and if we move the plane by $\epsilon > 0$ into the direction of the optimization function, then it does not intersect with the solution points any more. If we now assume that no vertex of the solution space polytope is one the plane of optimal solutions, every vertex of the solution polytope is at least ϵ away from the plane for some $\epsilon > 0$, which contradicts this assumption. So at least one polytope vertex must lie on the plane.

5.4.2 Description of the simplex algorithm

In order to find an optimal solution to a linear programming instance, the properties of the solution spaces of LP problem instances imply that it suffices to compute the values of all vertices of the solution space polytope and to then select the best vertex. Unfortunately, this approach does not scale very well with the number of dimensions. Take for example a *hypercube*. For a number of dimensions n, such a

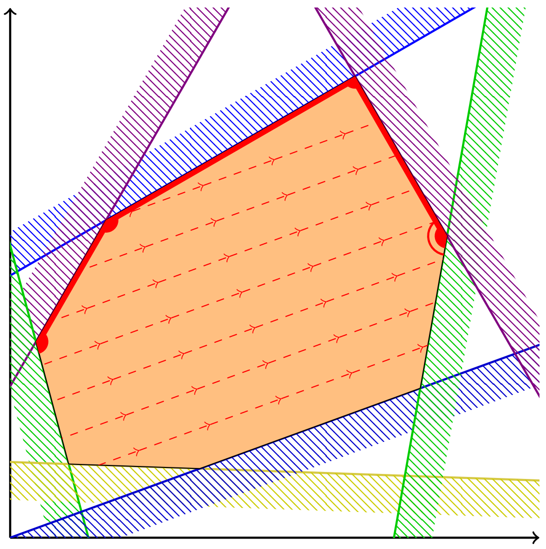


Figure 5.4: A graphical representation of a run of the simplex algorithm

hypercube has $2n$ faces, so $2n$ constraints suffice to write a linear program with the hypercube as solution space. The solution space however has 2^n vertices, leading to a lower bound on the computation time of 2^n of this LP solving approach.

The simplex algorithm (Dantzig, 1951) refines this idea by performing the search for an optimal solution in a smarter way. Starting from some vertex of the solution space polytope, it checks the adjacent edges if they lead to vertices that are better solutions. Whenever a better solution is found, the search for a best solutions continues from the new vertex. If at some point a vertex is reached that does not have a better neighbor, then by the convexity of the solution space, we know that it is globally optimal.

Figure 5.4 depicts an example search path through the solution space taken by the algorithm. Starting in the left-most point of the search space, it takes the *upper* path along the search space boundaries, and eventually arrives at the right-most point of the workspace, which we already know to be optional.

5.4.3 Complexity of the simplex algorithm

The computation time of the simplex algorithm depends heavily on the order in which edges of the solutions space polytope are evaluated and how the initial vertex is chosen. Yet, in our simple hypercube example from above, using the algorithm leads to an improvement that is relatively independent of the order and the initial vertex.

Assume that the algorithm starts in the vertex with the worst value. Regardless of which path it follows from there, whenever for some point $\vec{x} = (x_1, \ldots, x_n)$, some value of x_i is changed along the path, it is never changed back. This follows from the linearity of the optimization function. Thus, the path consists of at most n points, and for every step, at most n neighbour vertices are evaluated. Thus, overall, n^2 vertices are visited, which is much better than the number of 2^n that we had for the naive algorithm.

While the simplex algorithm was observed to work well on many practical LP instances relatively quickly after its first publication, its actual computational complexity remained unknown for many years. Klee and Minty (1972) later found out that there exist problem instances for which the algorithm can require time exponential in the number of dimensions and constraints in order to run. In a nutshell, the reason is that solution space polytopes can have strictly improving paths of length exponential in the number of dimensions and in the number of constraints along their edges. For the scope of this section, let us call such solution spaces *difficult*. Let us study difficult solutions spaces by means of examples for the case of two-dimensional and three-dimensional LP problems.

Figure 5.5 shows a simple two-dimensional solution space (for a non-normalized LP problem), generated by four linear constraints. Starting from the lower left corner, there are only two paths along the edges of

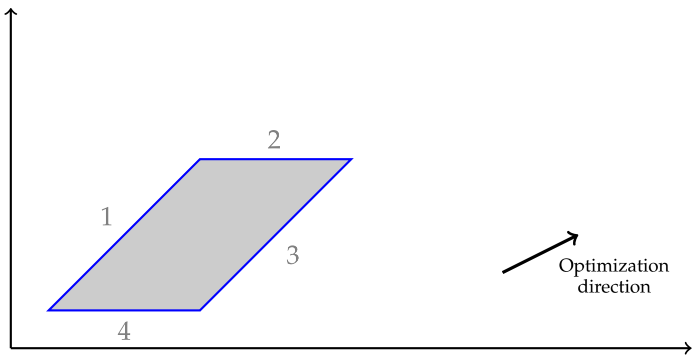


Figure 5.5: A simple convex two-dimensional solution space induced by a linear programming problem. The edges of the solution space are numbered.

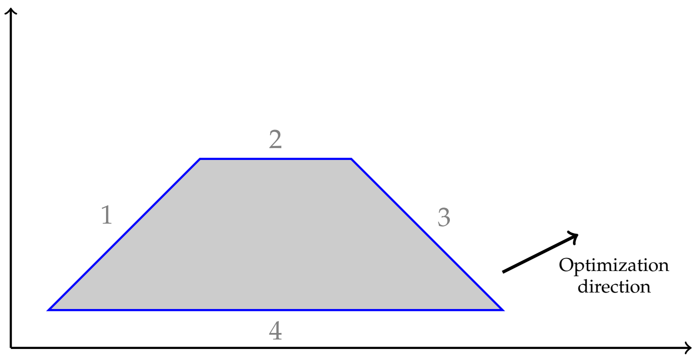


Figure 5.6: Another simple convex two-dimensional solution space induced by a linear programming problem.

the solution space that are strictly increasing in terms of the optimization function and cannot be further extended without losing this property: the path can either take edge 1 and edge 2, or alternatively take edge 4 and then edge 3. In both cases, three vertices are visited overall. Starting the path from another node, it can only be shorter. So the simplex algorithm can only visit at most three vertices, and not all of the four vertices. The solution space is thus relatively well-behaved and well-suited for the simplex algorithm.

However, we only need to alter the solution space a bit to remove this property. Figure 5.5 shows a simple two-dimensional solution space, generated by four linear constraints. Starting from the lower left corner of the solution space, there is a strictly increasing path that visits all four vertices. The path uses the edges 1, 2, and 3. Thus, the four-corner polytope represents a difficult solution space. We can generalize this example further by adding another dimension. We add two linear constraints to account for the added dimension, and the number of vertices of the solution space polytope doubles to 8 with the added dimension. Figure 5.7 depicts the new three-dimensional version of the example solution space. In it, the trapezoid shape from Figure 5.6 is used for all of its faces. In order to allow the combination of these trapezoid shapes, they have been slightly distorted. To show this distortion in a more clear way, Figure 5.8 shows an orthographic projection[1] of the solution space in Figure 5.7. It is roughly the view of the object in Figure 5.7 from above.

The solution space has a path along its edges that is strictly increasing and visits all of the vertices. It is also shown in the figures.

Solutions spaces such as the ones in Figure 5.6 and Figure 5.7 can also be constructed for higher dimensions. As the number of vertices of the solution space polytope grows exponential with the number of dimensions, the lengths of the longest strictly increasing paths grow exponentially with the

[1]In an orthographic projection, one or more dimensions are simply discarded. In the projection used here, three-dimensional coordinates of the shape (x, y, z) are mapped two-dimensional coordinates (x, y).

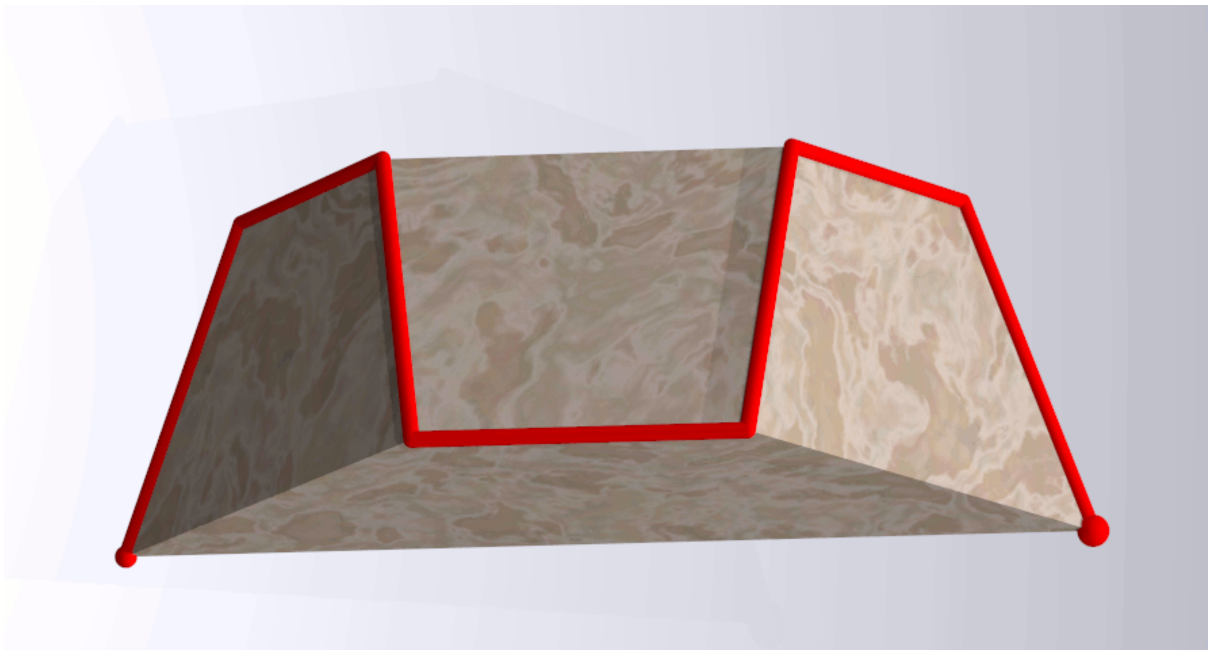


Figure 5.7: A (raytraced) three-dimensional solution space. The direction of the optimization function is (roughly) to the right.

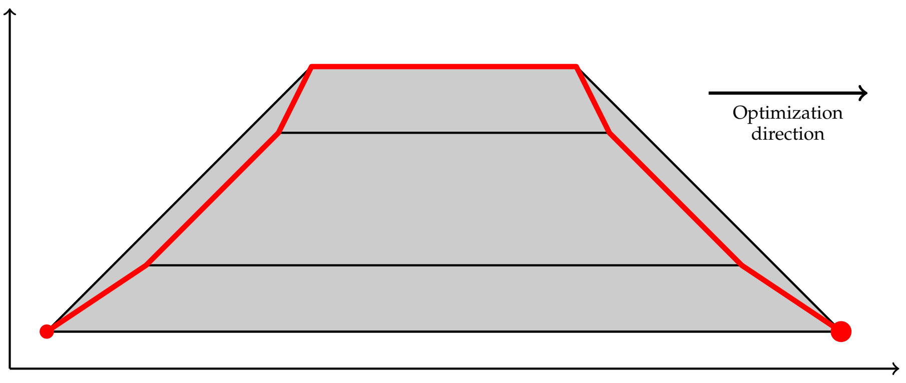


Figure 5.8: Orthographic projection of the solution space from Figure 5.7.

number of dimensions. At the same time, the number of constraints grow only polynomially. Thus, it the simplex algorithm starts in the worst vertex (or close to it) and always follows the "wrong" edges, it has an exponential computation time.

Depending on the edge selection and initial vertex selection heuristics used, the linear programming problems that induce the solution spaces in Figure 5.6 and Figure 5.7 are actually quickly solved. However, for the commonly used such heuristics, families of LP instances with exponentially long such paths that can be chosen by the heuristic have been found, leading to exponential worst-case computation times of the algorithm (see, e.g., Klee and Minty, 1972; Avis and Chvátal, 1978; Roos, 1990). The families of problems used to show this are more complicated than the ones above. Because linear programming is only one computational problem that we consider in this course, we will not dive into the intricacies of these results.

5.4.4 Some remarks

Missing details: It should be noted that despite the simplicity of the simplex method's main idea, there were a lot of details that we skipped over in the description above. For example, we did not discuss

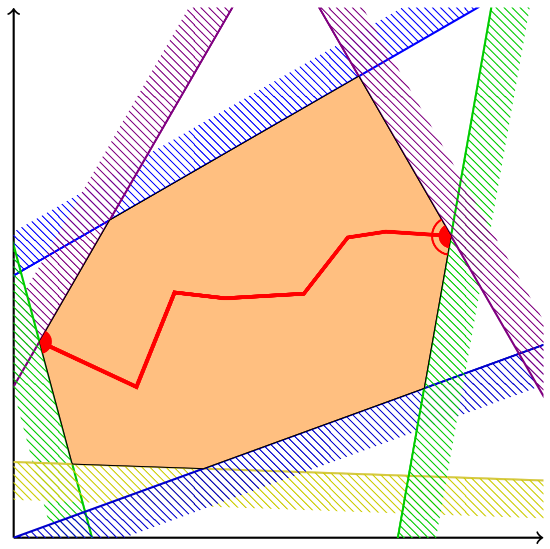


Figure 5.9: Geometric interpretation of interior-point methods to LP solving.

how to obtain a starting vertex for the algorithm or how to compute a vertex's neighboring vertices in the solution space polytope.

Complexity of the LP problem: Despite its exponential worst-case computation time, the simplex method is widely used to solve linear programs in practice. We have already seen for the case of satisfiability (SAT) solving that exponential-time algorithms can be useful in practice.

There is however also a big difference between the simplex algorithm and CDCL-style solvers for SAT: while for the latter problem, no polynomial-time algorithm is known, and hence using an exponential-time algorithm seems to be reasonable, the LP problem actually *has* a polynomial-time algorithm. Yet, because the simplex method is observed to perform well it practice, it is still often used.

One of the first polynomial-time algorithm for the LP problem has been given by Karmarkar (1984). It belongs to the class of *interior-point methods* that do not traverse the edges of the solution space polytope. Rather, they compute a solution by exploring the solution space itself. Figure 5.9 depicts this concept graphically.

Karmarkar (1984) gives the following characterization of the algorithm's computation time:

> *In the worst case, the algorithm requires $O(n^{3.5}L)$ arithmetic operations on $O(L)$ bit numbers, where n is the number of variables and L is the number of bits in the input.*

Thus, Karmarkar's method is actually quite fast, even when the elements of A, $\vec{b}$, and $\vec{c}$ need many bits to be represented with a high precision. However, since it has been observed that the simplex algorithm tends to work well on practical problems (see, e.g., Terlaky and Zhang, 1993, p.204), it is still often applied instead of Karmarkar's algorithm (or other polynomial-time linear programming algorithms) despite its exponential worst-case time complexity.

Unbounded solutions spaces and trivial optimization functions: In the description of the simplex algorithm, we assumed that the solution space is always bounded, i.e., there do not exist some vectors $\vec{x}$ and $\vec{y}$ such that for every $r \in \mathbb{R}_{\geqslant 0}$, we have that $\vec{x} + r \cdot \vec{y}$ is a correct solution. Practical LP solvers such as lp_solve can typically detect such instances and report this to the user. For optimization functions $\vec{c}$ for which $\vec{y} \cdot (\vec{c})^T > 0$, the unboundedness of the solution space actually implies that there does not exist an optimal solution (as for every solution, there is always a better one for some $r \in \mathbb{R}$. Thus, LP solving makes little sense in this case.

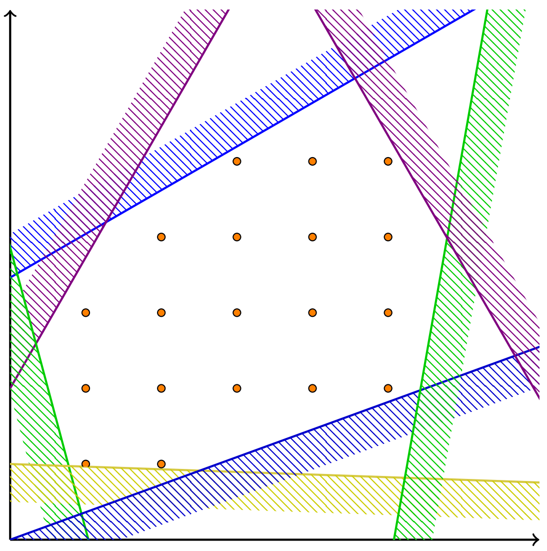


Figure 5.10: Graphical representation of the solution space of an ILP problem.

We also assumed that the optimization function vector $\vec{c}$ is not equal to $\vec{0}$, as otherwise there is nothing to optimize. This assumption was made without loss of generality, as in such a case, all solutions have the same quality, so that any other optimization function vector can be used.

Numerics: In practical LP solving, the rounding errors that occur when using the floating point unit to perform the necessary calculations are a problem. Modern LP solving tool apply techniques to mitigate this problem, so that the user of an LP solving tool normally does not need to take rounding errors into account.

5.5 Integer linear programming

In some problems, we can naturally describe the constraints on the solution space as linear inequalities while we are only interested in solutions that are completely integer-valued, i.e., where all components of a solution vector are integer numbers. We call such solutions *integral* and Figure 5.10 depicts the concept graphically. We use the same constraints as in Figure 5.1, and use the same color to depict the set of solutions. Because solutions are required to be integer-valued, there are only a few points that represent possible solutions.

We call linear-programming problems where all elements of a solution vector need to be integral *integer linear programming* (ILP) problems. In this section, we discuss an application of ILP solving and an algorithm to solve ILP problems.

5.5.1 Example application: processor frequency scheduling

Assume that we have a processor in an embedded system. In every time slot of T milliseconds, it needs to execute a workload of d computation cycles. The processor has a couple of clock speed settings available. At lower speed values, it consumes less energy, but also takes longer to perform the tasks. We now want to find a *clock speed schedule* $x_1, \ldots, x_n$ that states how many clock cycles at each of n speed settings should be executed by the processor such that after at most T time units, d cycles are executed, and the overall energy consumption for the d cycles is minimized.

The clock rates of the possible speed settings are given as frequencies $c_1, \ldots, c_n$, while the energy consumption per clock cycle is given as values $e_1, \ldots, e_n$. We need $n + 2$ constraints to obtain an ILP problem for finding an energy-optimal clock speed schedule. The first n of these constraints simply enforce that $x_i \geqslant 0$ for all $i \in \{1, \ldots, n\}$. When using a linear programming tool that does not support negative values in the solution vector, these constraints can be ommitted. The last two constraints ensure that the schedule contains at last d computation cycles, i.e., we have

$$x_1 + x_2 + \ldots + x_n \geqslant d,$$

and that all computation cycles in the schedule are executed before T time units have passed:

$$x_1 \cdot \frac{1}{c_1} + x_2 \cdot \frac{1}{c_2} + \ldots + x_n \cdot \frac{1}{c_n} \leqslant T.$$

Note that the later of these constraints is a linear constraint as $c_1, \ldots, c_n$ are constants. Also, note that despite the fact that in ILP solving, we are only interested in integer-valued solutions, there is no requirement for the constants appearing in the constraints and optimization function to be integer-valued.

As optimization criterion in our example problem, we want to minimize $e_1 \cdot x_1 + e_2 \cdot x_2 + \ldots + e_n \cdot x_n$, as it represents the overall energy needed for all of the clock cycles.

A more sophisticated version of this problem can be found in Marwedel (2003).

5.5.2 ILP Solving

Literature: *Kroening and Strichman, 2008, Section 5.3*

Despite the similarity of the ILP problem to the LP problem, there is a substantial difference in complexity. While LP solving can be performed in polynomial time, ILP solving is known to to be NP-complete.

This can be shown by a simple reduction from SAT to ILP solving: for every variable in the SAT problem, we introduce an integer variable and bound it to values in the range $\{0, 1\}$ by two simple linear constraints. Then, we translate every clause in a SAT instance to an ILP constraint. For example, the clause $\neg x_1 \vee x_2 \vee x_3$ is translated to $(1 - x_1) + x_2 + (1 - x_3) \geqslant 1$, which is equivalent to $-x_1 + x_2 - x_3 \geqslant -1$. The space of solutions of the resulting ILP problem is the same as for the original SAT problem, when interpreting 0 as **false** and 1 as **true**.

So we cannot assume that we are able to solve ILP problem instances in polynomial time. For problems in which all variables are bounded (i.e., for all variables x, we have constraints of the form $x \geqslant b$ and $x \leqslant c$ for some $b, c \in \mathbb{R}$), we could reduce ILP solving to pseudo-boolean constraint solving (see Chapter 3.1). However, by doing so, the dimensionality of the problem increases a lot, so it makes sense to check if we can take advantage of the properties of ILP instances directly.

Probably the most important property that the ILP problem inherits from the LP problem is that the solution spaces are always *convex*. This means that whenever there is a solution of the form $(x_1, x_2, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$ and a solution of the form $(x_1, x_2, \ldots, x_{j-1}, x'_j, x_{j+1}, \ldots, x_n)$ with $x_j < x'_j$, then $(x_1, x_2, \ldots, x_{j-1}, y_j, x_{j+1}, \ldots, x_n)$ is also a solution for any $y_j \in \{x_j, x_j + 1, \ldots, x'_j\}$. While the SAT problem (to which we could reduce bounded ILP instances) also has this property, it is hard to take advantage of it, as there are only two values for every dimension. So it seems to be worthwhile to consider solving an ILP problem instance in a more direct way as opposed to reducing it to a problem such as pseudo-boolean constraint solving that has less structure, where the number of dimensions of the solution is higher, and where it is more difficult to take advantage of the solution space's properties.

A commonly known approach approach to solve the ILP problem is to apply a *branch-and-bound* procedure, as described in Algorithm 5. The function B-And-B-ILP has four parameters: (1) the set of variables $\mathcal{V}$, (2) the set of constraints P, (3) the optimization function f, and (4) the value of the currently best know solution o. The function returns a tuple consisting of the best solution found (or $\emptyset$, which denotes that no solution has been found) and its value. The value of o is forwarded during the recursive calls performed by the algorithm. When calling the algorithm without already having a solution, o should be set to $-\infty$.

The main idea of the algorithm is to use a regular linear programming algorithm as an *oracle*. Whenever the LP algorithm returns a solution that is already integral, function function B-And-B-ILP returns this

Algorithm 5 A branch-and-bound algorithm for ILP solving.

```
function B-And-B-ILP($\mathcal{V}, P, f, o$)
    $(\vec{s}, q)$ = SolveLP($\mathcal{V}, P, f$)
    if $q \leqslant o$ then
        return $(\varnothing, -\infty)$
    else if $\vec{s}$ is integral then
        return $(\vec{s}, q)$
    else
        select $v \in \mathcal{V}$ such that $\vec{s}(v)$ is non-integral
        $(\vec{s}', q')$ :=B-And-B-ILP($\mathcal{V}$,$P \cup \{v \geqslant \lceil \vec{s}(v) \rceil\}, f, o$)
        $(\vec{s}'', q'')$ :=B-And-B-ILP($\mathcal{V}$,$P \cup \{v \leqslant \lfloor \vec{s}(v) \rfloor\}, f, \max(o, q')$)
        if $q' > q''$ then
            return $(s', q')$
        else
            return $(s'', q'')$
        end if
    end if
end function
```

solution. Otherwise, it tries to *fix* the solution by adding additional constraints. It does so by selecting a variable v with non-integral value $\vec{s}(v)$ (as returned by the LP solver) and trying to bound it from below and above to $\lceil \vec{s}(v) \rceil$. In each of these cases, the function B-Band-B-ILP is called recursively to check if there is a solution that satisfies the constraints P and the additional added constraint. The function returns the solution found with the higher value.

To speed up the computation, the function always maintains the quality of the best known solution so far. If the LP solving step produces a (integral or non-integral) solution that is worse than the best known solution so far, the algorithm immediately terminates. Also, in Line 10, the best known solution value is updated by the result of the recursive call one line earlier.

The algorithm uses the convexity of the solution only indirectly: it relies on the fact that calling the classical LP solver in Line 2 is not a costly operation. Efficient LP algorithms in turn use the convexity of the solution space.

The necessity of executing both Lines 9 and 10 may come as a slight surprise: intuitively, it should be sufficient to only look at only one side of the $v = \lceil \vec{s}(v) \rceil$ hyperplane, namely the one that maximizes the optimization function. If a solution is found there, then one may assume that it is the optimal one, as on that side of the hyperplane, we can find the optimal real-valued solutions. In the integer-case, however, this observation breaks down, as the following example shows. Figure 5.11 shows the solution space for an ILP problem with two linear constraints. To keep the figure clean, the shaded region denotes all solutions of the underlying LP problem (i.e., the same problem without the requirement that solutions must be integral). All ILP solutions are represented by the dots in the shaded region.

When the branch-and-bound algorithm starts, it computes the non-integral solution that is represented by the marked point. This solution is non-integral in both dimensions. The algorithm then recurses with an additional constraint that the variable that is drawn along the X axis in Figure 5.11 should have a value $\geqslant 5$. This is shown in the figure by the additional line with the arrow. During the recursive call, there is only one integral point left in the solution space, so the algorithm computes point a. However, there is also point b, which is a better solution. So the only way to find point b (or any of the other three solutions with the same value) is to also consider the case that the value of the variable along the X axis is $\leqslant 4$ in the branch-and-bound algorithm. So the recursive call in Line 10 is necessary even if the call in Line 9 yields a solution.

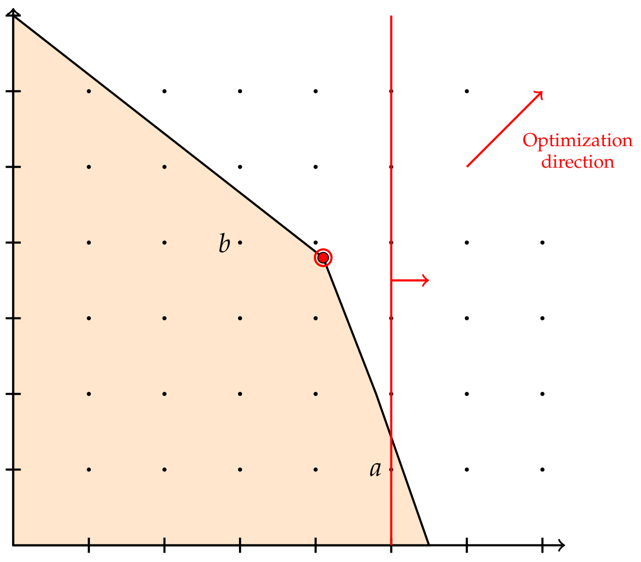


Figure 5.11: An Example for ILP Solving.

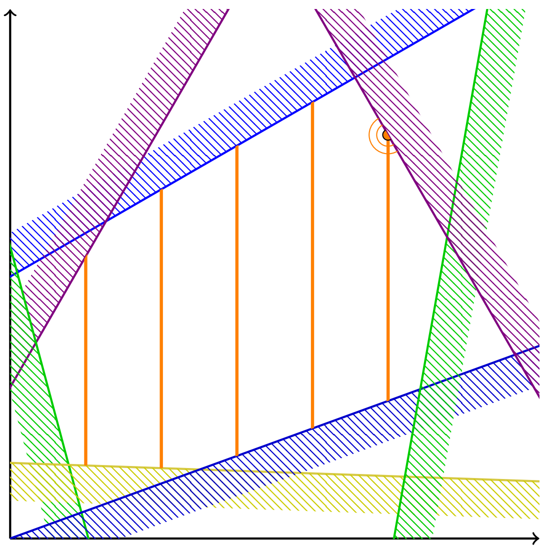


Figure 5.12: The solution space of an MILP problem.

5.6 Mixed-integer linear programming

In addition to the linear programming (LP) and integer linear programming (ILP) problems, there is also the concept of *mixed-integer linear programming* (MILP), which is a mixture of these two. In solutions to MILP problem instances, some variables must have integral values, whereas others can have arbitrary real values. Which variables must be integral is part of the problem instance.

Figure 5.12 examplifies the solution space of an MILP problem graphically. We use the same linear constraints as in Figure 5.1. The variable along the X axis must have an integral value in this case. The figure also shows the optimal solution if the optimization function has strictly positive factors for both dimensions.

To solve an MILP problem, the branch-and-bound algorithm from the preceding section can be used with minor modifications: it only needs to be ensured that in Line 8 of the algorithm, all variables that do not need to have an integral value are ignored and never selected.

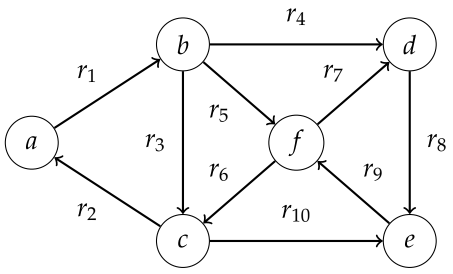


Figure 5.13: A map with depots and routes for Example 20.

Example 20. To study the usefulness of MILP solving in practice, let us consider an example application. A juice production company wants to distribute its juice. It has a couple of distribution depots $D = \{a, b, c, d, e, f\}$ for which it can hire a transportation agency for delivering **full truckloads** between the depots. The agency has a fixed set of routes $R = \{r_1, \ldots, r_{10}\}$. Figure 5.13 shows a simplified maps with the depots and the routes. At every depot $x \in D$, we have a *production* of p_x gallons of juice. Some of these values are positive (if the depot is adjacent to the production plant), while some are negative (if they sell juice to the end consumer or to supermarkets that are not managed by the juice company).

We have given cost values $c_1, \ldots, c_{10}$ per truck per route. Every truck can transport up to 6000 gallons of juice. The question that we want to solve is how to minimize the overall cost for all the trucks needed to transport juice in the network. In particular, we want to ensure that all depots d with $p_d < 0$ receive p_d gallons of juice, and all depots d with $p_d > 0$ send away p_d gallons of juice. A truck may have a load of between 0 and 6000 gallons of juice, but the price for the truck trip is the same regardless of the truck's utilization.

To model the problem as an ILP instance, we use integer variables $t_1, \ldots t_{10}$ that denote the numbers of **truckloads** to be transported on the routes 1 to 10. As not all truckloads need to be full, we also use real-valued variables $g_1, \ldots, g_{10}$ that encode how many gallons are actually transported along a route. The following constraints connect $t_1, \ldots t_{10}$ and $g_1, \ldots, g_{10}$:

$$\bigwedge_{i \in \{1,\ldots,10\}} +g_i - 6000t_i \leqslant 0$$

All variables $t_1, \ldots t_{10}, g_1, \ldots, g_{10}$ are furthermore bounded below by 0. To encode that at all depots, the incoming and outgoing flows should level out, we use the following constraints:

$$\begin{aligned}
&p_a - g_1 + g_2 = 0 \\
&p_b + g_1 - g_3 - g_4 - g_5 = 0 \\
&\ldots \\
&p_f + g_5 + g_9 - p_7 - p_6 = 0
\end{aligned}$$

Finally, we use the overall cost as optimization criterion:

$$f = -c_1 t_1 - c_2 t_2 - \ldots - c_{10} t_{10}$$

As in LP solving, the value of the solution is to be maximized, the cost is given negatively. ★

5.7 Conclusion

In this chapter, we discussed linear programming (LP), integer linear programming (ILP), and mixed-integer linear programming (MILP) as computational problems into which many interesting practical problems can be encoded. All of these problems are well-established in the areas of *operations research* and *mathematical optimization*, where most research on them is performed. We introduced the simplex algorithm as a computational engine for solving LP problems. This algorithm is commonly applied in practice despite its exponential worst-case computation time. Yet, it is observed to often work well in practice, and a theoretical analysis of mean computation times of the algorithm on LP problems from a reasonable probability distribution showed that it runs in expected polynomial time (Shamir, 1993).

To keep the presentation concise, we had to skip over many details, such as how to obtain initial solutions in the simplex algorithm, how to compute the edges of the solution space polyhedron in the simplex algorithm, and how to select a good variable to branch on in the branch and bound algorithm. Yet, we gave the basic intuitions on what makes these methods work in practice and showed the limits of the approaches (by giving an intuition on how to construct LP instances in which the simplex algorithm can run for an exponential amount of time and explaining why the branch-and-bound algorithm has to perform many branching operations).

It should also be noted that there are many more variations of the linear programming paradigm that also have efficient computational engines and are suitable for some practical applications. In *quadratic programming*, optimization functions can not only be linear, but also have quadratic terms. The problem is NP-complete, but can be solved in polynomial time for some interesting sub-classes of optimization functions. In *semidefinite programming*, the objective function is linear, but quadratic terms are allowed in the constraints. However, when writing the terms in a constraint into a matrix, the matrix has to be semidefinite. These extensions are beyond the scope of this course.

Chapter 6

Network Flow

Much of the power of the Maximum-Flow Problem has essentially nothing to do with the fact that it models traffic in a network. Rather, it lies in the fact that many problems with a nontrivial combinatorial search component can be solved in polynomial time because they can be reduced to the problem of finding a maximum flow or a minimum cut in a directed graph.

Kleinberg and Tardos (2006), Chapter 7.7

Among the problems considered in these lecture notes, the maximum-flow problem, which we will consider next, is a special case: a polynomial-time algorithm is known for this problem and unlike in LP solving, using a worst-case exponential-time algorithm for this problem is unheard of.

Yet, maximum-flow fits very well into the set of problems considered so far. There are many practical problems that we can encode as maximum-flow problem instances, so that algorithms for solving them can be used as computational engines. Furthermore, the *Ford-Fulkerson algorithm*, which we will use to solve network flow problems, has very interesting properties, which makes studying it worthwhile.

The Ford-Fulkerson algorithm differs from the computational engines that we discussed earlier by being seldomly applied as an external tool. Rather, it tends to be integrated tightly in applications that can make use of it in practice. This fact also motivates looking at the algorithm at a level of detail that is sufficient for reimplementing it.

6.1 Problem definition

Literature: *Kleinberg and Tardos, 2006, Section 7.1*

In a network flow problem, we are given a graph $G = (V, E)$, where V is the (finite) set of *nodes* of G (also called the set of *vertices*), and $E \subseteq V \times V$ is a set of *edges*. We assume that there are designated nodes s and t in V. The node s is called the *source node*, and t is called the *sink node*. There are no edges to s (i.e., we have that $(V \times \{s\}) \cap E = \emptyset$) and no edges from t (i.e., we have that $(\{t\} \times V) \cap E = \emptyset$). All notes from $V \backslash \{s, t\}$ are called *internal nodes* of V. The graph is also augmented by a *capacity function* $c : E \rightarrow \mathbb{R}_{\geqslant 0}$, which assigns to each edge a *capacity*.

We call a tuple (G, c) a *flow network* if G is a graphs that has the structure described above, and c is a capacity function.

In the network flow problem, we are searching for a *flow*. A flow is a function $f : E \rightarrow \mathbb{R}_{\geqslant 0}$ that satisfies two conditions:

- for every $e \in E$, we have $f(e) \leqslant c(e)$, and
- for every $v \in V \backslash \{s, t\}$, we have $\sum_{e \in (E \cap (\{v\} \times V))} f(e) - \sum_{e \in (E \cap (V \times \{v\}))} f(e) = 0$.

Given a node v, we call $\sum_{e \in (E \cap (\{v\} \times V))} f(e)$ the *sum of the outgoing flows of* v, while $\sum_{e \in (E \cap (V \times \{v\}))} f(e)$ is the *sum of the incoming flows of* v.

To define the *maximum-flow problem* as a particular network flow problem below, we need the following lemma:

Lemma 1. *In every flow f, the sum of the outgoing flows of s is the same as the sum of the incoming flows of t.*

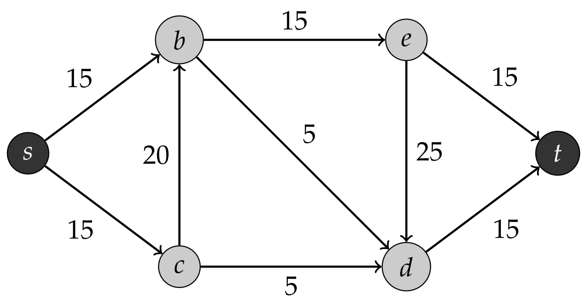


Figure 6.1: An graph with capacities that we use as example for the network flow problem.

Proof. All edges start in some node and end in some node. We know that

$$\sum_{v \in V} \left(\sum_{e \in (E \cap (\{v\} \times V))} f(e) - \sum_{e \in (E \cap (V \times \{v\}))} f(e) \right) = 0 \tag{6.1}$$

as the flow along every edge occurs exactly once positively and once negatively. For all inner nodes v, we know that $\sum_{e \in (E \cap (\{v\} \times V))} f(e) - \sum_{e \in (E \cap (V \times \{v\}))} f(e) = 0$ by the definition of a flow. Thus, we can simplify Equation 6.1 to:

$$\sum_{v \in V \setminus \{s,t\}} \left(\sum_{e \in (E \cap (\{v\} \times V))} f(e) - \sum_{e \in (E \cap (V \times \{v\}))} f(e) \right) = 0 \tag{6.2}$$

As we furthermore know that by definition, s has no incoming edges and t has no outgoing edges, we can furthermore simplify this equation to

$$\sum_{e \in (E \cap (\{s\} \times V))} f(e) - \sum_{e \in (E \cap (V \times \{t\}))} f(e) = 0, \tag{6.3}$$

which is exactly what was to be proven. □

We define the *value* $v(f)$ of a flow f to be the sum of the incoming flows of t, which we now know to be equivalent to the sum of the outgoing flows of s in f. In the *maximum-flow problem*, we search for a flow with the highest possible value.

Example 21. Let us consider the example flow network depicted in Figure 6.1. It has the set of nodes $V = \{s, b, c, d, e, t\}$, and the capacities of the edges are written as labels along the edges. There are no incoming edges for node s and no outgoing edges of node t, so the network fulfils the requirement outlined above.

Figure 6.2 depicts a flow for the network. It satisfies all of the constraints of a flow. In particular, none of the flows along some edge exceed the available capacity along the edge. Also, the incoming flows of all vertices in $\{b, c, d, e\}$ (the inner vertices) are the same as their respective outgoing flows. For example, b has an incoming flow of 15 (from s) and an outgoing flow of 15 (to d and e). The value of the flow is 20: we have $f((s,b)) = 15$, $f((s,c)) = 5$, and there are no other edges that start in s. By Lemma 1, we know that the sum of the flows into t then needs to be 20, which we can easily verify by observing that $f((e,t)) = 10$ and $f((d,t)) = 10$.

Figure 6.3 depicts another flow for the same network that has a higher value. While the flow from Figure 6.2 has a value of 20, the flow in Figure 6.3 has a value of 25. This shows that the flow from Figure 6.2 is **not** a solution to the maximum-flow problem. ★

It is easy to see that the value of a *maximal flow*, i.e., a flow that is a solution to the maximum-flow problem, cannot be higher than the sum of the capacities of all outgoing edges of s or higher than the sum of the capacities of all incoming edges of t. We call the smaller of these values the *trivial upper bound*

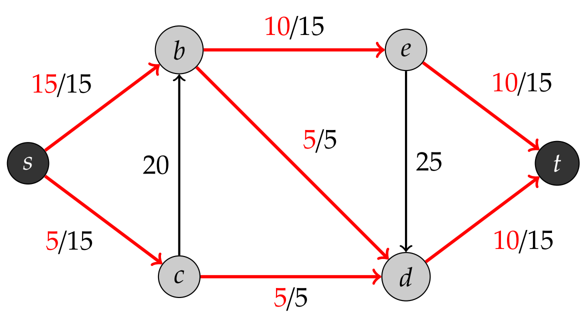


Figure 6.2: An flow network with capacities and a flow on the network. Edges that are used in the flow are colored and are labeled by the capacity used for the flow and the overall capacity of the edge.

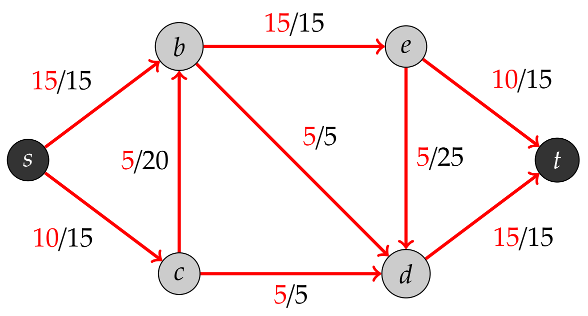


Figure 6.3: An graph with capacities and a flow on the graph.

on the flow of the network. Thus, if we had a flow with a value of 25 for the graph/capacity combination from Figure 6.1, we would already know that the flow is a solution to the maximum-flow problem. However, not even the flow from Figure 6.3 has this property, so we do not know at this point whether that flow is maximal or not.

6.2 Example application: Bipartite matching

Literature: *Kleinberg and Tardos, 2006, Section 7.5*

Let us discuss the usefulness of solvers for the maximum-flow problem as computational engines by reducing the *bipartite maching* problem to the maximum-flow problem.

Definition 12 (Bipartite matching). *Let K and L be sets and $\delta \subseteq K \times L$ be a* compatibility relation *between K and L. A* matching *between K and L is a partial function $g : K \rightharpoonup L$ such that for all $k \in K$ for which $g(k)$ is defined, we have $(k, g(k)) \in \delta$, and for which for no $k, k' \in K$, we have $g(k) = g(k')$. We aim for matchings for which the size of the domain of g is maximal, which we call* maximal matchings.

Let (K, L, δ) be a bipartite matching problem. We reduce it to a network flow problem (G, c) as follows:

- For the graph $G = (V, E)$, we set:
  - $V = \{s, t\} \uplus K \uplus L$
  - $E = (\{s\} \times K) \cup \delta \cup (L \times \{t\})$
- For all $e \in E$, we set $c(e) = 1$.

As an example, consider the bipartite matching problem with $K = \{a, b, c, d, e\}$, $L = \{1, 2, 3, 4, 5\}$, and $\delta = \{(a, 1), (a, 2), (b, 2), (b, 3), (b, 4), (c, 1), (c, 5), (d, 4), (d, 5), (e, 3), (e, 4)\}$. The flow network compiled from this problem is shown in Figure 6.4. As all edges have a capacity of 1, the capacities are not shown in the graph.

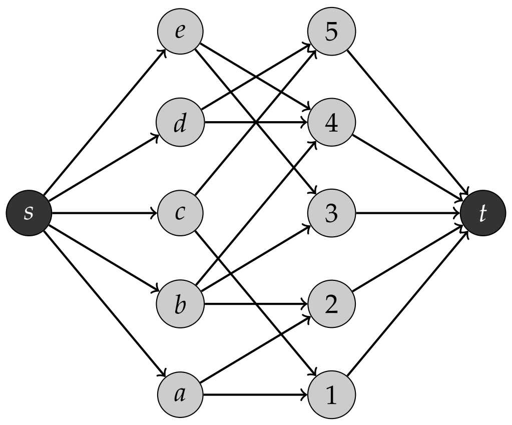


Figure 6.4: An example flow network that encodes a bipartite matching problem

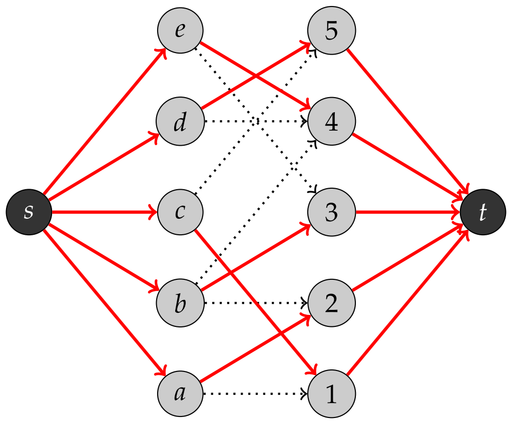


Figure 6.5: A maximal flow for the network given in Figure 6.4

The core idea of the reduction is captured in the following lemma:

Lemma 2. *Let (K, L, δ) be a bipartite matching problem and (G, c) be the maximum-flow problem derived from (K, L, δ) by the construction from above.*

There is a bijection between maximal matchings and maximum flows in (G, c) in which the flow along all edges is integer-valued.

Proof. From every bipartite matching g with domain size n, we can build a flow f for (G, c) with value n as follows:

- For all $k \in K$, we set $f((s,k)) = 1$ if $g(k)$ is defined, and set $f((s,k)) = 0$ otherwise.
- For all $l \in L$, we set $f((l,t)) = 1$ if $g(k) = l$ for some $k \in K$, and set $f((b,l)) = 0$ otherwise.
- For all $(k,l) \in \delta$, we set $f((k,l)) = 1$ if $g(k) = l$, and set $f((k,l)) = 0$ otherwise.

Is is not difficult to see that f is a valid flow, is integer-valued for every edge, and has value n.

For the converse direction, assume that f is an integer-valued flow with value n. As every edge has a capacity of 1, this means that there are n edges with flow that start in s and there are n edges with flow that end in t. Since there are no edges from nodes in L to nodes in K, this implies that all edges between K and L have distinct starting vertices and distinct target vertices. There have to be exactly n of these with a flow > 0, and by the construction of the network, they are all in δ. Thus, we have found a matching g where the domain of g has size n. $\square$

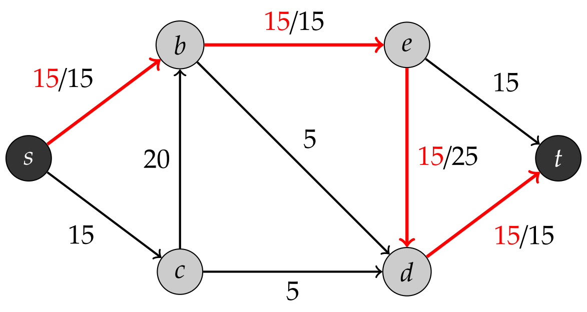


Figure 6.6: A non-maximal flow

Figure 6.5 shows a maximum flow for the network from Figure 6.4.

Note that the assumption that all flows along edges are integer-valued is crucial for the reduction to make sense. Thus, in order to effectively reduce the bipartite matching problem to the maximum-flow problem, we need a maximum-flow algorithm that guarantees the flow to be integer-valued whenever a maximum flow exists that is integer-valued. The algorithm for solving the maximum flow problem that we discuss in the next section has this property, so the this assumption is reasonable.

6.3 The Ford-Fulkerson algorithm

***Literature:** Kleinberg and Tardos, 2006, Section 7.1-7.2*

After motivating the maximum-flow problem, we now want to derive an algorithm for solving it. This section deals with a classical algorithm for the maximum-flow problem that is due to Ford and Fulkerson. We will derive the main ideas behind the algorithm step by step. In a sense, the Ford-Fulkerson algorithm is a smart version of a greedy optimization algorithm. The route that we will follow in this section is to first try to solve the maximum flow problem in a very basic greedy way, check where the approach breaks down, and then derive how we have to adapt the algorithm in order to circumvent the drawbacks of the very basic greedy algorithm.

6.3.1 A basic greedy approach

Let us start by reconsidering the simple network from Figure 6.1. There always exists the trivial flow f that assigns 0 to every edge. Starting from this trivial flow, we try to improve it. One way for doing so is to find a path from s to t along which every edge has some strictly positive capacity. For the example network from Figure 6.1, one such path is $\pi = s \to b \to e \to d \to t$. Every of the edges along this path has a capacity of 15 (except for one of the edges, which has a higher capacity), so we can update the empty-so-far flow f to the one depicted in Figure 6.6 by increasing the flow along the edges of π by 15.

The new flow f' has a value of 15. We can now try to improve f' by finding another path through the graph. This time, however, the path is only allowed to use *residual capacity* of the graph, i.e., it must not use edges whose capacity is already completely used by f'. In Figure 6.6, we can see that along edges (s,b), (b,e), and (d,f), there is no more free capacity. A close look at the graph reveals that this means that there there is no path from s to t along which only edges with some residual capacity are used. So a simple greedy approach to compute a maximum flow that repeatedly checks for paths that utilize only residual capacity and updates the flow until no new paths can be found would stop at this point. However, we know from Example 21 that there exists a flow with value 25 for the same graph, while flow f' has only a value of 15. This shows that this simple greedy approach does not solve the maximum-flow problem.

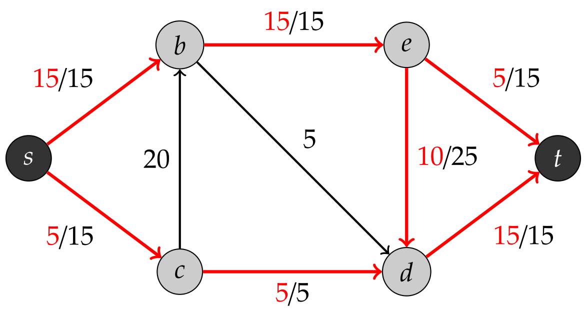


Figure 6.7: An improved flow

6.3.2 Refining the greedy approach

One of the core ideas of the Ford-Fulkerson algorithm is that when updating a flow of a network to increase its value by considering an additional path (as we did above), we can actually *push flow backwards*. This means that when searching for a path from s to t that represents a possibility to increase the current flow, we can take an edge that is already in use in the flow *backwards*.

As this is counter-intuitive at first, let us have a look at an example. We have already seen that the flow in Figure 6.6 is not maximal. If we are allowed to take an edge backwards on which the current flow is > 0, then the path $\pi = s \to c \to d \to e \to t$ is a possibility to direct flow from s to t without exceeding any of the edge capacities. While updating the flow, whenever an edge is taken in the reverse direction, the flow of the edge is updated by *subtracting* from it. There are 15 units of flow left on edge (s, c), 5 units of flow left on the edge (c, d), and 10 units of flow left along the edge (e, t). Along the edge (e, d), there is a flow of 10, so when pushing flow backwards, we can push back at most 10 units of flow. All in all, when updating the flow from Figure 6.6 by the path $s \to c \to d \to e \to t$, we can increase its value by 5 units, as this is minimum of the capacities left along the path. Figure 6.7 shows the resulting flow.

We can see that the flow is valid: for every inner node, the sum of the incoming and outgoing flows is the same and no edge capacity is exceeded. When having a closer look at what happens when we update a flow by some path, it becomes apparent why this is the case: at every inner node, we add the same amount of incoming flow and outgoing flow. Also, no edge capacity is ever exceeded. Both of these properties still hold when we allow some flow to be pushed backwards: added incoming and outgoing flows are the same for all inner nodes, and when pushing back flow, we never push back more flow than available.

Since we now know that when searching for a new path from s to t along which we can increase the value of a flow in a network, we can push back flow, the question arises how we can find paths from s to t that only use the remaining capacity of edges in a graph. In the literature, such a paths is also called *augmenting paths*. To find them, we build the *residual graph* for some flow and some graph.

Definition 13. *Given a graph $G = (V, E)$ with capacities c and a flow f, we define the* residual graph $G' = (V, E')$ *with capacities c' as:*

- $E' = E \cup \{(b, a) \mid (a, b) \in E\}$
- *For all $(a, b) \in E$, we have $c'((a, b)) = c((a, b)) - f((a, b))$*
- *For all $(a, b) \in E$, we have $c'((b, a)) = f((a, b))$*

In the preceding definition, we assumed that for the network $G = (V, E)$, there is no edge $(s, t) \in E$ such that (t, s) is also in E. The assumption has been made for notational convenience only – if we modify the shape of E to allow multiple edges with the same source and target, the assumption that no edge appears together with its reversed edge is not needed.

Figure 6.8 shows the residual graph of the flow network specified in Figure 6.6. It is not difficult to see that the path π from above exists in the residual graph as well, and we do not need to consider pushing back flow as a special case to find it. This fact follows directly from the definition of the residual

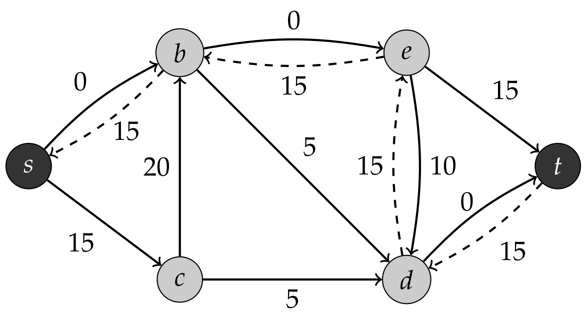


Figure 6.8: A residual graph. Edges that are not present in the graph from which the residual graph was built are dashed. Added edges with 0 capacity are not shown.

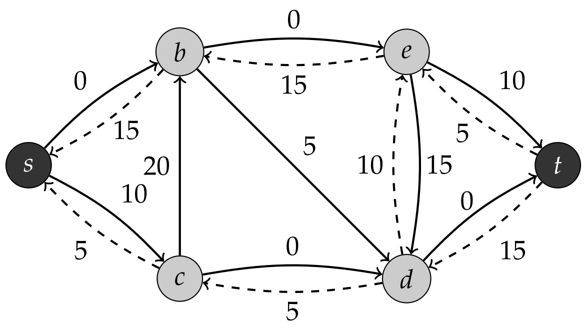


Figure 6.9: Another residual graph.

graph: whenever there is some flow that can be pushed back, a non-0-capacity reverse edge is added. At the same time, all regular edges that have capacity left appear in the residual graph with a positive capacity.

Thus, in order to find a new path that shows how we can increase the value of a flow, we only need to search for a path from s to t in a residual graph with a non-0 capacity along all edges of the path.

Using the concept of a residual graph, we can now easily check if we find a way to improve the flow from Figure 6.7. The residual graph built from it is shown in Figure 6.8. We can see that the graph has a path from s to t, namely the path $s \to c \to b \to d \to e \to t$. The capacity left on this path is 5. If we update the flow from Figure 6.7 by this augmenting path, we obtain the flow in Figure 6.3, which is the best flow that we know so far. But let us try to go one step further by checking if there is still an augmenting path left for this flow. The residual graph for the flow is shown in Figure 6.10. This time, there is no more path left from s to t with a capacity > 0. So our refined greedy approach to find a maximal flow stops here. What is left open at this point is the question whether the flow is actually maximal.

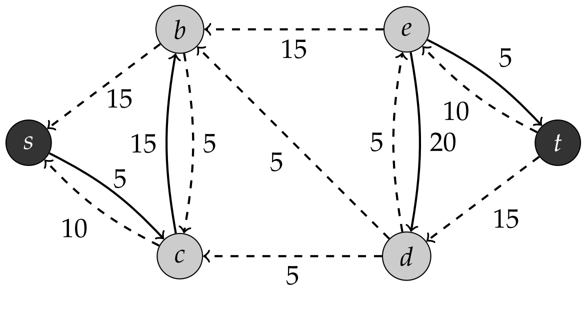


Figure 6.10: Yet another residual graph.

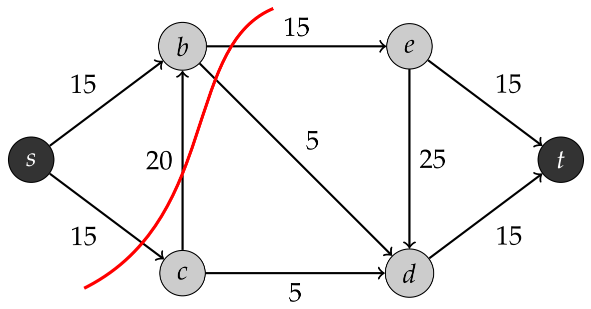


Figure 6.11: An flow network with a cut.

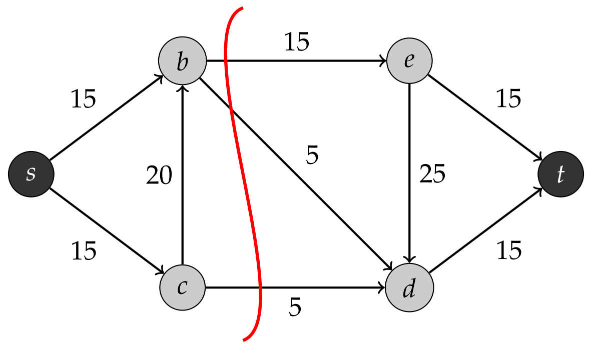


Figure 6.12: An flow network with another cut.

6.3.3 Cuts in graphs and maximal flows

Let us now have a look at the question whether the flow from Figure 6.3 is optimal. The flow has a value of 25, and 25 is less than the sum of the capacities of the edges from s and less than the sum of the capacities of the edges leading to t. So comparing the value of the flow with its trivial upper bound does not suffice to determine the maximality of the flow from Figure 6.3.

However, there is a more general version of this argument that can be used to show the optimality of flows in more general cases. Consider the *cut* of the network from Figure 6.1, which we depict in Figure 6.11. The cut partitions the network into two parts: the first part consists of the notes s and b, while the other part contains all other nodes.

Without loss of generality, we only consider cuts that separate s from t: this means that all paths from s to t contain at least one edge of the cut. The cut from Figure 6.11 consists of the edges (s, c), (c, d), (b, d), and (b, e). Let the part of the graph left of the cut (consisting of nodes s and b be called the A part, while the other nodes constitute the B part. All flow from s to t needs to flow through the edges of this cut. Even more, the edge from c to b can only carry flow from the B part to the A part of the graph, while s is in the A part and t is in the B part. So there are only three edges from the A part to the B part of the graph, and we know that all flow from s to t has to go through these edges at least once, and some flow needs to pass these three edge more than once if the edge from c to b is actually used.

As a consequence, we know that the sum of the capacities of the edges from A to B along the cut is an upper bound on the maximum flow. This sum of capacities is also called the *capacity of a cut* or the *value of a cut*. Here, the cut capacity is 35, which does not help to show that the flow in Figure 6.3 is already maximal. But perhaps there is another cut which allows us to establish this?

The cut in Figure 6.12 does this. The capacity of the cut is only 25, which matches the value of the flow in Figure 6.3. So we know at this point that the flow is optimal and cannot be improved and that the greedy approach to finding a maximum flow outlined in Section 6.3.2 was able to find a maximal flow for *this* network.

Algorithm 6 The Ford-Fulkerson algorithm

```
 1: function Ford-Fulkerson($(V, E), c$)
 2:     $f \leftarrow \{e \mapsto 0 \mid e \in E\}$
 3:     while true do
 4:         $(G', c') \leftarrow$ Build-residual-graph($(V, E), c, f$)
 5:         $\pi \leftarrow$ Find-augmenting-path($G', c'$)
 6:         if $\pi$ is empty then return $f$
 7:         $w \leftarrow \min\{c'(e) \mid e \in \pi\}$
 8:         for $e \in \pi$ do
 9:             $f(e) \leftarrow f(e) + w$
10:         end for
11:     end while
12: end function
```

In fact, reasoning over cuts allows us to show that the refined greedy approach from Section 6.3.2 *always* finds a maximal flow. For this, we look at the residual graphs of the flows that lead to the termination of the greedy approach with pushing flow backwards. For our running example, the residual graph is given in Figure 6.10. Recall that the approach terminates if no more paths from s to t with capacity > 0 can be found in the residual graph. We can see that along the cut from Figure 6.12, the residual graph has only backwards edges. This is a sufficient condition for the graph to have no more path from s to t (with strictly positive capacity).

Cuts in flow networks and residual graphs with no more path between s and t have a close connection: such a residual graph *induces* a cut in a network. To see this, assume that A is the set of nodes in the residual graph that are reachable from s along some non-0-capacity path. We already know that t is not reachable by the assumption that there is no path to t left, so we have $t \notin A$. Let B be the nodes of the graph that are not in A. We have that (A, B) is a valid cut, as s is in A, and t is in B. Furthermore, we know that in the residual graph, there is no edge from A to B with a capacity > 1. This is simply because if we have an edge (a, b) where $a \in A$, $b \in B$, and $c((a, b)) > 0$, then b would be included in A as well by the definition of A, which is a contradiction. By the definition of the residual graph, this means that the flow from which we built the graph already uses all edges from A to B completely. As (A, B) is a cut of the graph, this means that the flow is maximal.

So we know that our refined greedy approach to compute maximal flows is actually correct: whenever there is no more path from the source to the sink node left in the residual graph, then by partitioning the nodes into those that are still reachable from s and those that are not gives us a cut that shows that the computed flow is maximal. As a corollary, the approach also computes a *minimal cut*, i.e., a cut with minimal capacity. We will see in Section 6.5.2 that computing minimal cuts is of interest on its own.

6.3.4 Assembling the concepts to a proper algorithm

We have seen the main ingredients of the Ford-Fulkerson algorithm to solve the maximum flow problem in the preceding subsections, namely residual graphs and updates of the current flow based on augmenting paths. All of these ideas are assembled in Algorithm 6. In the algorithm, a *candidate flow* is gradually updated until no more improvement is found. In every step of the algorithm's execution, the residual graph is built. We discussed the construction of this graph in Section 6.3.2. Then, the algorithm tries to find an augmenting path in the residual graph. If one is found, the flow is updated according to the path, and the process is repeated. If no such path is found, the algorithm terminates. In case a path is found, the algorithm computes how much flow can be pushed along the path (line 7).

The only part that we did not discuss yet is how to actually find an augmenting path. Using Dijkstra's algorithm, we could find the augmenting path with the highest residual capacity. Alternatively, using depth-first-search, we could find some arbitrary path in time $O(|E|)$. Using the latter approach, every iteration of the algorithm's main loop would be executed in time $O(|E|)$.

In order to determine the overall computation time of Algorithm 6, we need to know how many iterations of the the main loop are performed. Let C be the trivial upper bound on the flow of the network. We know that that in every iteration of the loop, the value of the flow f increases by at least 1. This is because if all elements of f are integer (which they are at the beginning) and all capacities in the graph are integer, then so will be w. By induction, it trivially follows that all values of f are always integer, and so if a path is found in line 5, then it has an integer value of at least 1.

So we know at this point that the computation time of the Ford-Fulkerson algorithm is $O(C \cdot |E|)$. If we have large values of C, then this can be a lot! So the question is natural whether this analysis is actually tight. Unfortunately, it can be shown that in it current shape, the Ford-Fulkerson algorithm may need C many iterations: Kleinberg and Tardos (2006, Section 7.3) give an example flow network with $C = 200$ and a sequence of C many augmenting paths that each increase the flow by 1, while 1 is always the remaining capacity of the augmenting path. So if the calls to function Find-augmenting-path in line 5 of Algorithm 6 return the paths from this sequence, the outer loop indeed needs to be executed C many times.

For some problems with small values of C, this computation time may be acceptable. but for networks with large capacities, the computation time bound of $O(C \cdot |E|)$ is unsatisfactory. However, Edmonds and Karp (1972) and independently, Dinic (1970) showed that always picking the *shortest path* in the Find-augmenting-path function leads to shorter computation times. In particular, $O(|E| \cdot |V|)$ many iterations of the algorithm's main loop then suffice, leading to a computation time of $O(|E|^2 \cdot V)$. We will not prove this result here, but rather discuss a slightly more sophisticated algorithm to solve the maximum flow problem with high edge capacities in the next section.

Let us now quickly have a look at the bipartite matching problem from Section 6.2 again. First of all note that the value C in the networks built from problem instances is quite small: in fact, it is the same as the minimum of the sizes of the sets A and B. So regardless of how augmenting paths are chosen, the algorithm terminates quite quickly. Furthermore, we already discussed that the algorithm computes flows in which edges are assigned integer flow values, provided that the graph that we start with only has edges with integer capacities. In the reduction from bipartite matching to maximum flow, we had to assume that the solution to the network flow problem is integer-valued for the reduction to make sense.

For flow networks with only integer-valued capacities, the Ford-Fulkerson algorithm not only has the property that it finds integer-valued maximal flows whenever they exist, but its correctness actually shows that among the maximal flows, there is always one in which the flows along the edges are all integer-valued. This is an important corollary of the algorithm, as it tells us something about the maximum flow problem that we can exploit when reducing other practical problems to a maximum flow problem instance. Bipatite matching is one such problem, and our analysis shows that the assumption in the reduction that the computed flows are integer-valued is indeed justified.

Readers who are interested in developing the results presented in this section in a more thorough way may want to read Section 7.1-7.2 of Kleinberg and Tardos (2006).

6.4 A faster maximum flow algorithm for graphs with large capacities

Literature: *Kleinberg and Tardos, 2006, Section 7.3*

Algorithm 7 is a refinement of the Ford-Fulkerson algorithm that we discussed in the previous section. A couple of steps have been added to the algorithm, starting in line 3, where an initial *scaling parameter u* is computed. We initialize the scaling parameter by an upper bound on the flow along any augmenting path, namely the maximum capacity of any edge along the outgoing edges of s.

The algorithm has two nested main loops. The outer loop is executed as long as the scaling parameter is at least 1. At the end of every iteration of the outer loop, the scaling parameter is divided by 2. The outer loop terminates once the division operation in line 19 is executed when $u = 1$. The inner nested main loop is the same as for the classical Ford-Fulkerson algorithm, except that after building the residual graph, all edges with a capacity smaller than u are removed.

Algorithm 7 A scaled version of the Ford-Fulkerson algorithm

```
1: function Scaled-Ford-Fulkerson($(V, E)$, $c$)
2:     $f \leftarrow \{e \mapsto 0 \mid e \in E\}$
3:     $u \leftarrow \max\{c((s,a)) \mid a \in V, (s,a) \in E\}$
4:     while $s \geqslant 1$ do
5:         while true do
6:             $(G', c') \leftarrow$ Build-residual-graph($(V, E), c, f$)
7:             for $e \in E$ do
8:                 if $|c'(e)| < u$ then $c'(e) \leftarrow 0$
9:             end for
10:            $\pi \leftarrow$ Find-augmenting-path($G', c'$)
11:            if $\pi$ is empty then
12:                break
13:            end if
14:            $w \leftarrow \min\{c'(e) \mid e \in \pi\}$
15:            for $e \in \pi$ do
16:                $f(e) \leftarrow f(e) + w$
17:            end for
18:        end while
19:        $u \leftarrow \lfloor \frac{u}{2} \rfloor$
20:    end while
21:    return $f$
22: end function
```

The algorithm always maintains a valid flow in f: all augmenting paths found in residual graphs with some edges removed are also augmenting paths in the residual graph in which no edges capacities are zeroed out. Furthermore, in the last iteration of the outermost loop, we have $u = 1$. In this case, no capacities are altered in line 8, and when no more augmenting path are found afterwards, we know that the flow is maximal. So the algorithm computes the maximum flow, just like the classical Ford-Fulkerson algorithm.

Let us now analyze whether the scaled version of the algorithm is faster. The key to answering this question are the properties of the residual graph G' built by the algorithm when the inner loop terminates in line 12. At that point, we know that there is no path from s to t in G', as the statement in line 12 is only executed if no such path is found. Thus, there is a cut between the the vertices reachable from s in G' and those that are not. Because some capacities in c' have been zeroed out, the cut does not show that the flow in f is already maximal. However, a cut is a cut, and it can be used to derive upper bounds on the flow. Along the cut, we have zeroed out at most $|E|$ edges, and each of them had a residual capacity of less than u. So if we examine the cut, the residual capacity left along the cut in the unmodified residual graph is less than $u \cdot |E|$. Thus, the value of f is not more than $u \cdot |E|$ units of flow away from the maximum.

By induction, we can show that in line 6, we always have that the value of f is at most $(2 \cdot u + 1) \cdot |E|$ units of flow away from a maximal flow. Initially, this property holds by our choice of u: as all flow has to start with an edge that leaves s, there are at most u units of flow possible along such an edge, and there are $|E|$ edges, $2 \cdot u \cdot |E|$ is an upper bound on the value of a maximum flow. Since we know that when line 12 is executed, there is only at most $u \cdot |E|$ units of residual flow left along the cut mentioned above, and u is divided by 2 (and rounded down) in line 19, we know that after that line is executed, there are still at most $(2 \cdot u + 1) \cdot |E|$ units of residual flow left.

The inductive argument allows us to deduce that the inner loop of the algorithm is executed at most $2 \cdot |E| + 1$ times in every iteration of the outer loop: all augmenting paths found have a residual capacity of at least u, as edges with a lower capacity are removed in line 8. Thus, if at the start of the inner loop, there are at most $(2 \cdot |E| + 1) \cdot u$ units of capacity left, there are at most $2 \cdot |E| + 1$ many iterations.

For the outer loop, we can see that it is executed at most $\lceil \log_2(C') \rceil + 1$ times for $C' = \max\{c((s,a)) \mid a \in V, (s,a) \in E\}$, as u is initialized by C' and divided by 2 in every step.

	A	B	C	D	E
B	2	-			
C	5	4	-		
D	5	4	1	-	
E	3	5	2	3	-
F	5	5	5	5	5

Table 6.1: Numbers of games played between 6 teams played so far for the application from Section6.5.1.

Since the computation time of an iteration of the innermost loop is $O(|E|)$, we can now sum up the overall computation time to $O(\log_2(C') \cdot |E|^2)$. So while the classical Ford-Fulkerson algorithm has a computation time that is linear in $|E|$ and C', the scaling variant is quadratic in $|E|$, but logarithmic in C' (which we know to be $\leqslant C$). For large numbers in the capacities of the edges, it is thus a better choice than the classical Ford-Fulkerson algorithm (without any specific choice on the augmenting paths).

6.5 Example applications of the maximum flow problem

6.5.1 League championship

Literature: *Kleinberg and Tardos, 2006, Section 7.12*

As an example for a flow problem in which edge capacities higher than 1 occur naturally, we consider the *league championship* problem.

Assume that we have a set of teams T that compete in a tournament. Every team plays n matches against each other team and gets a point for every match that it wins. There are a no draws possible. Some matches have already been played, and others have not. We want to find out if is still possible for some team $t \in T$ to win the tournament, i.e., to achieve the highest sum of points from all matches among all teams.

To examplify the league championship problem, we consider the set of teams $T = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$ and ask the question whether team $t = \mathbf{E}$ can still win for $n = 5$. Table 6.1 shows the numbers of matches already played between all pairs of teams. After these matches have been played, we have the following distribution of points:

- Team A: 12 points
- Team B: 12 points
- Team C: 13 points
- Team D: 10 points
- Team E: 8 points
- Team F: 4 points

We can see that team F cannot win the tournament any more, as they have already played all of their matches (25 in total), but only won 4 of them. Team E still has 7 games to play, so the highest number of points that they can accrue overall is 15, assuming they win all of the remaining matches. So it looks like that they can still win against the teams **A**, **B**, **C**, and **D**, as no team has more than 13 points at the moment. However, there are also matches to be played in which team **E** does not take part. From Table 6.1, we can see that there are 9 of these matches. Team **E** can only still win if there is some way to allocate these 9 points to the teams such that the total number of points of every team sums up to $\leqslant 14$. Otherwise, after the games between teams **A**, **B**, **C**, and **D** have been played, one of teams would *have* to have more than 14 points, so that team **E** cannot win anymore.

It is not obvious if the teams **A**, **B**, **C**, and **D** can play their matches in a way such that no team has more than 14 points at the end of the tournament. We can however reduce this problem to an instance of the maximum flow problem to solve it. Figure 6.13 shows the resulting flow network. It has been built

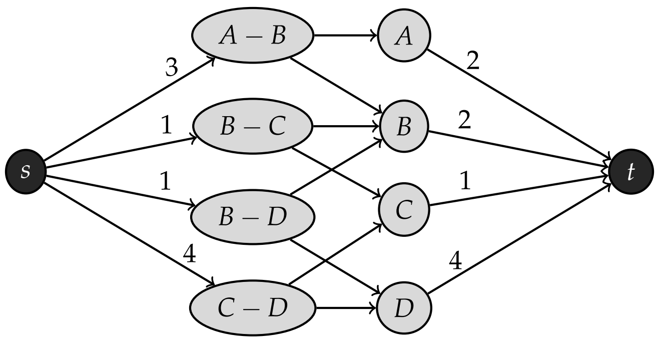


Figure 6.13: The flow network for the league championship problem from Section6.5.1. Unlabeled edges do have limits of their capacities.

using the following process:

- For every combination of teams that still has some matches to play, we added an inner node, and connected it to s by an edge whose capacity equals the number of matches that are still to be played.

- For every team in the tournament that still has some matches to play, we added an inner node and connected it to t by an edge whose capacity equals the maximum number of points that the team may obtain without overtaking team **E** in the tournament ranking.

- For inner nodes that represent matches, we add edges to team nodes for the teams that participate in the match. The edges have an unbounded capacity.

Note that strictly speaking, our definition of the maximum flow problem does not allow edges with an unbounded capacity. However, by replacing all unbounded capacities by some numbers that are higher than the sum of all bounded edge capacities, we can ensure that these edges never become a bottleneck, which makes the capacities unbounded in practice.[1]

Let b be the number of points to be distributed between the teams. The flow network built with these rules has the property that the value of the flow is bounded by the number of points that are still to be distributed between the teams that we want to lose the tournament. Furthermore, no team can accrue more points that they are allowed to without team $t =$ **E** losing the tournament, and a team can only accrue points from matches in which it participates. At the same time, every maximum flow with a value b shows how to distribute the points such that no team gets more points that it should. Likewise, if we have a distribution of points that lets team t win, then we can translate this distribution to a flow of value b. Thus, if and only if we find a flow of value b, team **E** can still win the tournament. Executing the Ford-Fulkerson algorithm thus allows us to find out if team **E** can still win.

Figure 6.14 shows a maximum flow. It has a value of $b = 9$, which proves that team **E** can still win the tournament. From the flow, we can see that this outcome is achievable if:

- in the matches between teams **A** and **B**, team **A** wins 2 matches, and team **B** wins one match,

- the one match between team **B** and team **C** is won by team **B**,

- the one match between team **B** and team **D** is won by team **D**, and

- in the matches between teams **C** and **D**, team **C** wins 1 match, and team **D** wins 3 matches.

[1]Note that this line of reasoning requires that there is no path from s to t in a flow network that completely consists of unbounded-capacity edges. None of the maximum-flow problems in this chapter have such paths, however, and as the value of the optimal flow would be unbounded in such a case, we could just augment the Ford-Fulkerson algorithm by a check for such a path and let it return ∞ as the value of the flow in such a case.

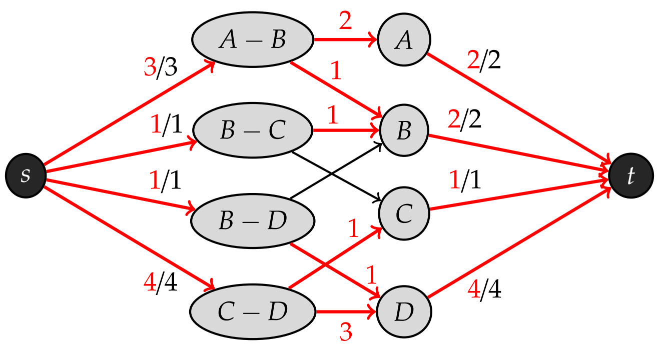


Figure 6.14: A maximum flow for the flow network for the championship problem from Section6.5.1. For every edge without a capacity bound, only the flow along the edge is shown.

6.5.2 Project selection

Literature: *Kleinberg and Tardos, 2006, Section 7.11*

Assume that a company has n projects that it can implement. Some projects earn the company money, whereas others cost money. Also, some projects are requirements for other projects. If project number i is a requirement for project j, then project number j cannot be implemented if project i is not also implemented. *The company wants to select a subset $U \subseteq \{0, \ldots, n-1\}$ of the projects to implement that maximizes the reward.* We will represent the monetary benefits of implementing a project i by a variable p_i in the following. Thus, the optimization criterion boils down to selecting some U that maximizes $\sum_{i \in U} p_i$.

Project selection can be a difficult problem. Take for example the set of projects $\{0, \ldots, 5\}$ with the rewards $p_0 = 3, p_1 = 2, p_2 = 3, p_3 = -2, p_4 = -2$, and $p_5 = -4$. If we had no dependencies between the projects, finding an optimal set U would be easy: we would just set $U = \{i \in \{0, \ldots, n\} \mid p_i > 0\}$. However, for this example we will have the dependency relation $R = \{(0,3), (0,4), (1,3), (1,4), (2,1), (2,5)\}$, where each element $(i, j) \in R$ represents that project i can only be implemented if project j is implemented as well.

We can reduce the project selection problem to finding a minimal cut as follows:

- We introduce one vertex for every project and add all elements of R as edges. The edges have unbounded capcities.
- We introduce a source vertex and for every project i with $p_i > 0$, we add an edge from the source vertex to the node for project i with capacity p_i.
- We introduce a sink vertex and for every project i with $p_i < 0$, we add an edge from the node for project i to the sink vertex with capacity $-p_i$.

Figure 6.15 shows the flow network for the running example. We will show that *minimal cuts* represent optimal sets of projects, so that we can use the Ford-Fulkerson algorithm (which computes minimum cuts as as a side-product) to obtain them.

To show this, we need two lemmas.

Lemma 3. *Let (A, B) be a minimal cut of a network built according to the construction from above. For no edge (a, b) in the graph where $a \neq s$ and $b \neq t$, we have that $a \in A$ and $b \in B$.*

Proof. All edges of this form have an unbounded capacity (i.e., one that is larger than the trivial upper bound on the flow of the network). This means that along the minimal cut, such edges always have some residual capacity. Since along a minimal cut, no edges from A and B have some residual capacity, this means that the cut cannot span edges (a, b) for which $a \neq s$ and $b \neq t$. □

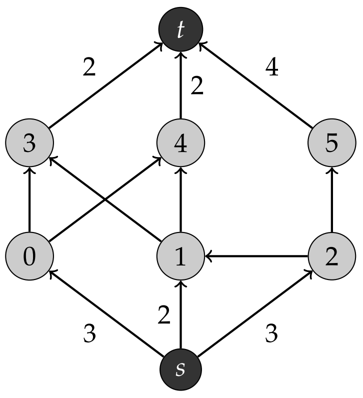


Figure 6.15: A flow network for project selection. Edges without labels have an unbounded capacity.

The lemma shows that if we have a minimal cut (A, B) and A contains some project i, then it also contains all projects on which project number i depends.

Let us now connect the value of a cut (A, B) to the profit that results from selecting the projects in A as the ones to be implemented.

Lemma 4. *Let (A, B) be a cut in a flow network built by the construction from above that does not separate internal nodes, and C be the sum of the positive payoffs. We have that:*

$$value(A, B) = C - \sum_{i \in A} p_i$$

Proof. The assumption in the statement already excludes some edges, and all edges with $a \in B$ and $b \in A$ do not contribute to the value of the flow. So we know that the only edges that can contribute to $value(A, B)$ are those that start in s or lead to t. These edges have capacities that represent the profits of the projects. Thus, we can deduce:

$$\begin{aligned}
value(A, B) &= \left(\sum_{i \in A, p_i < 0} -p_i \right) + \left(\sum_{i \in B, p_i > 0} p_i \right) \\
&= \left(\sum_{i \in A, p_i < 0} -p_i \right) + \left(\sum_{p_i > 0} p_i \right) - \left(\sum_{i \notin B, p_i > 0} p_i \right) \\
&= \left(\sum_{i \in A, p_i < 0} -p_i \right) + \left(\sum_{p_i > 0} p_i \right) - \left(\sum_{i \in A, p_i > 0} p_i \right) \\
&= \left(\sum_{p_i > 0} p_i \right) - \left(\sum_{i \in A} p_i \right) \\
&= C - \left(\sum_{i \in A} p_i \right)
\end{aligned}$$

□

So we know that by minimizing the value of a cut among those that do not separate internal edges, we can maximize the overall profit. By the preceding lemma, we know that minimal cuts never have this property. So it suffices to apply the Ford-Fulkerson algorithm to find a minimal cut that represents an optimal project selection.

If we apply the algorithm to the network from Figure 6.15, we obtain the flow depicted in Figure 6.16. We can see that projects 0, 1, 3, and 4 have been selected to be implemented, while projects 2 and 5 are left out. The overall profit is 1.

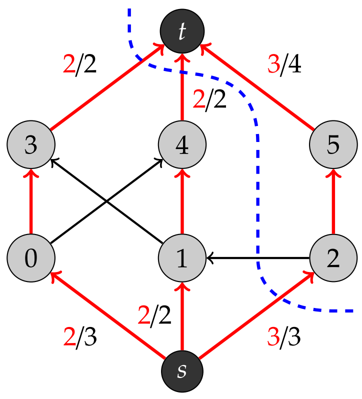


Figure 6.16: A flow network and a flow for project selection.

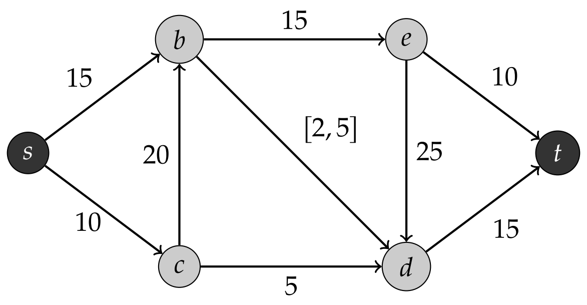


Figure 6.17: An example flow network with a lower bound on one edge. We label edges with a lower bound of a on the flow and a capacity of b as $[a, b]$.

Note that in Figure 6.16, there is also flow along paths that connects projects that have not been selected for implementation. In a sense, projects 5 and 2 are implemented *partially* in this flow. Since the cost of project 5 is however more than the profit of project 2 (which depends on 5), the edge from s to 2 saturates and the cut separates vertex s from vertex 2. The cut (A, B) also includes the edge from 2 to 1 - but because this edge leads from B to A, and not from A to B, this does not contradict Lemma 3.

6.6 Lower bounds on flows

There are some practical problems whose simple encoding into maximum-flow problem instances requires that we can impose *lower bounds* on flows along some edges of a network. Since the maximum-flow problem's definition does not include lower bounds, we need to either extend the problem and the algorithm to support lower bounds, or to devise a procedure to modify networks with lower bounds on edge flows to networks without lower bounds, such that algorithms for the standard maximum-flow problem can be used.

We follow the latter approach here, and restrict out attention to the case that we are only interested in maximal flows where the value of the flow is $\sum_{e \in (\{s\} \times V) \cap E} c(e)$, i.e., where the value of the flow is the same as the trivial upper bound on the flow.

6.6.1 Modifying networks with lower bounds

Literature: *Kleinberg and Tardos, 2006, Section 7.7*

Consider the flow network depicted in Figure 6.17. It has a lower bound of 2 on one edge. Recall that we are are only interested in flows whose values are trivially maximal, i.e., is equal to value $\sum_{e \in (\{s\} \times V) \cap E} c(e)$. In the network from Figure 6.17, this means that we are only interested in flows with a value of 25.

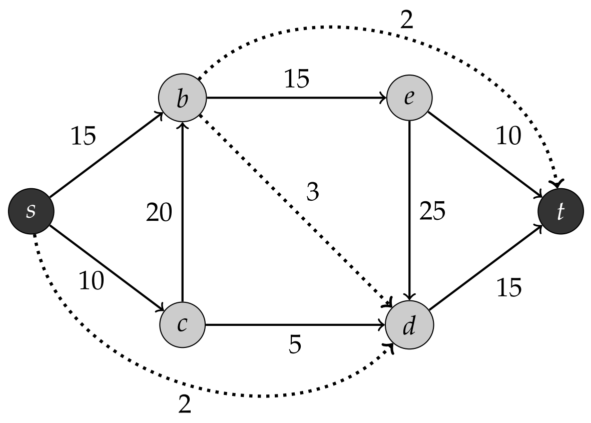


Figure 6.18: An modified flow network without lower bounds. The modified lines are dotted.

Let us assume that f is a maximum flow with this value that satisfies the requirement that the flow along the (b, d) edge is at least 2. There are then at least two units of flow into b that are used along this edge, and there are at least two units of flow out of vertex d that come from this edge. This gives rise to the following idea: we can easily reroute this flow such that it does not take the (b, d) edge any more. To accomplish this, we *collect* this flow from b, and *inject* it back into d. Then, we can reduce the flow along the (b, d) edge by 2, set its capacity to 3, and set its lower bound to 0. This leads to a network without lower bounds. In order to inject 2 units of flow into d, we just add an edge from s to d with a capacity of 2. In order to collect 2 units of flow from b, we add an edge from b to t with two units of flow. The resulting network is shown in Figure 6.18.

This network has the following important property: if we have a trivially maximal flow in the new network (i.e., one with a value that is equal to the trivial upper bound on a flow in the network), then we can easily get a maximal flow for the old network: we just remove all flow from the added edges, and assign them to the edge that had a lower bound in order to ensure that the lower bound is met. Likewise, any trivially maximal flow for the old network can be translated to a trivially maximal flow for the new network by reducing the flow along all edges by their respective minimum flow, and use the now free units of flow to maximally utilize the newly added edges.

As a result, we now know how to deal with flow networks with lower bounds on the flows of some edges. Note however that our assumption that we are only interested in trivially maximal flows is crucial for the reduction to work: if a modified network does not admit a trivially maximal flow, then the flow that we compute with the Ford-Fulkerson algorithm may not utilize the added edges to the maximum, and hence when translating the flow to the old graph, it may not satisfy the lower bounds.

6.6.2 Example application: airline scheduling

Literature: *Kleinberg and Tardos, 2006, Section 7.9*

As example application for maximum-flow with lower bounds, we consider the *airline scheduling problem*. We explain the problem and its encoding to maximum-flow by means of an example.

Assume that an airline, let it be called *K1 airlines*, wants to serve a given set of flights every day. Table 6.2 contains a list of flights for the airline, with departure and arrival times for every connection. Furthermore, we assume that K1 airlines has three planes, which are stationed in Bremen, Copenhagen, and Hamburg. Given a list of turn-around times for each airport, we now ask the question *if there is a feasible schedule for the planes such that all flights are served if the planes start their days at their home bases, and at the end of the day, the planes have to be back at their home bases.*

To encode the problem, we first of all model all flights to be served on a day as vertex tuples in a flow network, where each tuple is connected by an internal edge. Figure 6.19 shows the network for the flight schedule from Table 6.2. To simplify the graphical representation of the network, we arranged the

Flight #	From - To	Times
K1 152	Bremen - Munich	1400 - 1530
K1 171	Munich - Madrid	1700 - 2100
K1 181	Hamburg - Bremen	0900 - 1015
K1 182	Munich - Zurich	1900 - 2005
...	...	...

Table 6.2: Flights served by *K1 Airlines*.

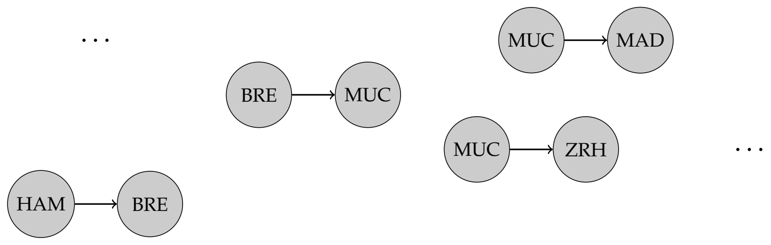


Figure 6.19: Building a flow network for the the airline scheduling problem, part 1

fights roughly by the times of departure of the arrival along the x axis. Since Table 6.2 does not show all of the flights, we have two "..." placeholders in Figure 6.19 for the flights not shown.

In the next step, we connect the arrival nodes of flights to departure nodes of flights that can be served by the same plane. For example, between the flights from Hamburg to Bremen and the one from Bremen to Munich, there is 3h45m of buffer time, which should be enough to complete ground handling of the plane. To check which connecting edges should be present, we need to differences between the arrival and departure times of the individual flights with the minimum ground handling times of the respective airport. Furthermore, we add lower and upper bounds of 1 on the edges of the network that represent serving a flight. This should intuitively enforce that in our solution to the flow problem, every flight is served exactly once. The resulting network is shown in Figure 6.20.

Next, we add the information where the planes are stationed, which flights can be the first one of a plane on a day, and which ones can be the last ones of a plane on a day. We add edges to represent the latter types of information, and introduce source and sink nodes for the stationing of the planes. The resulting network is shown in Figure 6.21. For the dotted lines, the capacities do not matter, as long as they are at least 1.

Finally, because the network flow problem does not support multiple source and sink nodes, we add

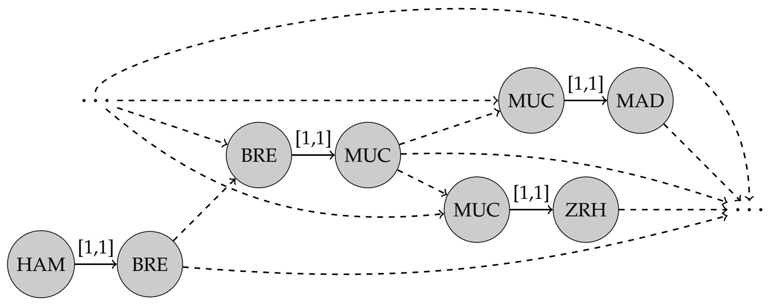


Figure 6.20: Building a flow network for the the airline scheduling problem, part 2

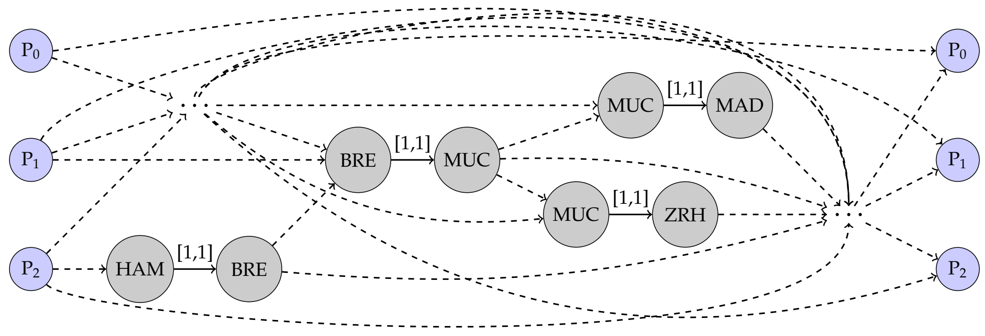


Figure 6.21: Building a flow network for the the airline scheduling problem, part 3

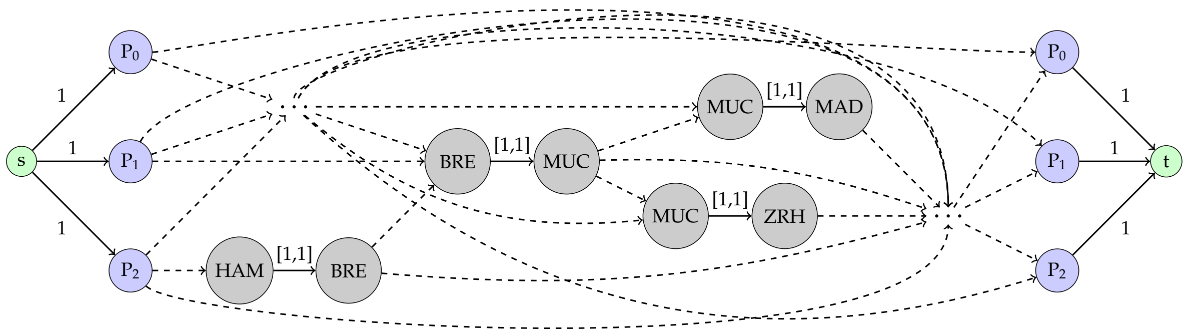


Figure 6.22: Building a flow network for the the airline scheduling problem, part 4

global source and sink nodes and connect these to the plane source and sink nodes to make them regular inner nodes. The final network is shown in Figure 6.22.

Assume now that we have *any* valid flow for this network. We know that the flow represents a correct flight schedule for the planes: because all lower bounds are satisfied, all flights are served, and by the construction of the graph, flow can only jump from one flight to the next one if the two flights can be served by the same plane. Likewise, any correct flight schedule for the planes induces a trivially maximal flow in the network by choosing the flight lists of the planes as augmenting paths.

Thus, the construction outlined in this section is suitable to reduce the airline scheduling problem to the maximum-flow problem with lower bounds.

6.7 Circulation problems

Literature: *Kleinberg and Tardos, 2006, Section 7.7*

In the maximum-flow problem, we have designated source and sink notes. There are some practical problems that do not have this property, such as the *circulation problem*. In this section, we want to

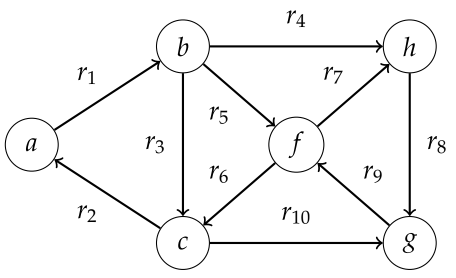


Figure 6.23: A network for the circulation problem.

discuss how despite the absence of designated source and target nodes, circulation problems can still be solved with algorithms for the maximum-flow problem.

In the circulation problem, we are given a set of *depots* D, a set of *routes* $E \subseteq D \times D$, a family of real-valued *production rates* $\{p_d\}_{d \in D}$ with $\sum_{d \in D} p_d = 0$, and a family of positive real-valued route capacities $\{c_e\}_{e \in E}$. We want to find a *utilization* of the routes $u : e \to \mathbb{R}$ such that for all $e \in E$, $0 \leqslant u(e) \leqslant c_e$ and for all depots $d \in D$, we have:

$$\left(\sum_{(d',d) \in E} u((d',d)) \right) - \left(\sum_{(d,d') \in E} u((d,d')) \right) + p_d = 0. \tag{6.4}$$

To concretize the problem a bit, let us consider a slightly abstracted example for the circulation problem. Assume that a juice company has a couple of factories and distribution centers. In factories, juice is produced, and juice is distributed to the customers from the distribution centers. Some juice is also delivered to the customers directly from the factories. We can consider both factories and distribution centers as depots, where (most) factories have a positive overall production rate, and the distribution centers have a negative production rate. We also have a couple of transportation routes that are known to be served by transport companies. We assume that for reasons of environmental protection, we want to only use trucks that already have a transport job for the reverse direction of the route, so that we can simply utilize them on their way home. This motivates that the routes are singly-directional and have a limited capacity. Figure 6.23 depicts a circulation problem with six depots/factories and ten routes.

Reducing a circulation problem to the maximum flow problem can be done in a way that is similar to the one that we followed for performing project selection in Section 6.5.2. We introduce a dedicated source node s and a dedicated sink node t. Then, for all nodes d with $p_d > 0$, we add an edge from s with capacity p_d. For all notes d with $p_d < 0$, we add an edge to t with capacity $-p_d$. Unlike in the project selection problem, we are however not interested in a maximum cut here, but rather in an actual maximum flow. In particular, we are interested in trivially maximal flows. There is a bijection between trivially maximal flows in the network and utilization functions by the fact that when adding the nodes from s and to t, the requirement from Equation 6.4 is exactly implemented at the inner nodes of the network – but this property only holds when the flow is trivially maximal.

Note that because we are only interested in trivially maximal flows, we can also use the technique from Section 6.6 to impose lower bounds on flows if this is wanted.

6.8 Conclusion

In this section, we discussed algorithms and applications of the maximum-flow problem. It turned out that this problem is useful for a variety of practical applications, and the Ford-Fulkerson algorithm can solve it in polynomial time. Interestingly, when this algorithm terminates, it also (implicitly) computes a minimum cut of the flow network, which is of interest on its own.

We have discussed the Ford-Fulkerson algorithm in far more detail than the computational engines in the preceding sections. This choice was rooted in the fact that unlike the other computational engines considered in this course, the algorithm is seldomly used as a stand-alone tool. Rather, due to its simplicity, it is often reimplemented from scratch. We discussed the algorithm in enough detail to allow an easy reimplementation.

The maximum flow problem is one of only two problems considered in this course that are known to be solvable in polynomial time. Yet, it is very useful to encode practical problems into it, and the idea of the Ford-Fulkerson algorithm that flow can also be pushed *backwards* is a little bit surprising at first. The network flow problem thus serves as a good *checkpoint* to use this idea to solve practical applications in polynomial time.

Chapter 7

SMT

In the preceding chapters, we have discussed many practical problems that have a rich combinatorial structure but only a boolean domain of the variables, and we have discussed practical problems that have non-boolean variables, but a relatively simple structure of the constraints on the solutions. It is natural to ask the question if complex constraints on solutions and non-boolean variable domains can also be combined without losing the efficiency of the computational engines.

Exactly this combination is the realm of *satisfiability modulo theory* (SMT) solvers. The allow to specify arbitrary boolean combinations of expressions in some *theories*, such as integer arithmetic or the theory of uninterpreted functions (which we will both get to know later). An SMT solver then checks if values for all variables in their respective domains can be found such that the overall *SMT formula* is satisfied.

Example 22. Let us consider a simple example SMT formula that only makes use of one theory, namely the one of linear arithmetic over real values:

$$(x \geqslant 5 \vee y \geqslant 9) \wedge (x - y \leqslant 4 \vee x \geqslant 9) \wedge (y - x \leqslant 4 \vee x \leqslant 3)$$

Due to the use of the disjunction operator, this formula cannot be represented as a set of constraints for the linear programming problem. Nevertheless, it has a solution. For example, the assignment $(x, y) = (5\frac{1}{2}, 5\frac{1}{2})$ satisfies the SMT formula. ★

The SMT problem, i.e., checking if an SMT formula has a solution, is an *extension* of the SAT problem from Chapter 2. SMT solvers are nowadays used in various application domains, such as program verification, operations research, and for solving *planning* problems.

7.1 Problem definition

A *theory* consists of a set of *theory symbols* and axioms on their interpretation. Theory symbols can either be

1. functions that map some value in the domain of the theory to other values in the domain of the theory, or
2. predicate symbols, which map values in the domain(s) of the theory to **true** or **false**.

We treat equality (=) as a special binary predicate and only consider theories that have the equality predicate in the following. The set of function and predicate symbols of a theory is also called its *signature*. If a theory has *constants*, we consider them to be 0-ary functions.

A *term* in some theory T is an expression consisting of function and predicate symbols that is *syntactically valid*, i.e., applies functions only according to their *arity*. For the simplicity of presentation, we assume that all function and predicate symbols in the theory have a fixed arity and consider only syntactically valid terms henceforth.

An *SMT formula* for some theory T is a Boolean formula over *predicate terms*, i.e., syntactically valid terms that have an outermost predicate. We say that an SMT formula is satisfiable if there is some interpretation for the sub-terms that is consistent with the axioms of the theory. We call an interpretation of all terms a *model* for the formula.

Definition 14. *The theory of uninterpreted functions consists of a single binary prediate "=" and functions symbols $\{f, g, \ldots\}$ of some fixed arities. The theory works with* arbitrary *types of values.*

Example 23. The following expression is a well-formed SMT formula under the theory of uninterpreted functions:

$$(f(g(x,y)) = x \vee f(x) = x) \wedge ((f(x) = x) \rightarrow (f(y) = y)) \wedge (f(y) \neq f(x))$$

Here, f is a 1-ary function, g is a 2-ary function, and x and y are 0-ary functions. The terms in this formula are x, y, $g(x,y)$, $f(g(x,y))$, $f(g(x,y)) = x$, $f(x)$, $f(x) = x$, $f(y)$, and $f(y) = y$.

The SMT formula is satisfiable with the model

- $x = \sqrt{-1}$,
- $y = \text{“}\mathit{sometext}\text{''}$,
- $z = \frac{1}{2}$,
- $g(x,y) = \frac{1}{2}$,
- $f(x) = \frac{1}{2}$,
- $f(z) = \frac{1}{2}$,
- $f(y) = \text{“}\mathit{sometext}\text{''}$.

Because the value domain for the theory of uninterpreted functions is not fixed, the values $\sqrt{-1}$, “*sometext*”, and $\frac{1}{2}$ can be replaced by any possible triples of (distinct) values. ★

Later in this chapter, we also consider the combination of several theories in a singe SMT formula. In that cases, we assume that all function and predicate symbols of the theories are disjoint except for the equality predicate. We defer discussing examples for combined theories to later.

7.2 A first example

We consider a simple example for SMT solving, which shows what types of practical problems can be tackled with SMT solvers.

The operations research office of an aluminium factory wants to schedule its production. In every quarter of the year, the factory can produce at most 1,000,000 units of aluminium. We want the factory to produce 1.5 million units of aluminium overall, with an overall budget of 5.736 million currency units. The market demand for aluminium is 1000 units in the first quarter, 1500 units in the second quarter, 2000 units in the third quarter, and 2500 units in the fourth quarter. The other produced aluminium should be in stock at the end of the fourth quarter. Having aluminium in stock is costly: every unit of Aluminium that is in stock at the end of a quarter costs 0.1 currency units (CUs).

We only take the energy and storage costs into account. Energy is needed to produce aluminium from raw materials. The electricity company already announced that one kWh will cost 0.235 CUs in the first quarter, 0.245 CUs in the second quarter, 0.248 CUs in the third quarter, and 0.255 CUs in the fourth quarter. In order to get these (good) prices, the factory has to count as a major consumer of energy, which requires it to consume at least 1.6 million KWh in some quarter of the year. In every quarter, the factory can either run in *eco mode*, in which producing a unit of aluminium consumes 12 kWh, or in normal mode, in which producing a unit of aluminium consumes 15 kWh. In eco mode, the factory can produce at most 10000 units of aluminium in a quarter.

We want to find a *production schedule* that conforms to all of these constraints. There are many ways to model this problem as an SMT instance, and we choose a variant in which we make use of three theories:

- uninterpreted functions
- linear real arithmetic
- linear integer arithmetic

While the setting can be modelled as an SMT instance that uses fewer theories, we will see that the combination of these three theories does not pose a problem for a modern SMT solver.

We use the following function symbols in the SMT formulation of the problem:

- `production` is a function that maps the number of a quarter (in $\{1, 2, 3, 4\}$) to the number of aluminium units produced in the quarter
- `cost` is a function that maps the number of a quarter to the storage and production cost in the quarter
- `energy` is a function that maps the number of a quarter to the number of kWh consumed by the factory
- `stock` is a function that maps the number of a quarter to the number of aluminium units in stock at the end of the quarter
- `x` is a variable that denotes in which quarter the factory consumes at least 1.6 million kWh in order to be considered to be a major consumer.

Note that `x` is a function, too - it is just a 0-ary function, so it does not have any parameters.

In the SMT instance, we need a couple of constraints, namely

- those that connect the `cost` in a quarter with the `energy` and `stock` in a quarter,
- those that connect the processed `energy` in a quarter with the `production` in a quarter,
- one that restricts the overall cost to be at most 5.737 million CUs,
- one that requires the overall production to be at least 1.5 million units of aluminium,
- those that require the production in each quarter not to exceed 1 million units of aluminium,
- those that relate the `stock` in a quarter with the `production` in the quarter and the `stock` at the end of the preceding quarter, and
- those that ensure that the value of x denotes the quarter in which at least 1.6 million kWh are consumed.

In addition, we need some constraints that restrict the values to correct domains (e.g., the production may never be negative). Rather than first stating these constraints in mathematical terms and then translating them into the input format of an SMT solver, let us write the constraints as an SMTLIB2 instance directly. This is a file format that most modern SMT solvers understand, which allows us to try out multiple SMT solvers on our problem.

7.2.1 SMTLIB2 modelling

The SMTLIB2 file format is based on the concept of *S-expressions*. Every statement in an SMT2LIB file is an S-expression, which consists of (1) an opening brace, (2) a list of sub-expressions, and (3) a closing brace. Subexpressions are also enclosed in braces if they form a list. All operator applications are lists that begin with the operators, which are followed by all parameters. The location of line breaks and spaces in the file are not of relevance as long as they do not split up some *token* such as operators and function names.

Our SMT2LIB file starts with a declaration of the constant and function symbols that are specific to our scenario.

```
(declare-fun energy (Int) Real)
(declare-fun production (Int) Real)
(declare-fun stock (Int) Real)
(declare-fun cost (Int) Real)
(declare-const x Int)
```

The first four lines declare the four functions `energy`, `production`, `stock`, and `cost`. The parameter list to these functions consists of a single `Int` value, and their return type is a `Real` value. The last line declares a constant *x* of type Int. It can equivalently declared as a 0-ary function by the line "`(declare-fun x () Int)`".

The first constraint in our SMTLIB2 file now require that the overall cost in all four quarters does not exceed the cost bound defined above:

```
(assert (>= 5737000 (+ (cost 1) (cost 2) (cost 3) (cost 4))))
```

Constraints in SMTLIB2 files are written as `assert` statements. Here, the `assert` statement consists of a comparison between the constant 5736000 and an addition of four numbers. Each of these four numbers is obtained by evaluating the function `cost` with the parameters 1, 2, 3, and 4.

In the same manner, we can require the production in the four quarters to sum up to at least 1.5 million units of aluminium.

```
(assert (>= (+ (production 1) (production 2)
            (production 3) (production 4)) 1500000))
```

Next, we connect the energy consumption in each quarter to the production in each quarter.

```
(assert (or (<= (production 1) 100000)
            (>= (energy 1) (* (production 1) 15) ) ) )
(assert (or (> (production 1) 100000)
            (>= (energy 1) (* (production 1) 12) ) ) )
(assert (or (<= (production 2) 100000)
            (>= (energy 2) (* (production 2) 15) ) ) )
(assert (or (> (production 2) 100000)
            (>= (energy 2) (* (production 2) 12) ) ) )
(assert (or (<= (production 3) 100000)
            (>= (energy 3) (* (production 3) 15) ) ) )
(assert (or (> (production 3) 100000)
            (>= (energy 3) (* (production 3) 12) ) ) )
(assert (or (<= (production 4) 100000)
            (>= (energy 4) (* (production 4) 15) ) ) )
(assert (or (> (production 4) 100000)
            (>= (energy 4) (* (production 4) 12) ) ) )
```

This rather long list of constraints encodes the two modes of operation of the plant in every of the four quarters. Whenever in a quarter the production is 100000 units or less, then every unit requires 12 kWh of energy, whereas when in a quarter, more than 100000 units are produced, 15 kWh of energy are required. We can assume without loss of generality that the factory runs in eco mode whenever the factory does not product too much Aluminium in a quarter to allow operation in eco mode. In any case, no more than 1 million units can be produced in a quarter, which is encoded by the following constraints:

```
(assert (<= (production 1) 1000000))
(assert (<= (production 2) 1000000))
(assert (<= (production 3) 1000000))
(assert (<= (production 4) 1000000))
```

Then, we connect the stock of the factory to the shipment of aluminium and the production of aluminium in the individual quarters. Aluminium in stock is carried over to the respective next quarter.

```
(assert (= (stock 1) (- (production 1) 1000)))
(assert (= (stock 2) (- (+ (stock 1) (production 2)) 1500)))
(assert (= (stock 3) (- (+ (stock 2) (production 3)) 2000)))
(assert (= (stock 4) (- (+ (stock 3) (production 4)) 2500)))
```

Next, we compute the cost induced by production and stock in every quarter. The following constraints use the different prices of energy in the individual quarters.

```
(assert (= (cost 1) (+ (* (energy 1) 0.235) (* 0.1 (stock 1)))))
(assert (= (cost 2) (+ (* (energy 2) 0.245) (* 0.1 (stock 2)))))
(assert (= (cost 3) (+ (* (energy 3) 0.248) (* 0.1 (stock 3)))))
(assert (= (cost 4) (+ (* (energy 4) 0.255) (* 0.1 (stock 4)))))
```

Right now, there are no constraints that force the production in each quarter to be positive. In order to prevent the SMT solver from generating a solution with negative production in some quarter, we add suitable constraints.

```
(assert (>= (production 1) 0))
(assert (>= (production 2) 0))
(assert (>= (production 3) 0))
(assert (>= (production 4) 0))
```

Finally, we want the variable x to tell us in which quarter at least 1.6 million kWh are consumed by the factory.

```
(assert (>= x 1))
(assert (<= x 4))
(assert (> (energy x) 1600000))
```

The SMTLIB2 file now contains all of the constraints. By the definition of the file format, it is necessary to explicitly add a command that asks the solver to check if the SMT instance is satisfiable. Adding a line with the command "`(check-sat)`" suffices. Since we are interested in models for the SMT instance as well, we also add the line "`(get-model)`". Strictly speaking, the command is only valid for satisfiable instances, so we will get an error if the instance turns out to be unsatisfiable.

Calling the SMT solver z3 by Microsoft Research on the SMT instance yields the following output:

```
sat
(model
  (define-fun x () Int
    1)
  (define-fun stock ((x!1 Int)) Real
    (ite (= x!1 1) 299000.0
    (ite (= x!1 2) 397500.0
    (ite (= x!1 3) 1395500.0
    (ite (= x!1 4) 1493000.0
      299000.0)))))
  (define-fun production ((x!1 Int)) Real
    (ite (= x!1 1) 300000.0
    (ite (= x!1 2) 100000.0
    (ite (= x!1 3) 1000000.0
    (ite (= x!1 4) 100000.0
      100000.0)))))
  (define-fun cost ((x!1 Int)) Real
    (ite (= x!1 1) 1087400.0
    (ite (= x!1 2) 333750.0
    (ite (= x!1 3) 3859550.0
    (ite (= x!1 4) 456300.0
      1087400.0)))))
  (define-fun energy ((x!1 Int)) Real
    (ite (= x!1 1) 4500000.0
    (ite (= x!1 2) 1200000.0
    (ite (= x!1 3) 15000000.0
    (ite (= x!1 4) (/ 61400000.0 51.0)
      4500000.0)))))
)
```

The first line of the output states that the problem instance is satisfiable. It is the result of the "`(check-sat)`" command. The S-expression printed afterwards contains a *model* of the SMT instance. The first definition in the model fixes the value of x. It is a function without parameters (i.e., a constant) of `Int` type with a value of 1. The lines afterwards contain a definition of the `stock` function. It is a function from a single `Int` (which is named `x!1` by `z3`) to a `Real` value. The function is defined as a nested `ite` expression. The keyword `ite` stands for *if-then-else*. For the `stock` function, the nested `ite` expression states that if `x!1` is 1, then `stock`(x!1) = 299000.0. For `x!1` being 2, 3, or 4, the values of `stock`(x!1) are 397500, 1395500, and 1493000, respectively. For all other values of `x!1`, the value of `stock`(x!1) is 299000.

The functions `production`, `cost`, and `energy` are defined similarly. While we evaluate all four functions only at the positions 1, 2, 3, and 4, all of the functions have *default values* for all other parameter values. If we sum up the overall cost manually, we see that it is 5737000, which is exactly what we set as upper bound.

The aluminium factory production planning example shows a couple of key features of SMT solvers: we have combined three theories and made use of disjunction between *ground term* (i.e., those without boolean operators). Recall that in ILP and LP solving, the use of disjunction was not possible (although it can be simulated in ILP solving at the expense of introducing additional variables). We have used disjunction in all constraints that connected the production and energy consumption in the individual quarters. In a quite implicit way, we have also used disjunction in the constraints that encode that in some quarter, the energy consumption must be at least 1.6 million kWh. However, this disjunction is camouflaged by the use of uninterpreted functions: we have asked the SMT solver to compute an $\mathtt{x} \in \{1, 2, 3, 4\}$ such that $\mathtt{energy}(\mathtt{x}) \geqslant 1600000$. By writing the requirement in this way instead of requiring that $\mathtt{energy}(1) \geqslant 1600000 \vee \mathtt{energy}(2) \geqslant 1600000 \vee \mathtt{energy}(3) \geqslant 1600000 \vee \mathtt{energy}(4) \geqslant 1600000$, we made the encoding more compact and shifted some reasoning work to the domain of uninterpreted functions. As we will see later in this chapter, there exist efficient decision procedures for uninterpreted functions, which motivates this compact encoding.

7.3 Introduction to eager SMT solving

Literature: *Barrett et al., 2009, Chapter 26.3*

There are two major types of approaches for SMT solving: *eager SMT solving* and *lazy SMT solving*. In the first of these approaches, an SMT problem is translated to another problem that is equi-satisfiable to the original problem while the translated problem can be solved by a single-theory solver (such as a SAT solver). In lazy SMT solving, we use multiple sub-theory solvers at the same time and propagate results between these solvers lazily, i.e., at runtime only after solving has been started. We want to start discussing these two approaches with eager SMT solving.

Eager SMT solving is based on the idea to reduce a given SMT problem to a simpler problem. It allows to take advantage of existing highly-tuned solvers in order to tackle new classes of problems. It is not suitable for all classes of problems. For example, encoding an SMT problem with linear arithmetic over the reals into SAT is not possible in a direct manner. This is simply rooted in the fact that boolean values cannot simulate real values as the latter have an infinite domain. There are however other theories that induce classes of SMT problems that can be encoded relatively easily to SAT, which advocates for tackling them with an eager encoding. For instance, the *pseudo-boolean constraint solving* problem from Chapter 3.1 is often eagerly encoded into SAT.

7.3.1 Example

As the eager encoding of many practical SMT problem classes into SAT is quite complicated, we discuss eager encodings only for a *toy theory* in this section.

Example 24. We consider the toy theory of *partially ordered objects*. Terms consist of 0-ary functions (variables), and we have the binary predicates $\nleqslant$, $\leqslant$, $=$, and $\neq$ that can be applied to variables.

Given a set of variables $\mathcal{V}$ and a set of terms T over $\mathcal{V}$, we say that T is satisfiable if there exists a set $\mathcal{S}$, a mapping $f : \mathcal{V} \rightarrow \mathcal{S}$, and a transitive, reflexive, and antisymmetric relation $R \subseteq \mathcal{S} \times \mathcal{S}$ such that

(1) R is transitive,

(2) R is reflexive,

(3) for all $v \leqslant v'$ in T, we have $(f(v), f(v')) \in R$,

(4) for all $v = v'$ in T, we have $f(v) = f(v')$,

(5) for all $v \neq v'$ in T, we have $f(v) \neq f(v')$, and

(5) for all $v \nleqslant v'$ in T, we do not have $(f(v), f(v')) \in R$

Let for example be $T = x \leqslant y, x \leqslant z, x \nleqslant y, y \nleqslant z$, then T is satisfiable with $S = \{a, b, c\}$, $f(x) = a$, $f(y) = b$, $f(z) = c$, and $R = \{(a,a), (b,b), (c,c), (a,b), (a,c)\}$.

An SMT formula over this theory is, for example, $(x \leqslant y) \wedge (y \leqslant z \vee x \leqslant z) \wedge (z \nleqslant x) \wedge (x \neq z)$. As an example, selecting $\mathcal{S} = \{a, b, c\}$, $f(x) = a$, $f(y) = b$, $f(z) = c$, and $R = \{(a,a), (b,b), (c,c), (a,b), (b,c), (a,c)\}$ satisfies the ground terms $x \leqslant y$, $y \leqslant z$, $z \nleqslant x$, and $x \neq z$ at the same time. As satisfying these ground terms suffices to satisfy the SMT formula, we can conclude that it is satisfiable. ★

This toy theory has an important property: it has the so-called *finite model property*: for every satisfiable set of constraints T, there exists a finite set $\mathcal{S}$ and finite f and R such that T is satisfied on $\mathcal{S}$, f, and R. The finite model property is important for the SMT solving algorithms to be presented in this section.[1]

The toy theory from Example 24 actually has a very strict version of the finite model property. If if is satisfiable, then it is satisfiable with with $|\mathcal{S}| \leqslant n$, where n is the number of variables that occur in the set of constraints. This follows directly from the fact that every subset of a partial order forms a partial order as well, so if a set of constraints has a model, then we can make a (smaller) model of it by removing all elements that are not mapped to by f for some variable.

This observation gives rise to an encoding for the theory of partially ordered objects into SAT: we allocate enough variables to encode a partial order over n objects that satisfy all constraints in T. Without loss of generality, we can assume the ith object to be assigned to the ith variable in T (for all $1 \leqslant i \leqslant n$). To account for the fact that multiple variables can refer to the same object, the relation that we encode does not need to be anti-symmetric – in this way, multiple variables can point to objects that are equivalent.

To encode an SMT formula using the theory of partially ordered objects with n many variables from the theory, we use $n \times n$ many variables $x_{i \leqslant j}$ for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$ and $n \times n$ many SAT variables $y_{i=j}$ for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$.

We first of all have to encode that the R function implemented in the $x_{i \leqslant j}$ operation is reflexive and transitive. We use n many constraints of the form $y_{i \leqslant i}$ and $O(n^3)$ many clauses of the form

$$\bigwedge_{i,j,k \in \{1,\ldots,n\}} (\neg x_{i \leqslant j} \vee \neg x_{j \leqslant k} \vee x_{i \leqslant k})$$

[1]Note that there are theories that do not have the finite model property but for which the satisfiability of a set of ground terms is still decidable. Their discussion is however beyond the scope of this book. The interested reader is referred to Börger et al., 1997 for more information.

for this purpose.

Then, we connect the $x_{i \leqslant j}$ and $y_{i=j}$ variables using $O(n^2)$ constraints of the form

$$\begin{gathered} x_{i \leqslant j} \vee \neg y_{i=j} \\ x_{j \leqslant i} \vee \neg y_{i=j} \\ \neg x_{j \leqslant i} \vee \neg x_{i \leqslant j} \vee y_{i=j} \end{gathered}$$

for all $i, j \in \{1, \ldots, n\}$. Finally, we take the SMT formula and replace all constraints of the form $v_i \leqslant v_j$ by $x_{i \leqslant j}$, all constraints of the form $v_i \nleq v_j$ by $\neg x_{i \leqslant j}$, all constraints of the form $v_i = v_j$ by $y_{i=j}$, and all constraints of the form $v_i \neq v_j$ by $\neg y_{i=j}$ for some renaming of the variables in the SMT formula to $v_1, \ldots, v_n$. The constraints from above together with the modified SMT formula are now satisfiable if and only if the original SMT formula is satisfiable. A model can be easily read off from the SAT assignment.

7.3.2 Discussion

Eager SMT solving allows to make use of the reasoning efficiency of modern (SAT) solvers without the need to extend them. Thus, advances in the area of SAT solving can be used without the need to modify the encoding.

The approach makes most sense if a theory has a compact encoding into SAT. There are some theories that can be eagerly encoded, but do not have straight-forward SAT encodings. In these cases, the SAT instances tend to be huge, which reduces the efficiency of this approach (although *preprocessing* the SAT instance can help to some extent). And then there are even theories that do not allow a relatively straight-forward SMT encoding, for example because the finite model property does not hold. For such cases, an alternative concept to eager SMT solving has been proposed, which we want to discuss next.

7.4 Introduction to lazy SMT solving

In lazy SMT solving, we combine specialized decision procedures for the individual theories to a decision procedure for the combination of the theories. Information is propagated between the solvers in a *lazy* fashion, i.e., only when needed. In this introductory section to lazy SMT solving, we want to study the combination of linear programming (LP) and satisfiability solving (SAT). The lessons learned in the study of this particular case will lay the foundations for the discussion of what is needed to make SMT solving efficient in the remainder of this chapter.

7.4.1 A first SMT solver : SAT+LP

Let us consider the combination of the satisfiability problem and the linear programming problem (without an optimization function). The sub-theory in this context is often called *real-valued linear arithmetic*. Problem instances are boolean formulas (in conjunctive normal form), where the atoms are either literals built from the boolean variables or are comparisons between linear sub-terms over the real-valued variables.

Example 25. An example SAT+LP problem over two real-valued variables $\{x, y\}$ and no boolean variables is

$$\psi = (x \geqslant 5 \vee y \geqslant 10) \wedge (x - y \leqslant -1 \vee x \geqslant 9) \wedge (y - x \leqslant 4 \vee x \leqslant 3). \qquad \bigstar$$

A model of a SAT+LP formula is an assignment to all variables that make the formula satisfied under the standard semantics of the $\vee$, $\wedge$, $\neg$, $+$, $\cdot$, $-$, $\leqslant$, $<$, $=$, $\neq$, $\geqslant$, and $>$ operators. For example, the SAT+LP formula from Example 25 has $\{x \mapsto 7, y \mapsto 10\}$ as one of its models.

The main difference between the LP problem and SAT+LP is that SAT+LP has support for *disjunctions*, where in the LP problem, all linear constraints have to be fulfilled. Without loss of generality, we

Algorithm 8 A first SMT Solver. The parameter $\mathcal{V}$ represents the set of propositional variables (including the variable $\mathcal{V}_L$ for the linear constraints), ψ is the CNF formula over $\mathcal{V}$ that represents the problem, and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment to the propositional variables.

```
1: function SEARCH($\mathcal{V}$,$\psi$,$\psi'$,$A$)
2:     for all assignments $v_k = b_k$ implied by $(\psi, A)$ by unit propagation do
3:         $A \leftarrow A \cup \{v_k \mapsto b_k\}$
4:     end for
5:     if some clause in $\psi$ is falsified by $A$ then
6:         return $\varnothing$
7:     end if
8:     if $|A| = |\mathcal{V}|$ then
9:         $B \leftarrow$ solveLP($\{x \in \mathcal{V}|_L : A(x) = \textbf{true}\}$)
10:        if $B \neq \varnothing$ then
11:            return $B \cup A|_{\mathcal{V} \setminus \mathcal{V}_L}$
12:        else
13:            return $\varnothing$
14:        end if
15:    end if
16:    Pick a variable $v$ that is not yet in the domain of $A$
17:    $A' \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto \textbf{false}\}$)
18:    if $A' \neq \varnothing$ then return $A'$
19:    $A' \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto \textbf{true}\}$)
20:    if $A' \neq \varnothing$ then return $A'$
21:    return $\varnothing$
22: end function
```

assume in the following that the atoms from the theory do not occur negated in the formula, i.e., that all negations are pushed into the comparison itself. So, for example, the clause $(x - y \leqslant -1 \vee \neg(x < 9))$ would need to be translated to $(x - y \leqslant -1 \vee x \geqslant 9)$ first.

The SAT+LP problem can be solved by integrating a SAT solver with an LP solver. Algorithm 8 describes how this can be done on the algorithmic level. It is essentially a DPLL-style SAT solver in which not only the boolean variables are treated as such, but also the ground terms from the LP theory. Whenever a complete SAT assignment is found, the LP solver is called to check if the ground terms that are chosen to hold by the SAT solver are actually *consistent*, i.e., whether a model for the real-valued variables exists that satisfies all constraints that A maps to **true**. Whenever this is the case, then a model for all variables together has been found, and otherwise, the Search function returns no model, so that the search process can continue. By pushing all negations into the ground terms themselves, we made sure that there is no necessity to add to the LP instance that the terms that are mapped to **false** should not hold in the model.

Example 26. Reconsider Example 25. Algorithm 8 could compute the candidate assignment $\{x \geqslant 5 \mapsto \textbf{true}, y \geqslant 10 \mapsto \textbf{false}, x - y \leqslant -1 \mapsto \textbf{false}, x \geqslant 9 \mapsto \textbf{true}, y - x \leqslant 4 \mapsto \textbf{false}, x \leqslant 3 \mapsto \textbf{true}\}$. In Line 9 of algorithm, it is then checked if the LP problem with the constraints

- $x \geqslant 5$
- $x \geqslant 9$
- $x \leqslant 3$

has a solution. Since the first and the last constraint are contradictory, the LP solver does not find a solution.

If later in the search, the Search procedure finds the (candidate) assignment $\{x \geqslant 5 \mapsto \textbf{true}, y \geqslant 10 \mapsto \textbf{false}, x - y \leqslant -1 \mapsto \textbf{false}, x \geqslant 9 \mapsto \textbf{true}, y - x \leqslant 4 \mapsto \textbf{true}, x \leqslant 3 \mapsto \textbf{false}\}$, then solveLP is called on an LP problem with the constraints

- $x \geqslant 5$

- $x \geqslant 9$
- $y - x \leqslant 4$,

which admit the solution $\{x \mapsto 9, y \mapsto 9\}$. So the SMT solver finds a satisfying assignment to ψ at that point. ★

In contrast to the standard DPLL algorithm for SAT (Algorithm 2 on page 26), Algorithm 8 does not apply the pure literal rule. We introduced it as a sound rule to speed up SAT solving in Chapter 2.4.1.3. The ommission of the pure literal rule in Algorithm 8 is rooted in the fact that it is actually *unsound* in the SAT+LP case.

To see this, take for example the specification

$$(x \geqslant 5 \vee y \geqslant 3) \wedge (x \leqslant 4 \wedge y \leqslant 2).$$

None of the ground terms is a negation of another ground term. The SAT solver sees four unconnected terms to which **true** could be assigned by the pure literal rule. After applying the pure literal rule to select $x \geqslant 5$ to hold and $x \leqslant 4$ to hold, we have an inconsistent set of constraints to fulfill. So applying this rule can lead to inconsistencies. This is a problem as after finding an inconcistency without having made decisions, the solver would deduce unsatisfiability without trying further assignments. So if the LP constraints are inconsistent, it incorrectly determines that the SMT formula is unsatisfiable.

With the unit propagation rule, this problem does not exist. For any partial assignment A of the boolean variables and the theory ground terms to **true** and **false** that can be extended to an assignment A' for all boolean and non-boolean variables such exactly the ground terms that are valid over the assignment to the non-boolean variables are mapped to **true**, unit propagation can never deduce a fact that is not contained in A', simply because unit propagation only derives facts that are implied by A.

7.4.2 Improved SAT+LP

Algorithm 8 has the problem that it only calls the LP solver when a complete assignment to all ground terms (and boolean variables) has been found. Thus, inconsistencies that are introduced early in the search process (e.g., after two decisions have been made and the partial assignment is $\{x \geqslant 4 \mapsto$ **true**, $x < 3 \mapsto$ **true**$\}$ for some SMT formula in which both $x \geqslant 4$ and $x < 3$ are ground terms) are only detected after **true** or **false** have been assigned to all ground terms. This is normally inefficient. Algorithm 9 shows a modified SMT decision procedure for SAT+LP in which this problem has been fixed. It is essentially the same decision procedure, except that this time, the procedure calls the theory solver early in order to find conflicts early in the search process.

7.4.3 Adding unit propagation & learning

While Algorithm 9 is already an improvement, it does not make much use of unit propagation, which is one of main pillars of the efficiency of search-based SAT solving. The algorithm does not detect implementations between ground terms and hence has to wait for a conflict to become apparent before undoing incorrect decisions. This fact not only restricts the usefulness of unit propagation, but also limits the use of *clause learning*. Recall from Chapter 2.4.2 that effective clause learning requires building an implication graph, which encodes which literals have been propagated for which reason. If unit propagation is not performed on the theory level, then this means that conflict clauses must be large, which reduces the efficiency of Algorithm 9 compared to standard SAT solving.

Modern SMT solvers use theory sub-solvers for linear real arithmetic that helps the outer SAT solver with unit propagation. They allow to infer ground terms that are implied by the ground terms that are deemed to be satisfiable by the ground terms mapped to **true** in a partial assignment. When solving, for example, the SMT instance

$$(x \geqslant 3 \vee x \leqslant -3) \wedge (y \geqslant 5 \vee y \leqslant -3) \wedge (x + y \leqslant 7 \vee y \geqslant 6),$$

Algorithm 9 A revised version of our first SMT Solver. The parameter $\mathcal{V}$ represents the set of propositional variables (including the variable $\mathcal{V}_L$ for the linear constraints), ψ is the CNF formula over $\mathcal{V}$ that represents the problem, and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment to the propositional variables.

```
function Search($\mathcal{V}$,$\psi$,$A$)
    for all assignments $v_k = b_k$ implied by $(\psi, A)$ by unit propagation do
        $A \leftarrow A \cup \{v_k \mapsto b_k\}$
    end for
    if some clause in $\psi$ is falsified by $A$ then
        return $\varnothing$
    end if
    $B \leftarrow$ solveLP($\{x \in \mathcal{V}|_L : A(x) = \mathbf{true}\}$)
    if $B = \varnothing$ then
        return $\varnothing$
    end if
    if $|A| = |\mathcal{V}|$ then
        return $B \cup A|_{\mathcal{V}\setminus\mathcal{V}_L}$
    end if
    Pick a variable $v$ that is not yet in the domain of $A$
    $A' \leftarrow$ Search($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto \mathbf{false}\}$)
    if $A' \neq \varnothing$ then return $A'$
    $A' \leftarrow$ Search($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto \mathbf{true}\}$)
    if $A' \neq \varnothing$ then return $A'$
    return $\varnothing$
end function
```

the partial valuation $\{x \geqslant 3 \mapsto \mathbf{true}, y \geqslant 5 \mapsto \mathbf{true}\}$ can immediately be extended by $x + y \leqslant 7 \mapsto \mathbf{false}$, as the partial valuation implies that $x + y \leqslant 7$ does not hold. Thus, the theory solver could add this to the partial valuation and then standard unit propagation could derive that $y \geqslant 6$ needs to hold as well. For all practical means, finding out that the current partial valuation implies that some ground term must hold or not hold is the same as unit propagation, and when building the conflict graph for clause learning, we would connect such derived facts with their implicants.

The discussion of how exactly such implications can be computed by an LP solving procedure is beyond the scope of this book. However, we can study the incorporation of generalized unit propagation by means of a simpler theory. This time, the theory is concerned with only boolean variables, so that the SAT part of the SMT solver uses the same variables as the theory solver.

Definition 15. *A* SAT+XOR problem instance *consists of a set of variables $\mathcal{V}$, a conjunction of clauses ψ, and a set of* parity constraints *of the form $v_1 \oplus \ldots \oplus v_k = b$, where $b = 0$ or $b = 1$. A valuation $V : \mathcal{V} \to \{\mathbf{false}, \mathbf{true}\}$ is a model of the problem instance if $V \models \psi$ and for all constraints $v_1 \oplus \ldots \oplus v_k = b$, we have that the number of variables v_i in $v_1, \ldots, v_k$ with $V(v_i) =$ **true** modulo 2 is b.*

The $\oplus$ in Definition 15 is an exclusive or operator (if we define **false** to be equivalent to 0 and **true** to be equivalent to 1).

While it is possible to encode the parity constraints as SAT clauses (so that no specialized theory solver is necessary), sets of parity constraints have the property that we can apply Gaussian elimination to compute implicants of partial valuations to the variables. This is because parity constraints are linear equations in the $\mathbb{Z}_2$ ring, consisting only of the values 0 and 1.

Example 27. Consider the two parity constraints $a \oplus b \oplus d = 1$ and $b \oplus c \oplus d = 1$. Given the partial valuation $\{a \mapsto \mathbf{false}\}$, we cannot infer the values of b or d. Hence, unit propagation could not extend this partial valuation after translating all constraints to clauses. We can however add the two constraints in $\mathbb{Z}_2$ to obtain:

$$\begin{array}{cccccccccc} & a & \oplus & b & & & \oplus & d & = & 1 \\ + & & & b & \oplus & c & \oplus & d & = & 1 \\ \hline = & a & & & \oplus & c & & & = & 0 \end{array}$$

Thus, the partial valuation $\{a \mapsto \mathbf{false}\}$ can be safely extended to $\{c \mapsto \mathbf{false}\}$. ★

Algorithm 10 A solver for the theory consisting of SAT and XOR clauses. The parameter $\mathcal{V}$ represents the set of propositional variables, ψ is the CNF formula over $\mathcal{V}$ that represents the problem, ψ' is a conjunction of XOR clauses, and $A : \mathcal{V} \rightharpoonup \mathbb{B}$ is the current partial assignment to the propositional variables.

```
1: function SEARCH($\mathcal{V}$,$\psi$,$A$)
2:     for all assignments $v_k = b_k$ implied by $(\psi, A)$ by unit propagation do
3:         $A \leftarrow A \cup \{v_k \mapsto b_k\}$
4:     end for
5:     for all assignments $v_k = b_k$ implied by $(\psi, A)$ by pure literal elimination do
6:         $A \leftarrow A \cup \{v_k \mapsto b_k\}$
7:     end for
8:     for all assignments $v_k = b_k$ implied by $(\psi', A)$ by gaussian elimination do
9:         $A \leftarrow A \cup \{v_k \mapsto b_k\}$
10:    end for
11:    if some clause in $\psi$ is falsified by $A$ or $A$ is inconsistent then
12:        return $\emptyset$
13:    end if
14:    if $|A| = |\mathcal{V}|$ then
15:        return $A$
16:    end if
17:    Pick a variable $v$ that is not yet in the domain of $A$
18:    $A' \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ false$\}$)
19:    if $A' \neq \emptyset$ then return $A'$
20:    $A' \leftarrow$ SEARCH($\mathcal{V}$,$\psi$,$A \cup \{v \mapsto$ true$\}$)
21:    if $A' \neq \emptyset$ then return $A'$
22:    return $\emptyset$
23: end function
```

By performing Gaussian elimination when checking a partial assignment for consistency during runtime of the SMT solver, we can infer all implicants of the set of parity constraints of a partial valuation in polynomial time. Algorithm 10 gives the resulting decision procedure.

It is worthwhile mentioning that although we introduced XOR constraints as an example for how a sub-theory can give rise to unit propagation in the enclosing SAT solver, adding special support for XOR clauses has been proposed in the past to make tackling cryptographic problems with SAT solvers more efficient (Soos et al., 2009). The use of the exclusive or operator is, for example, quite common in encryption algorithms or cryptographic hash functions. Also, older versions of the `minisat` solver (Eén and Sörensson, 2010) have a well-defined interface for adding support for additional constraint types, which has however only been used quite seldom. Ehlers and Finkbeiner (2010), for example, describe an automata-theoretic application of this interface.

It should also be noted that in order make most use of the SAT solver's reasoning power, supporting unit propagation alone is not enough. Rather, the sub-theory solver needs to be able to give reasons for why a unit clause was propagated when it is propagated. This is important in order to make *clause learning* work well, as its efficiency depends on the ability to make small cuts though the conflict graph. If the sub-theory solver cannot give a reason for unit propagation, then in the conflict graph, the newly induced literal depends on *all* variable values that have been fixed earlier and that occur in the XOR clauses, which would lead to huge cuts in the conflict graph. The interested reader is referred to Soos et al. (2009) for details on how to avoid this for the sub-theory of XOR clauses.

7.5 Combining theories in lazy solvers

The power of practical satisfiability modulo theory solving lies in the fact that multiple theories can be combined in the same SMT formula, which allows to represent many practical problems concisely and in a shape that is amenable to efficient reasoning. We have seen in the preceding section that combining

SAT with another theory is relatively straight-forward, as we could just let the SAT solver take care of selecting which ground term should hold, and then leave everything else to the theory solver. When combining multiple theories in a tight way, this idea is no longer applicable. For example, the SMT formula

$$f(2 \cdot x) = x \wedge (f(x) = f(y) \vee x \geqslant 2)$$

combines the theory of uninterpreted functions with linear arithmetic. Separating them on the Boolean level is not possible, so a different approach is needed. The *Nelson-Oppen* framework, which we discuss in this section, is based on propagating equalities of terms between specialized solvers for the individual theories.

To keep the presentation simple, we discuss Nelson-Oppen without a surrounding SAT solver, so we only consider conjunctions of ground terms in the following. However, the Nelson-Oppen procedure can be combined with a SAT solver in the same way as we combined SAT+LP solving in the preceding section.

In order to be able to discuss illustrative examples for the Nelson-Oppen procedure, we need at least two theories that make sense to be combined. So before proceeding with the description of the Nelson-Oppen procedure, let us take the time to study the theory of uninterpreted functions, which is a useful complement to many other theories and has a relatively simple decision procedure.

7.5.1 Excursion: The theory of uninterpreted functions

We have already specified the theory of uninterpreted functions in Definition 14. In the example in Section 7.2, we used them for a relatively simply purpose, namely to compress the representation of the solution and to state in a compact way that in one quarter, the energy consumption has to be above a certain level. Uninterpreted functions however have more uses in practice. For example, Kroening and Strichman (2008, p. 62) note:

> *Uninterpreted functions are widely used in calculus and other branches of mathematics, but in the context of reasoning and verification, they are mainly used for simplifying proofs. Under certain conditions, uninterpreted functions let us reason about systems while ignoring the semantics of some or all functions, assuming they are not necessary for the proof.*

Since one of the prime application areas of SMT solvers is verification of software systems, this statement explains why most practical SMT solvers nowadays have support for uninterpreted functions. Uninterpreted functions for example allow to abstract from the behavior of some of the functions of a program. When performing the abstraction, we only retain the information that whenever the program function is called, it behaves in the same way, i.e., it retains the same output value(s) for the same input values. As we will see next, uninterpreted functions also have a relatively simple decision procedure, and the theory is relative simple to implement in an SMT solver. Let us now discuss how this can be done, i.e., how we can check for a conjunction of ground terms if they are consistent.

Terms and sub-terms can either be equivalent or inequivalent. There are no other properties of the terms that are to be checked. Thus, in order to determine if a set of ground terms in the theory of uninterpreted functions is consistent, it suffices to compute all equalities between terms and their sub-terms, and then check if they are inconsistent with the inequalities specified by the ground terms.

Example 28. Consider the following conjunction to ground terms from the theory of uninterpreted functions:

$$f(a) = f(b) \wedge b = c \wedge f(a) \neq f(c)$$

In this example, f is a unary function, a, b, and c are 0-ary functions (i.e., constants), and the set of sub-terms is $\{a, b, c, f(a), f(b), f(c)\}$. The fact that $b = c$ holds implies that $f(b) = f(c)$ holds as well. Furthermore, this fact together with $f(a) = f(b)$ also implies that $f(a) = f(c)$. This is in contrast to the last ground term, which states the opposite. Thus, this conjunction of ground terms is inconsistent. ★

We can concretize the idea for inconsistency checking used in Example 28 into a *congruence closure* algorithm, which we want to sketch next:

1. Enumerate all sub-terms of every ground term.
2. Build a partition of all sub-terms that initially puts all sub-terms into different equivalence classes.
3. For all ground terms $t = t'$, merge the partition elements of t and t'.
4. Execute the following step until the partition does not change any more:
   - For all n-ary function symbols f with $(t_0, \ldots, t_{n-1})$ and $(t'_0, \ldots, t'_{n-1})$ being n-tuples of sub-terms such that for all $0 \leqslant i < n$, we have that t_i and t' are in the same partition elements, we merge the partition elements of $f(t_0, \ldots, t_{n-1})$ and $f(t'_0, \ldots, t'_{n-1})$.
5. Check if for some ground term $t \neq t'$, we have that t and t' are in the same partition element. Return that the set of constraints is *unsatisfiable/inconsistent* if such a ground term is found, and that it is *satisfiable/consistent* otherwise.

An algorithm built from this sketch runs in time polynomial in the number of terms for all fixed maximum arities of the function symbols, and it was first described (in a more concise and slightly more general form) by Shostak (1978). The choice to return *satisfiable* in the algorithm whenever no inconsistent inequality term is found is rooted in the idea that the algorithm computes all equalities between the terms that are implied by the equality ground terms. If we assign different values to all equivalence classes and return for every class the respective value, we obtain a *maximally diverse model* that satisfies as many inequality terms as possible. Thus, it suffices to checks if such a model violates an inequality constraint, which is exactly what happens in the last step of the algorithm.

Example 29. We want to examine the behavior of the congruence closure algorithm on the following set of ground terms:

- $c = f(g(c, f(f(a))))$
- $b = f(f(c))$
- $a = f(g(g(a,b), d))$
- $c = f(g(a,a))$
- $f(g(a,a)) = g(a,b)$
- $d = f(f(a))$
- $b \neq d$

There are 14 different sub-terms: $a, b, c, d, g(c, f(f(a))), f(g(c, f(f(a)))), f(g(g(a,b),d)), f(c), f(f(c)), g(a,b)$, $g(g(a,b),d)$, $g(a,a)$, $f(g(a,a))$, $f(a)$, $f(f(a))$. We start by putting all of them into individual equivalence classes and then merge them according to the equalities. This yields the following partitioning:

a $f(g(g(a,b),d))$	b $f(f(c))$	c $g(a,b)$ $f(g(a,a))$ $f(g(c,f(f(a))))$	d $f(f(a))$	$g(c,f(f(a)))$
$g(g(a,b),d)$	$g(a,a)$	$f(a)$	$f(c)$	

We now execute the part of the algorithm sketch in which new equalities are found by matching sub-terms. The first match that we found is for the 2-ary function g using $g(a,b)$ and d for the terms t_0 and t_1, and c and $f(f(a))$ as terms t'_0 and t'_1. We can thus conclude that $g(g(a,b),d) = g(c, f(f(a)))$. So we have to merge their equivalence classes and obtain:

a $f(g(g(a,b),d))$	b $f(f(c))$	c $g(a,b)$ $f(g(a,a))$ $f(g(c,f(f(a))))$	d $f(f(a))$	$g(c,f(f(a)))$ $g(g(a,b),d)$
$g(a,a)$	$f(a)$	$f(c)$		

The second match that we find is for function f, which is unary, with terms $t_0 = g(c, f(f(a)))$ and $t'_0 = g(g(a,b),d)$. Thus, we can merge the equivalence classes of $f(g(c, f(f(a))))$ and $f(g(g(a,b),d))$. This leads to the following set of equivalence classes:

a	b	d	$g(c, f(f(a)))$	$g(a,a)$	$f(a)$	$f(c)$
c	$f(f(c))$	$f(f(a))$	$g(g(a,b),d)$			
$g(a,b)$						
$f(g(g(a,b),d))$						
$f(g(a,a))$						
$f(g(c, f(f(a))))$						

Since $a = c$, we can now also merge $f(a)$ and $f(c)$ into the same equivalence class, by the same line of reasoning as before. We obtain:

a	b	d	$g(c, f(f(a)))$	$g(a,a)$	$f(a)$
c	$f(f(c))$	$f(f(a))$	$g(g(a,b),d)$		$f(c)$
$g(a,b)$					
$f(g(g(a,b),d))$					
$f(g(a,a))$					
$f(g(c, f(f(a))))$					

Since $f(a) = f(c)$, we can now also merge the equivalence classes of $f(f(c))$ and $f(f(a))$. This leads to our final set of equivalence classes, for which no more classes can be merged:

a	b	$g(c, f(f(a)))$	$g(a,a)$	$f(a)$
c	d	$g(g(a,b),d)$		$f(c)$
$g(a,b)$	$f(f(a))$			
$f(g(g(a,b),d))$	$f(f(c))$			
$f(g(a,a))$				
$f(g(c, f(f(a))))$				

In the final step of the algorithm, we check if any inequality in the set of ground terms is inconsistent with the equality classes. Since $b \neq d$ is part of the set of ground terms, but b and c are in the same equality class, we conclude that the overall set of ground terms is inconsistent. ★

The theory of uninterpreted functions is often used in conjunction with *predicates*, i.e., functions with the co-domain $\{\textbf{false}, \textbf{true}\}$. The decision procedures for uninterpreted functions can easily be extended to also support predicates. For this, we do two things:

- We introduce designated 0-ary functions **true** and **false**.
- We introduce a new 1-ary function symbol f_P for every predicate P, and replace every ground term $P(t)$ for some term t by $f_P(t) = \textbf{true}$ and every ground term $\neg P(t)$ for some term t by $f_P(t) = \textbf{false}$.
- After executing the above congruence closure algorithm on the modified problem instance, we conclude that the problem instance is satisfiable if and only if the congruence closure algorithm not already determined it to be inconsistent *and* **true** and **false** have not been merged into the same equivalence class.

The trick that we use in this approach is that we do not explicitly search over an assignment of the terms to $\mathbb{B}$ for every predicate, but rather make use of the congruence closure algorithm. By introducing **false** and **true** as atoms, we can incorporate the part of the predicates' assignment that has already been fixed.

7.5.2 Nelson-Oppen

Literature: *Kroening and Strichman, 2008, Chapter 10.3*

The Nelson-Oppen framework connects theories by propagating *equalities* between them. Its main ideas form the conceptual basis of modern SMT solvers, and it has relatively few requirements on the properties of the theories in order to be applicable. The following definition lists them:

Definition 16. *Given a list of theories $T_1, \ldots, T_n$, we call they* compatible with the Nelson-Oppen procedure *if they fulfil the following requirements (taken from Kroening and Strichman, 2008, Definition 10.5, with modifications given by Barrett et al., 2009, Chapter 26.6.1):*

- *$T_1, \ldots T_n$ are first-order theories with equality without quantifiers*
- *There is a decision procedure for each of the theories $T_1, \ldots, T_n$ that can infer equality*
- *The signatures of $T_1, \ldots, T_n$ are disjoint except for the 0-ary theory symbols and the shared 2-ary equality operator*
- *$T_1, \ldots, T_n$ are theories that are interpreted over infinite domains*

Both the theories of linear real arithmetic and the theory of uninterpreted functions that we have seen so far satisfy these requirements. While we did not discuss extensions of the LP solver that can infer that two variables' values are equal from the other constraints, it can be extended accordingly. A simple way for doing so is to check for all pairs of variables v and v' whether $v < v'$ or $v > v'$ are satisfiable together with the other LP ground terms. Whenever this not the case for both such extensions, then the variables v and v' must have the same value.

The disjointness requirement for the theory signatures in the requirement list may come as a surprise as one of the motivations for SMT solving was the ability to tightly integrate terms from different theories. For example, in the specification

$$g(x) \geqslant 3 \wedge f(x) \leqslant 3,$$

we mix uninterpreted functions and linear arithmetic. But this is no contradiction: while f and x are terms in the theory of uninterpreted functions, for the theory of linear real arithmetic, $g(x)$ can be seen as a constant, i.e., a 0-ary term. The following *purification procedure*, which is the first step of the Nelson-Oppen procedure, showcases why this combination is indeed unproblematic:

Definition 17 (Purification). *Let an SMT formula ψ over the theories $T_1, \ldots, T_n$ be given. We say that a subterm is from theory T_i if its outermost operator is part of T_i's signature. If a term from theory T_i also uses operators from T_j for $i \neq j$, then we say that the (outermost) term starting with an operator from T_j is an* alien subterm.

When purifying an SMT formula, we replace all occurrences of alien subterms t' by a new variable v while adding $v = t'$ as a new ground term. We repeat this process until no more such replacements can be performed.

Example 30. Take, for example, the SMT formula

$$(g(x) \leqslant y) \wedge (f(f(2 \cdot x)) \geqslant g(y+1))$$

over the theories of uninterpreted functions and linear real arithmetic. Many terms in it have alien sub-terms. For example, the ground term $g(x) \leqslant y$ applies an uninterpreted function g to x and then preforms a real-valued comparison on the result. We substitute the new variable a for $g(x)$ and similarly replace $2 \cdot x$ by a new variable b and $y + 1$ by a new variable c. We obtain:

$$(a \leqslant y) \wedge (f(f(b)) \geqslant g(c)) \wedge (a = g(x)) \wedge (b = 2 \cdot x) \wedge (c = y + 1)$$

There are still some alien subterms in the ground terms, so we continue with the process:

$$(a \leqslant y) \wedge (d \geqslant e) \wedge (a = g(x)) \wedge (b = 2 \cdot x) \wedge (c = y + 1) \wedge (d = f(f(b))) \wedge (e = g(c))$$

This time, no more purification is possible, so the process terminates. ★

After purification, all ground terms belong to only one theory each, except for possibly the ones that concern the equality of two constants, which are by Definition 16 supported by all theories. We can now apply the core part of the Nelson-Oppen procedure.

For simplicity, we only consider the case of applying the procedure on *convex theories* here. We will discuss later how the procedure for the convex case can be lifted to the general case. Convexity is defined as follows:

Definition 18 (Kroening and Strichman, 2008, Chapter 10.3). *A theory is* convex *if for every set of ground atoms φ that are all in the theory, we have the following:*

Whenever for some finite set of ground terms $g_1, \ldots, g_n$, we have that (a) all of their outermost operators is the equality operator and (b) the validity of φ on a model implies that one of the ground terms $g_1, \ldots, g_n$ also holds on the model, then there exists a ground term g_i in this set such that the validity of φ also implies that g_i holds.

Let us consider two examples:

Example 31. The theory of linear integer arithmetic is *not* convex. For example, the constraints

$$(x \geqslant 4) \wedge (x < 6)$$

imply that either $x = 4$ or $x = 5$ hold, but they do not imply one of them (directly). ★

Example 32. The theory of linear real arithmetic is convex. If a set of constraints implies a set of equality ground terms G, then by the convexity of the solution space, it holds that all linear interpolations between terms in G are also ground terms implied by the constraints. As there are infinitely many different interpolations if not all terms in G are equivalent or trivially satisfied, the only way for G to be finite is for all non-trivially implied terms in G to be the same. In this case, we have one element in G that is also implied by the constraints. ★

The main loop of the Nelson-Oppen procedure on a set of ground terms after purification consists of the following steps:

1. Check the satisfiability of φ for every theory separately, ignoring the ground terms of the respective alien theories.
2. If for some theory, the result was "unsatisfiable", abort with an "unsatisfiable" result.
3. Derive the *induced equalities* in all sub-theories and add them to φ
4. Repeat all steps until no more equalities are found.

If the loop terminates because no more equalities are found, the procedure returns "satisfiable".

Example 33. We consider the following conjunction of ground terms as an example for the Nelson-Oppen procedure:

$$\begin{aligned}&(b = c - 1) \wedge (c \geqslant f(c) + 1) \wedge (c \leqslant f(a) + 1) \wedge (f(b) > f(a))\\ \wedge&(f(a) = f(f(a))) \wedge (f(f(f(a))) = f(c))\end{aligned}$$

The theories involved are the theory of uninterpreted functions and linear real arithmetic, both of which are convex. The first step in the Nelson-Oppen procedure is to apply the purification procedure. This step introduces three additional variables, which we call x, y, and z in the following. We obtain the following purified set of ground terms:

$$\begin{aligned}&(b = c - 1) \wedge (c \geqslant y + 1) \wedge (c \leqslant x + 1) \wedge (z > x) \wedge (f(a) = f(f(a)))\\ \wedge&(f(f(f(a))) = f(c)) \wedge (x = f(a)) \wedge (y = f(c)) \wedge (z = f(b))\end{aligned}$$

For notational convenience, it makes sense to split up the ground terms according to their theories:

UIF: $f(a) = f(f(a))$

UIF: $f(f(f(a))) = f(c)$

UIF: $x = f(a)$

UIF: $z = f(b)$

UIF: $y = f(c)$

LRA: $b = c - 1$

LRA: $c \geqslant y + 1$

LRA: $c \leqslant x + 1$

LRA: $z > x$

The Nelson-Oppen procedure proceeds with checking the two theories individually for inconsistency of their respective ground terms and deriving additional equalites, which can then be propagated to both theory solvers. In the first round, the arithmetic solver does not find any additional equivalence, but using all ground terms from the theory of uninterpreted functions, we find that $y = f(c) = f(f(f(a))) = f(f(a)) = f(a) = x$. So the following new ground term is deduced:

LRA+UIF: $x = y$

This fact has an effect in the LRA solver: since we now know that $x = y$ holds, the second and third LRA constraints together imply that $c = x + 1$. But this is not an equality that can or needs to be propagated to the uninterpreted function solver, as it does not concern the quality of variables. However, the solver can also combine this with the fact that $b = c - 1$ and infer:

LRA+UIF: $b = x$

Additionally, $y = b$ could be inferred, but that equality is rendundant as it is inferred by $x = y$ and $b = x$. The new ground term $b = x$ allows the uninterpreted function solver to conclude that $z = f(b) = f(x) = f(f(a)) = f(a) = x$, so the following equality is added:

LRA+UIF: $z = x$

In addition, $y = z$ and $b = z$ could be inferred, but that would be redundant again. Since we have the constraint $z > x$ in the list of ground terms, the LRA solver now reaches the unsatisfiability conclusion. Hence, the Nelson-Oppen procedure determines that the set of ground terms is inconsistent. ★

We can integrate the Nelson-Oppen procedure into a SAT solver in the same way as we integrated an LP solver into SAT. The SAT solver selects which ground terms should hold, and whenever a (partial) variable valuation is to be checked for consistency, the Nelson-Oppen procedure is called.

7.5.3 Using non-convex theories

One of the requirements of the Nelson-Oppen procedure is that the theories have to be convex. Let us look at by means of an example why it does not work with non-convex theories.

Example 34. Consider the following SMT formula over the the theories of linear integer arithmetic and uninterpreted functions:

$$a \geqslant 3 \wedge a \leqslant 4 \wedge b \geqslant 3 \wedge b \leqslant 4 \wedge f(3) = f(4) \wedge f(a) \neq f(b)$$

The formula is purified to:

$$a \geqslant 3 \wedge a \leqslant 4 \wedge b \geqslant 3 \wedge b \leqslant 4 \wedge f(x) = f(y) \wedge f(a) \neq f(b) \wedge x = 3 \wedge y = 4$$

Calling Nelson-Oppen on these ground terms yields that they are consistent: none of the theories can infer any new quality, and the disequality between $f(x)$ and $f(y)$ does not contradict any of the existing atoms.

This satisfiability result is not correct. The first four constraints imply that a and b are in $\{3, 4\}$. The constraints furthermore say that $f(3) = f(4)$. But that also means that $f(a) = f(b)$, as either a and b are unequal, in which case $f(3) = f(4)$ implies that $f(a) = f(b)$, or we have $a = b$, in which case we also have $f(a) = f(b)$. This is inconsistent with $f(a) \neq f(b)$ and hence the formula is actually unsatisfiable. ★

So it suffices that just one sub-theory is not convex for the Nelson-Oppen procedure to return spurious consistency/satisfiability results. This problem can however fixed relatively easily if the procedure is used by an enclosing SAT solver. For that to work, the non-convex sub-theory solvers need to be able to infer disjunctions of equalities. Recall that for theories that do not conform to Definition 18 (i.e., that are not convex), a set of terms may imply a disjunction of more than one other equality. If the solver for the non-convex theory can infer such disjunctions, then this disequality could be added as a SAT clause over these equality terms to the SMT instance. The SAT solver can then take care of iterating through the possible disjuncts. This yields a sound and complete decision procedure, provided that the other requirements of the Nelson-Oppen procedure are met.

It should be noted that this change makes the SAT instance grow over time, both in the number of clauses and variables/ground terms. However, whenever the solver back-tracks a decision that led to a disjunction of equalities being inferred, the added clause also has to be removed. Furthermore, the sub-theory solver needs a way to detect when disjunction actually has to be inferred, as otherwise we get a substantial blow-up of the SMT instance even in cases where this is actually not needed. For example, if $(x \geqslant 0) \wedge (x < 10000)$ are terms in an SMT instance, the clause $x = 0 \vee x = 1 \vee \ldots \vee x = 9999$ should only be added to the instance if needed.

7.5.4 Notes & conclusion

We have seen in this section how multiple theories can be combined in the same reasoner. The Nelson-Oppen procedure that we discussed forms the conceptual basis for most modern SMT solvers. There are many details that we left out for conciseness reasons:

- Our treatment of the Nelson-Oppen procedure is not complete enough to show how it can be combined with *learning*: we did not show how to perform unit propagation in a way that also inequalities between terms can be inferred, and we did not touch the topic of computing a useful conflict graph.
- So far, we only considered theories with an infinite data domain (as required by the Nelson-Oppen procedure). However, there are also other theories without this property (such as the one discussed in Section 7.6.1) that are nevertheless supported in many modern SMT solver.
- Some theories have highly complex models. We also did not cover how to cope with them in an efficient way.
- In order to apply Nelson-Oppen to non-convex theories, we had to assume that the non-convex solver can infer a disjunction between additional equality terms. How this works, for example, for the theory of linear integer arithmetic, is not obvious, even after discussing ILP solving in Chapter 5.5.

7.6 Some standard SMT theories

Many state-of-the-art SMT solvers, such as `z3` and `cvc4` (Barrett et al., 2011) support a multitude of different theories that can be combined in the SMT formulas. We have already gotten to know some of them, such as linear integer and real arithmetic and uninterpreted functions. But there are more theories that many solvers offer.

7.6.1 Bitvectors

Most advances in SMT solving are driven by the prospect to improve the performance of *program verification*. It is not surprising that theories that are especially adapted towards this application are often part of modern SMT solvers. One such example is the theory of *bitvectors*. This theory uses a new data type that represents integer numbers with a fixed number of bits. Operations such as addition and multiplication operate on this data type in the same way as they are commonly implemented in real-world computer hardware.

As an example, consider the following SMT specification file that is compatible with the SMT solver `z3`.

```
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))

(assert (bvult b #x00000020 ))
(assert (bvugt c #x00000050 ))
(assert (bvugt (bvsub b #x00000002 ) c))

(check-sat)
(get-model)
```

The variables b and c are declared as constants (0-ary functions) that have a 32 bit bitvector as co-domain. Three constraints are imposed on them: First of all, the value of b, when interpreted as an unsigned integer, should be smaller than 32 (the `#x00000020` is the hexadecimally encoded constant 32; the operator name `bvult` stands for "bitvector unsigned less than"). Likewise, the value of c as an unsigned integer should be greater than $5 \cdot 16 = 80$. Finally, it is mandated that when subtracting (`bvsub`) 2 from b, we obtain a value that is, when interpreted as an unsigned integer, greater than b.

If b and c were regular integers, then the SMT instance had no model: there is no way in which for some constants b and c, we could have

$$\begin{aligned} a &< 20, \\ b &> 50, \text{ and} \\ a - 2 &> b \end{aligned}$$

all holding at the same time. However, for the theories of bitvectors, a solution is found by `z3`:

```
sat
(model
  (define-fun b () (_ BitVec 32)
    #x00000001)
  (define-fun c () (_ BitVec 32)
    #x00000054)
)
```

Interpreted as integers, the values for b and c are 1 and 84. The subtraction of 2 from 1 actually leads to an *underflow* of a (for unsigned integers), which explains how the result of subtracting 2 from 1 can be larger than 84.

It should be noted that bitvector logic cannot be handled by the Nelson-Oppen procedure as the assumption that the theory is interpreted over an infinite data domain is not satisfied. Hence, practical SMT solvers use extensions of the Nelson-Oppen procedure to deal with this theory or encode it to SAT eagerly.

7.6.2 Difference logic

Integer and Real-valued difference logic are the little sisters of integer and real-valued linear arithmetic. Essentially, they simplify the latter by not allowing multiplication with constants other than 1. So comparisons such as $a + b + c \leqslant 42$ are allowed, whereas $a + 2 \cdot b + c \leqslant 23$ is not.

The advantage of having specialized difference logic procedures in an SMT solver is that they can be more efficient than the general ILP and LP procedures. Also, inferring reasons for the inconsistency of ground terms is normally simpler, which facilitates learning in the SMT solver's SAT procedure. For the case of integer variables, the problem of checking the consistency of a set of ground terms is also solvable in polynomial time (Kroening and Strichman, 2008, Chapter 5.7.1), whereas in the ILP case, it is NP-complete.

7.6.3 Non-linear arithmetic

So far, we only considered linear arithmetic over the integer and real numbers. Linear arithmetic is sufficient for many applications, and it is computationally simpler to handle than non-linear arithmetic. For example, the theory of integer numbers with the $\leqslant, \geqslant, =, -, +, \cdot$ operators and the 0 and 1 constants is undecidable. This result is commonly attributed to Church (1936), as for example done by Bozga and Iosif (2005). In the case of real-valued arithmetic, it is currently not exactly known where the borderline between decidability and undecidability lies regarding which operators can be supported, but the *sine function* is known to make showing the undecidability of a theory over the real numbers much easier.

For all practical means, it suffices to know that solving SMT formulas with non-linear arithmetic is very difficult, and the best we can hope for in an SMT solver is that we *sometimes* get results. The theory solver is always *sound* (modulo possible bugs in the solver), but also *incomplete*, meaning that the solver can sometimes output a "don't know" result or diverge. To see why this is the case, consider the following SMT instance:

```
(declare-const b Int)
(declare-const c Int)

(assert (= (* b c) 7044306370411))
(assert (> b 1))
(assert (> c 1))

(check-sat)
(get-model)
```

The SMT instance searches for a factoring of the number 7044306370411 such that both factors are > 1. The ability to factor large numbers efficiency would allow SMT solvers to be used to break many public-key cryptographic systems, so it is not surprising that this SMT instance is very difficult to solve even for a modern SMT solver such as `z3` (12m53.943s on an i5 1.6 GHz computer) for a satisfiable SMT instance (as the number 7044306370411 is the product of the two numbers 771853 and 9126487).

7.6.4 Arrays

Since SMT solvers are often used for software verification (as we will see in Chapter 7.7), they often have special support for the *theory of arrays*. Consider the following SMTLIB2 specification:

```
(declare-const a Real)
(declare-const b Real)
(declare-const c Int)
(declare-const d Int)
(declare-const m1 (Array Int Real))
(declare-const m2 (Array Int Real))
(declare-const m3 (Array Int Real))
(assert (= (store m1 c b) m2))
(assert (= (store m2 d a) m3))
(assert (= (select m3 c) a))
(assert (= (select m3 d) b))
(check-sat)
(get-model)
```

The variables m_1, m_2, and m_3 are arrays, which are essentially functions from some enumerable data type to an arbitrary data type. Here, the arrays are indexed by integers and store real values.

The `assert` statements express that m_2 is obtained from the array m_1 by storing b at position c into the array. Likewise, m_3 is the same as m_2, except that at position d, we stored a. Afterwards, it is checked that at position c in the array m_3, the value of a is found, and that position d in the array m_3, the value of b is found. In a nutshell, the assert statements express that the values a and b that have been stored into positions d and c in the array have swapped their positions.

The SMT formula is satisfiable and running `z3` 4.4.1 on it yields a model with $a = b = 1$. Adding `(assert (not (= a b)))` as an additional constraint leads to an unsatisfiable SMT instance: in arrays, elements stay where they are, and the indices c and d can also not be the same, as then

```
(assert (= (select m3 c) a))
(assert (= (select m3 d) b))
```

cannot hold at the same time when $a \neq b$.

Since we introduced arrays as function from an enumerable type to another type, it is fair to ask the question what the differences between the theory of arrays and uninterpreted functions are. In a nutshell, the values of array elements are guaranteed to stay the same unless changed. If we used uninterpreted functions to state that two functions f and f' are the same on all domain elements except for one, we would need to add constraints of the form $f(x) = f'(x)$ for all elements x that *could* be of relevance. The theory of arrays does this for us, which is especially handy if we do not know the set of values for x that are of relevance. This also leads to a higher complexity of the theory solver: in contrast to uninterpreted functions, where checking the consistency of a set of ground terms can be performed in polynomial time, the same question is NP-hard to solve for the theory of arrays (de Moura and Bjørner, 2009, see, e.g.,)

7.6.5 Floating point arithmetic

Another theory that is important for program verification is the theory of floating point arithmetic. It allows to reason about floating point computation in actual programs and the rounding errors that can occur in the process. Let us consider the following example:

```
(set-logic QF_FP)

(declare-const a (_ FloatingPoint 11 52))
(declare-const b (_ FloatingPoint 11 52))
(declare-const c (_ FloatingPoint 11 52))
(declare-const d (_ FloatingPoint 11 52))
(declare-const e (_ FloatingPoint 11 52))

(assert (= c (fp.add roundTowardZero a b)))
(assert (= d (fp.sub roundTowardZero c b)))
(assert (= e (fp.sub roundTowardZero d a)))
(assert (fp.geq e ((_ to_fp 11 52) roundTowardZero 23)))

(check-sat)
(get-model)
```

This example concerns five floating point numbers with 11 bits for the exponent and 52 bits for the significand. The data type concerned is the same as a 64-bit floating point number used in computers following the IEEE 754 standard for floating point computation.

The constraints state that c stems from adding a to b, d stems from subtracting b from c, and that e is obtained by subtracting a from d. The computation always rounds towards 0. Without rounding errors, we would have $c = a + b - b - a = 0$. The last assertion now states that e should be greater than or equal to the result to translating the integer number 23 into the floating point type.

Due to the rounding errors, the SMT instance is satisfiable and z3 returns the following model:

```
sat
(model
  (define-fun c () (_ FloatingPoint 11 52)
    (fp #b1 #b10001001101
        #b1111111111111111111111111111111111111111111111111111))
  (define-fun a () (_ FloatingPoint 11 52)
    (fp #b1 #b10001001110
        #b0000000000000000000000000000000000000000000000001000))
  (define-fun b () (_ FloatingPoint 11 52)
    (fp #b0 #b10000011110
        #b0000010000100011111111111111111111111111110101110 10))
  (define-fun d () (_ FloatingPoint 11 52)
    (fp #b1 #b10001001110
        #b0000000000000000000000000000000000000000000000000111))
  (define-fun e () (_ FloatingPoint 11 52)
    (fp #b0 #b10000011011
        #b0000000000000000000000000000000000000000000000000000))
)
```

Interpreted as human-readable numbers, a and b are approximately $7.09317 \cdot 10^{18}$ and $2.37343 \cdot 10^{18}$.

7.7 SMT applications

Let us consider a few example applications for SMT solving to see why SMT solvers are useful in practice.

7.7.1 Resource-constrained scheduling

Literature: *Ansótegui et al., 2011*

Ansótegui et al. (2011) use SMT solvers to solve the resource-constrained scheduling problem. In the problem, a set of resources and a set of tasks are given. Every task consumes fixed fractions of the resources during its execution, so that not all tasks can be executed at the same time. There is also a dependency relation between the tasks, which restricts some tasks to be executed after specific other tasks. All tasks also have fixed durations. The problem is now to find a schedule for the execution of the tasks such that all the ordering constraints are met and the resources are never exceeded by running too many tasks in parallel. At the same time, the tasks should be executed as quickly as possible.

The authors propose three different encodings:

- In the first encoding, integer constants encode the starting times of all tasks, using integer variables restricted to $[0, 1]$ for representing that a task is running at certain times.
- In the second encoding, for every task and a time frame of n, the authors use $n - 1$ variables for a ladder encoding of where a tasks starts.
- In the third encoding, for m many tasks, they introduce $m \cdot m$ many variables for encoding for every task after which other task it starts.

In the first two encodings, they optimize over the values of n.

The authors note that although the problem could also be encoded into ILP, the performance of the SMT solver is more robust. They applied the SMT solver `yices`, using linear arithmetic, and made good use of the `ite` operator of the solver's input language.

7.7.2 SMT for software analysis

The many theories geared towards software analysis already foreshadowed that this is one of the main applications of SMT solvers nowadays. Let us discuss this application by means of examples.

Consider the following function given in the programming language C over 32 bit integer variables:

```
int myfun(int b, int c) {
    int d = ((b & c) >> 16) & b;
    if ((d + 32)==c) {
        b = b ^ 123;
    }
    if ((b + 16)==d) {
        d = d ^ 256;
    }
    if ((b | d)>256) {
        b |= c & d;
    }
    int res = d | (c + b);
    return res;
}
```

The function `myfun` may be part of a bigger program and we want to make sure that it is tested thoroughly. Testing it thoroughly means in the context that we have at least one test case for every path through the function. For example, setting $b = 0$ and $c = 0$ leads to the none of the three `if` branches being executed. There are also other branch execution combinations possible, and we want to find one test case for every *feasible* such combination. Since the function is side-effect-free and the only parameters to the function are b and c, a test case is an assignment of b and c to integer constants such that the chosen branch combination is taken in the function. Due to the complexity of the operations performed in the function, it is not obvious which branch combinations are actually feasible, and we want to use an SMT solver to find this out for us.

We construct the SMT instance in two steps. First, we need to represent the program in a *single static assignment* (*SSA*) form. In this form, the value of a variable never changes once it is assigned. This is in contrast to imperative programming languages, where such changes are normally possible. We use the SSA form to bridge the SAT/LP/ILP/SMT-solving world, where the notion of a variable value changing over time does not even make sense, with the world of programming, where this concept does make sense.

For this example program, the SSA form could look as follows:

```
b := havoc;
c := havoc;
d1 := ((b & c) >> 16) & b;
b2 := ite((d1 + 32)==c,b ^ 123, b);
d3 := ite((b2 + 16)==d1,d1 ^ 256, d1);
b4 := ite((b2 | d3)>256,b2 | (c & d3),b2);
res := d3 | (c + b4);
```

The first two lines express that b and c can have arbitrary non-deterministic values. The lines afterwards assign values to d_1, b_2, d_3, and d_4, which express the evolution of the values of b and d during the execution of the function. Apart from the syntax, these lines are compatible with SMT solving. The first two lines are actually not needed as by default, the solver assumes to be able to assign arbitrary values to variables.

If we now want to check if the first and the third `if` conditions can fire at the same time, we add the following lines to the specification:

```
assert((d1 + 32)==c);
assert(not((b2 + 16)==d1));
assert((b | d)>256);
```

They encode that exactly the first and third `if` conditions hold. In order to now apply an SMT solver to check if we find such values for b and c, we need to encode the problem in SMTLIB2 format, using the theory of bitvectors to ensure that the semantics of C are captured faithfully. The final SMT instance looks as follows:

```
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))

(declare-const d1 (_ BitVec 32))
(declare-const b2 (_ BitVec 32))
(declare-const d3 (_ BitVec 32))
(declare-const b4 (_ BitVec 32))
```

```
(declare-const res (_ BitVec 32))

(assert (= d1 (bvand (bvashr (bvand b c) #x00000010) b) ))
(assert (= b2 (ite (= (bvadd d1 #x00000020) c) (bvxor b #x0000007B) b)))
(assert (= d3 (ite (= (bvadd b2 #x00000010) d1) (bvxor d1 #x00000100) d1)))
(assert (= b4 (ite (bvsgt (bvor b2 d3) #x00000100)
                  (bvxor b2 (bvand c d3)) b2)))
(assert (= res (bvor d3 (bvadd c b4))))

(assert (= (bvadd d1 #x00000020) c))
(assert (not (= (bvadd b2 #x00000010) d1)))
(assert (bvsgt (bvor b2 d3) #x00000100))

(check-sat)
(get-model)
```

The solver z3 finds a model quite quickly (< 1 second):

```
sat
(model
  (define-fun d1 () (_ BitVec 32)
    #x00000000)
  (define-fun b () (_ BitVec 32)
    #x7fffff8b)
  (define-fun c () (_ BitVec 32)
    #x00000020)
  (define-fun b2 () (_ BitVec 32)
    #x7ffffff0)
  (define-fun d3 () (_ BitVec 32)
    #x00000000)
  (define-fun res () (_ BitVec 32)
    #x80000010)
  (define-fun b4 () (_ BitVec 32)
    #x7ffffff0)
)
```

So the input to execute this combination is $(b, c) = (2147483531, 32)$. By iterating over all $2^3 = 8$ combinations of `if` branches and testing for all of them if suitable values for b and c exist, we can obtain test vectors for all combinations that can actually be taken. The test vectors can then be used for automated testing or given to the software engineer for inspection on whether they make sense and should actually be tested – in this way, SMT solving can be used to refine the preconditions for executing a piece of code given in its documentation.

A slight variation of the problem of finding suitable test vectors is the detection of superfluous sanity checks in programs. Consider, for example, the following modification of the problem from above:

```
int myfun(int b, int c) {
    int d = ((b & c) >> 16) & b;
    if ((d + 32)==c) {
        b = b ˆ 123;
    }
    if ((b + 16)==d) {
        d = d ˆ 256;
    }
    if ((b | d)>256) {
        b |= c & d;
    }
    int res = d | (c + b);
    assert ((res & 1)==0);
    return res;
}
```

The `assert` command may or may not be superfluous, as the code may always make sure that regardless of the values of *b* and *d*, the least significant bit of the *res* variable is 0. In order to improve performance of the code, we want to remove it if and only if it is actually superfluous.

Checking if the `assert` statement can ever fire can be done by SMT solving again using the following instance:

```
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))

(declare-const d1 (_ BitVec 32))
(declare-const b2 (_ BitVec 32))
(declare-const d3 (_ BitVec 32))
(declare-const b4 (_ BitVec 32))
(declare-const res (_ BitVec 32))

(assert (= d1 (bvand (bvashr (bvand b c) #x00000010) b) ))
(assert (= b2 (ite (= (bvadd d1 #x00000020) c) (bvxor b #x0000007B) b)))
(assert (= d3 (ite (= (bvadd b2 #x00000010) d1) (bvxor d1 #x00000100) d1)))
(assert (= b4 (ite (bvsgt (bvor b2 d3) #x00000100)
                   (bvxor b2 (bvand c d3)) b2)))
(assert (= res (bvor d3 (bvadd c b4))))

(assert (= #x00000001 (bvand res #x00000001)))

(check-sat)
(get-model)
```

The SMT instance is satisfiable, which implies that the `assert` statement should stay in the code in order to ensure that the function can only be called successfully for values of *b* and *c* for which it does not incorrectly set the least significant bit of the result to 1.

At this point, it should be stated that removing superfluous `assert`s and test vector generation are only two software quality assurance applications of SMT solving. With a surrounding framework, we can also use SMT solvers to prove the termination of programs (in certain cases), prove invariants for which expressing them in the form of `assert` statements would require additional storage of data, and proving the correctness of a program modulo the correct implementation of some API (application programming interface). The details of such applications however are beyond the scope of this book.

7.8 SMT extension: Quantifiers

Before we conclude with SMT solving, we want to also discuss an extension of it that is sometimes useful: Quantifiers. Some SMT solvers, such as `z3`, allow to introduce quantifiers into an SMT problems. For example, the following instance checks if there is an integer y such that for all integers x, we have that x is either greater than y, or x is smaller than 5.

```
(declare-const y Int)

(assert (forall ((x Int)) (or (< y x) (< x 5))))

(check-sat)
(get-model)
```

The solver concludes that there is no such value y – which makes sense. Introducing quantifiers fundamentally breaks the Nelson-Oppen procedure, so not all solvers support it. Often, such support is implemented by employing *quantifier elimination*, which transforms the problem into an equivalent quantifier-free one, which can be solved by SAT and the Nelson-Oppen procedure again. Quantifier elimination is a difficult problem, which limits the practical scalability of using quantifiers in SMT problems substantially.

7.9 Conclusion

SMT solvers allow to tackle a very diverse set of application by giving the user the possibility to combine constraints from a multitude of theories. While one of the main motivating factors for the development has been the area of software verifications, they have been increasingly scalable in last few years, such that they are now usable in a lot of other application areas as well.

The core idea of SMT solvers is to combine the reasoning efficiency of a SAT solver with specialized decision procedures for the individual theories. They are combined by the Nelson-Oppen procedure or a variant thereof. While the procedure enables the rich combination of theory solvers, it is also the limiting factor of SMT solvers' performance in addition to the computational difficulty of some theories themselves. The procedure only propagates equalities between the solvers. Depending on the problem to be solved, this can be quite inefficient. Nevertheless, SMT solvers have shown their usefulness in the field, so that they are an integral part of the practitioners' toolbox nowadays.

Chapter 8

Games of Infinite Duration

Most problem that we dealt with so far had no quantifiers or a fixed quantifier depth. There are problems, however, where we have to go beyond a fixed quantifier depth in order to solve them, and they also have practical applications. We want to deal with *games with infinite duration* in this chapter, which fall into this category. We start with a definition of the main concepts before discussing two applications. Afterwards, we give a glimpse of how games can be solved in practice.

8.1 Problem definition

A game of infinite duration is a tuple $\mathcal{G} = (V_0, V_1, E, \mathcal{F})$, where V_0 and V_1 are the two sets of *vertices* (also called *positions*) of the two players, $E \subseteq (V_0 \uplus V_1) \times (V_0 \uplus V_1)$ is the set of *edges* in the game, and $\mathcal{F}$ is a *winning condition*. The tuple may also contain an *initial position* v_{init}, but does not have to. The *transition structure* of games, i.e., the tuple $\mathcal{G}$ without the $\mathcal{F}$ element, can also be represented graphically as done in Figure 8.1. The vertices are drawn as nodes, and the edges are drawn as arrows between the nodes.

Games are a model for the interaction between two entities. Starting from some vertex v_{init} that may or may not be fixed, the two players gradually build a *play* $\pi = \pi_0\pi_1 \ldots$ such that $\pi_0 = v_{init}$ and for all i, we have $(\pi_i, \pi_{i+1}) \in E$. The play starts in π_0, and we say that it is player 0's turn at some position π_i whenever $\pi_i \in V_0$. Otherwise, it is player 1's turn.

A play is finite if at some point, it enters a position π_i that is a *dead end*, i.e., for which we have that $\{(\pi_i, v') \mid v' \in V_0 \uplus V_1\} \cap E = \emptyset$. All other plays are *infinite* in length and go on forever. A play is either winning for player 0 or for player 1, and it is losing for the respective other player. We say that player 0 loses the play if it is finite and ends in a position in V_0. The other finite plays (i.e., the ones that end in a position in V_1) are losing for player 1. For the infinite plays, $\mathcal{F}$ determines which player wins. We keep $\mathcal{F}$ abstract in the examples to come in order to avoid notational overload.

Since the two players decide the next successor positions whenever it is their respective turn, and a play can be winning for only one of them, they can play according to pre-set *strategies* in order to maximize the likelihood of winning the play. A strategy is essentially a function that maps all prefixes of plays to the next move by the player (i.e., which edge is taken by the player).

Example 35. Consider the game in Figure 8.1, where we assume that the play starts in vertex v_0. Player 0 is the one to which the rectangle-shaped states belong, and player 0 wins all infinite plays, so that it only needs to avoid that the play reaches her dead ends. Only v_3 is such a dead end.

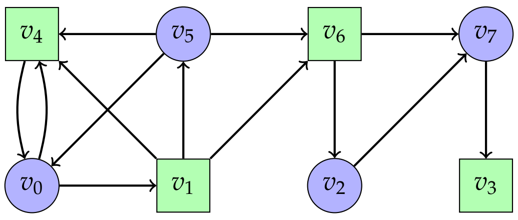


Figure 8.1: A first example for a game of infinite duration. The vertices of player 0 are given as rectangles, whereas the vertices of player 1 are given as circles.

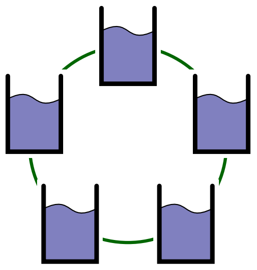


Figure 8.2: Bucket arrangement in the Cinderella-stepmother game

Analyzing the game, we can see that player 1 can win a play once it enters v_7, as it can just move the play into the dead end v_3. Likewise, in position v_2, player 1 can choose to take the edge to v_7, from which we know player 1 to be winning already.

But now, if we look at v_6, we see that player 0 has no way of avoiding to reach v_3 (in which she loses the play), regardless of whether player 0 chooses to take the transition to v_2 or v_7, she knows that the other player has a strategy to win from there. So avoiding to reach v_6 must also be one of the aims for player 0 to win the play.

In the same way, v_5 is also bad for player 0, as player 1 can choose to play to v_6 from there. However, from v_1 and v_4, there is a simple strategy for player 0 to avoid that v_5 or v_6 are ever visited: whenever the play is in v_1, player 0 should choose v_4 as the successor, and whenever the play is in v_4, it can choose v_0 as the successor. These choices make sure that the play always stays in $\{v_0, v_1, v_4\}$ and hence is winning for player 0. ★

It has been shown for the vast majority of possible winning conditions $\mathcal{F}$ that for every game with a finite number of vertices, one of the two players has a strategy that is surely winning. For some winning conditions, such as the one used in Example 35, the winning strategies for the two players are even *positional*, i.e., it suffices for the winning player to look at only the last position in a prefix play in order to determine the player's next move in the strategy.

Finding out for which starting vertices which player have a winning strategy is also called *solving the game*.

8.2 Examples

8.2.1 The Cinderella-stepmother game – abstract version

As a first example, we consider a game proposed as a benchmark for a different problem, namely *syntax-guided synthesis* (SyGuS) by Rajeev Alur (see, e.g., Beyene et al., 2014). While we are not concerned with SyGuS here, the game is still a good example for games of infinite duration, which is why we use it here.

Assume that there are five buckets standing in a circle, just like depicted in Figure 8.2. The buckets have a fixed capacity of c for some $c \in \mathbb{R}$. Cinderella and the stepmother play a game in which the stepmother tries to make one of the buckets overflow, while Cinderella tries to avoid this. They start with empty buckets and alternate turns – first, it is the stepmother's turn to fill in 1 liter of water into the buckets. She can distribute it among the buckets however she likes to. Afterwards, Cinderella empties two adjacent buckets.

Clearly, if $c < 1$ (we drop the unit "liter" in the following), then the stepmother wins, by just pouring the whole liter of water into one bucket. As the bucket has less than one liter of capacity, it overflows.

If however $c \geqslant 3$, then Cinderella clearly wins by executing the following emptying schedule infinitely often in succession:

Time step	**1**	**2**	**3**	**4**	**5**
Bucket 1 to empty	0	2	4	1	3
Bucket 2 to empty	1	3	0	2	4

With this schedule, in between two emptying step of a bucket, there are at most three time steps. Since the stepmother can only pour at most 1 liter of water in every step, this means that every bucket is always emptied before it overflows, no matter which moves the stepmother performs.

When considering these two cases together, it becomes clear that somewhere between $c = 1$ and $c = 3$ must be a borderline between the values for c for which Cinderella has a winning strategy, and the ones for which the stepmother has one. Since the stepmother and Cinderella can react to each other's moves, which is a property of the game that we did not use in the $c = 1$ and $c = 3$ cases, we should be able to refine the analysis.

For this, consider the case of $c < 1 + \frac{1}{5} + \frac{1}{3}$. Initially, the stepmother can distribute her liter of water evenly over the buckets. After the first move of the stepmother, it does not make a difference which buckets Cinderella empties: there are always 3 buckets remaining with $\frac{1}{5}$ liter of water each, and they are locate adjacently. The stepmother can now distribute her one liter of water over these three buckets, so that they contain $\frac{1}{5} + \frac{1}{3}$ liter of water each. Cinderella can empty two of them, but not all of them. So when it is then the stepmother's turn again, she can just fill the liter of water into (one of) the remaining bucket with $\frac{1}{5} + \frac{1}{3}$ liter of water in it, and if $c < 1 + \frac{1}{5} + \frac{1}{3}$, this means that an overflow occurs.

So we now know that the boundary is between $c = 1 + \frac{1}{5} + \frac{1}{3}$ and $c = 3$. For a fixed value of c, we can model the problem of checking if Cinderella has a winning strategy as a game solving problem. We define the game $\mathcal{G} = (V_0, V_1, E, v_{init}, \mathcal{F})$ as follows:

$$\begin{aligned}
V_0 &= [0, c + \epsilon]^5 \times \{0\} \\
V_0 &= [0, c + \epsilon]^5 \times \{1\} \\
E &= \{((r_0, r_1, r_2, r_3, r_4, 0), (r'_0, r'_1, r'_2, r'_3, r'_4, 1) \mid \\
&\qquad (r'_0 \geqslant r_0) \wedge (r'_1 \geqslant r_1) \wedge (r'_2 \geqslant r_2) \wedge (r'_3 \geqslant r_3) \wedge (r'_4 \geqslant r_4) \\
&\qquad \wedge (r'_0 + r'_1 + r'_2 + r'_3 + r'_4 - r_0 - r_1 - r_2 - r_3 - r_4 \leqslant 1)\} \\
&\quad \cup \{((r_0, r_1, r_2, r_3, r_4, 1), (r'_0, r'_1, r'_2, r'_3, r'_4, 0) \mid \max(r_0, r_1, r_2, r_3, r_4) \leqslant c \wedge (\\
&\qquad\quad (r'_0 = r_0 \wedge r'_1 = r_1 \vee r'_2 = r_2 \wedge r'_3 = 0 \wedge r'_4 = 0) \\
&\qquad \vee (r'_1 = r_1 \wedge r'_2 = r_2 \wedge r'_3 = r_3 \wedge r'_4 = 0 \wedge r'_0 = 0) \\
&\qquad \vee (r'_2 = r_2 \wedge r'_3 = r_3 \wedge r'_4 = r_4 \wedge r'_0 = 0 \wedge r'_1 = 0) \\
&\qquad \vee (r'_3 = r_3 \wedge r'_4 = r_4 \wedge r'_0 = r_0 \wedge r'_1 = 0 \wedge r'_2 = 0) \\
&\qquad \vee (r'_4 = r_4 \wedge r'_0 = r_0 \wedge r'_1 = r_1 \wedge r'_2 = 0 \wedge r'_3 = 0))\}
\end{aligned}$$

We used ϵ as a symbolic constant for an infinitesimally small value in the modelling for simplicity – the game can be expressed without it, but its definition is then more complicated. The winning condition $\mathcal{F}$ is the same as in Example 35. Here, player 0 (Cinderella) wins if it never reaches a deadlock. A position of a player always represents the bucket fill status.

Solving the game $\mathcal{G}$ in practice is tricky, as it has an infinite number of positions. This makes general computational engines that aim at game solving infeasible to use. We can however *discretize* the scenario by requiring the stepmother to always pour multiples of $\frac{1}{k}$ liter of water for some $k \in \mathbb{N}$ into the buckets. Then, we only have $c \cdot k + 1$ different filling status possibilities for every bucket plus one for the $c + \epsilon$ case, which makes the game finite. Figure 8.3 shows a graphical representation of a snippet of the resulting game.

Performing this discretization takes some power away from the stepmother, so whenever we find out that for some value of c, the stepmother loses, we do not know if this result is correct or not. However, whenever the stepmother wins under this restriction, then it surely wins in the real game.

When discretizing, we can however also give *more* power to the stepmother. For example, we could have that after the stepmother's move, all bucket contents are rounded up to the next multiple of $\frac{1}{k}$.

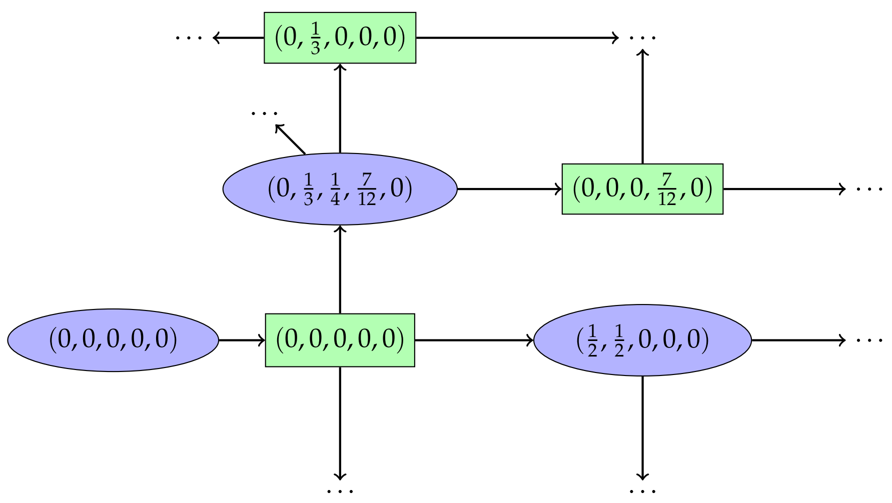

$(0,\frac{1}{3},0,0,0)$
$(0,\frac{1}{3},\frac{1}{4},\frac{7}{12},0)$
$(0,0,0,\frac{7}{12},0)$
$(0,0,0,0,0)$
$(0,0,0,0,0)$
$(\frac{1}{2},\frac{1}{2},0,0,0)$

Figure 8.3: A part of the Cinderella-stepmother discretized game for $c = 2$ and $k = 3$.

In this case, we still approximate the real result, but from the other side: if the stepmother loses the resulting game, then it surely loses the original game, but not the other way around.

If for some value for c, we want to find out if the stepmother or Cinderella wins, we can perform game solving for both approximation variants and increase the value of k until we find that either the game in which the stepmother's possibilities are weakened is found to be winning for her, or the game in which the stepmother's possibilities are strengthened is found to be losing for her (for some k). Only in these cases, we know that the result is precise and that the refinement process for k can be terminated.

8.2.2 The Cinderella-stepmother game – concrete version

We now want to apply a game solving tool to solve the Cinderella-stepmother game. As the tool requires the game to have a finite number of vertices, we can only solve an approximate version with it.

The tool that we will use is called `slugs`, and it is actually marketed as a *reactive synthesis* tool rather than a game solver. In reactive synthesis, we want to compute a piece of software or hardware that satisfies some given specification. There is no principal difference between such an implementation and a winning strategy in the Cinderella-stepmother game, which makes the tool applicable. The syntax of the tool's input language is however adapted towards the reactive synthesis world, and we take this into account during modelling.

In the `slugs` tool, there is a strong distinction between input and output. There are two players, of which one player chooses the valuation of the input variables, and the other player chooses the valuation of the output variables. Since the tool comes from the reactive synthesis domain, the setting is normally that we want to synthesize an implementation that generates the output. Since we are interested in a strategy for Cinderella, we associate her with the output, and let the stepmother produce the input.

`Slugs` has a relatively weak notion of state in the game. Rather than explicitly modelling the sets of vertices of the two players, the state needs to be completely contained in the input and output variable assignments. One can however add restrictions to the allowed moves of the players to force them to maintain important state information in their inputs and outputs. These restrictions are called *transition constraints* in the tool, and they essentially take the role of the edge relation E in the game definition.

Input files to `slugs` are simple text files. The tool supports bounded integer and boolean variables, so in order to model the scenario, we will assume that the stepmother can distribute k liters of water in every step, where she can only fill multiples of 1 liter into the buckets. The capacity variable is still c. We consider the case of $c = 17$ and $k = 9$.

In every round of the game in `slugs`, the input player can always move first, and then it is the output player's turn. This is compatible with our definition of the moves: we assume that the stepmother moves first, and only then it is Cinderella's turn.

Our `slugs` encoding starts with the following lines:

```
[INPUT]
x0:0...9
x1:0...9
x2:0...9
x3:0...9
x4:0...9

[OUTPUT]
y0:0...17
y1:0...17
y2:0...17
y3:0...17
y4:0...17
e:0...4
```

The variables x_0 to x_4 indicate how many units of water the stepmother wants to pour into every bucket (in a step). The output variables are used to indicate how many unit of water are in each bucket. The variable e represents the number of the first bucket for Cinderella to empty. The other bucket shall be the one right of the bucket number encoded in e.

Choosing y_0 to y_4 to be part of the output variables requires Cinderella to not only choose e wisely, but also assigns the job of keeping track of the bucket contents to Cinderella. This modeling choice is rooted in the fact that there is no strong notion of state in `slugs`, so that one of the players has to fulfil this task with her variables. The ranges behind the variable names in the `slugs` encoding indicate the minimal and maximal variable values. Since we have $k = 9$, there are 10 different choices for the values of every variable x_i. The variables y_i need a larger range as the buckets can hold more water than 9 units.

Choosing the initial state of the game is done in `slugs` by giving initialization constraints for the environment (setting the input variable values, i.e., the stepmother) and for the system (setting the output variable values, i.e., Cinderella). This is done with the following lines:

```
[ENV_INIT]
x0 = 0 & x1 = 0 & x2 = 0 & x3 = 0 & x4 = 0

[SYS_INIT]
y0 = 0 & y1 = 0 & y2 = 0 & y3 = 0 & y4 = 0
e = 0
```

Next, we restrict the allowed combinations of valuations to x_0 to x_4; they need to be $\leqslant c$ in order to be valid.

```
[ENV_TRANS]
x0'+x1'+x2'+x3'+x4' <= 9
```

The meaning of the dashes needs to be explained. The `ENV_TRANS` section in the input file describes the allowed input variable combinations, which are allowed to depend on the last input and output variable valuations. Variables with a dash at the end refer to the value after a transition, whereas the others represent the values before a transition. A transition in `slugs` consists of changes both in the input and the output variables. In a sense, a single transition in `slugs` refers to taking two edges successively in a game with strict alternation of the players.

Being able to reason about the variable values before and after a transition is useful in situations in which variable updates are performed. In this scenario, it is Cincerella's job to update the bucket contents. Overall, the system player transitions can be stated as follows:

```
[SYS_TRANS]
y0 + x0' <= 17
y1 + x1' <= 17
y2 + x2' <= 17
y3 + x3' <= 17
y4 + x4' <= 17
!((e<=0 & e+3 > 0) | (e+2 >= 5)) -> y0' = y0 + x0'
!((e<=1 & e+3 > 1) | (e+2 >= 6)) -> y1' = y1 + x1'
!((e<=2 & e+3 > 2) | (e+2 >= 7)) -> y2' = y2 + x2'
!((e<=3 & e+3 > 3) | (e+2 >= 8)) -> y3' = y3 + x3'
!((e<=4 & e+3 > 4) | (e+2 >= 9)) -> y4' = y4 + x4'
((e<=0 & e+3 > 0) | (e+2 >= 5)) -> y0' = 0
((e<=1 & e+3 > 1) | (e+2 >= 6)) -> y1' = 0
((e<=2 & e+3 > 2) | (e+2 >= 7)) -> y2' = 0
((e<=3 & e+3 > 3) | (e+2 >= 8)) -> y3' = 0
((e<=4 & e+3 > 4) | (e+2 >= 9)) -> y4' = 0
```

There are $3 \cdot 5$ constraint lines in total, all of which require discussion. The first five lines check that no overflow occurred along the transition *before* Cinderella gets the opportunity to empty two buckets. The lines may look a bit weird, as they do not actually talk about the output after the transition, while the SYS_TRANS section in the input file is supposed to state which next outputs are allowed. But this is no contradiction! All next output values are allowed that fulfill all constraints. If the bucket overflows right after the stepmother poured in water, then the requirement is violated, and hence no possible next output variable valuation exists. In the game built from the constraints, this means that in such a case, we have a deadlock for the system player, i.e., Cinderella, so she loses the game.

The bottom ten constraint lines ensure that if a bucket is one of the two buckets that Cinderella empties, then its next content is 0 units of water. Otherwise, its next content is obtained by adding the water that the stepmother pours into the bucket to its current value.

Running the `slugs` tool on this example yields the result that there exists a strategy for Cinderella to win the game. We can run an interactive simulator to play against Cinderella in order to convince ourselves that Cinderella indeed has a winning strategy. An example play from the simulator starts as follows:

Round	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	...
x_0	0	0	6	0	5	2	3	0	0	0	0	...
x_1	0	5	0	8	1	0	2	0	5	7	0	...
x_2	0	0	0	0	1	0	2	3	0	0	7	...
x_3	0	4	3	1	2	7	0	3	4	2	0	...
x_4	0	0	0	0	0	0	2	3	0	0	2	...
y_0	0	0	0	0	5	0	0	0	0	0	0	...
y_1	0	0	0	0	1	0	0	0	0	0	0	...
y_2	0	0	0	0	0	0	2	0	0	0	0	...
y_3	0	4	7	8	0	7	7	0	4	6	0	...
y_4	0	0	0	0	0	0	2	5	5	5	7	...
e	0	0	0	0	2	0	0	2	0	0	2	...

8.2.3 Maritime inventory routing problem (MIRP)

Let us now consider a more serious application for games of infinite duration. In the maritime inventory routing problem (MIRP), we are concerned with the problem of scheduling the trips of a fixed set of ships. There are source and target ports, each of which have an inventory and a production/consumption rate. There are known distances between the ports and the ships may have different travelling times for each source/target trip combination while also having different transport capacities.

The trip schedule should be selected such that at every port with a net production, the inventory never overflows (i.e., it exceeds its capacity) while at every port with a net consumption, the inventory never underflows (i.e., falls below 0).

The problem is quite similar to the circulation problem discussed in Section 6.7, only that this time, we have an inventory for every node in the graph, we have a fixed number of ships, the schedule for the ships is being searched for, and a ship cannot jump from one port to another one without a time penalty.

There is no a-priori bound on the length of the schedule, which makes encoding the problem into the computational engines discussed in the previous chapters difficult. The scenario can however easily be modelled as a game with only one player, which can direct the ships onto their routes. The vertices in the game encode the inventories of the ports, the filling grades of the ships and where each ship is. The game is a *safety game*, in which all infinite plays are declared to be winning for one of the players (which is only one here). Since we want the schedule to be correct if and only if no over- or underflow ever occurs, this winning condition is appropriate. After discretizing the game, it can then be solved with current game solving methods.

The classical formulation of the problem does not actually need a full game solver as there is only one player. However, the second player becomes useful when considering variants of the problem that take into account some uncertainties in the modelling. For example, the real production and consumption rates are not always constant, but can rather lie in some corridor of *min*/*max* values. As a fully loaded ship at a port may have to wait until there is enough inventory space to completely discharge its load, the occurrence of the minimal consumption rate may actually be worse than the maximal consumption rate in a sink port. Similarly, we can model that sometimes, cargo routes take an additional day, but we could assume that this only happens a fixed maximum number of times per scheduling period.

All of these extensions make the problem more realistic, but also more difficult to solve. Scientific research on the topic mostly considers a very weak version of the problem, which only encompasses the finite-horizon case of the single-player version (see, e.g., Papageorgiou et al., 2014).

8.3 Game Solving

In the following, we assume that the games considered have a finite number of vertices, such as the one in Figure 8.1. Let us first consider a game with a simple type of winning condition, namely *safety games*, in which every play of infinite length is won by the *safety player* (which can be player 0 or 1).

8.3.1 Safety Games

In Example 35, we derived a partitioning of the positions of the game in Figure 8.1 into the ones that are winning for player 0 and the ones that are winning for player 1. The rules of the games allowed us to start with assigning a few positions to the partitioning elements:

- If a position for the safety player does not have successor positions, then it is winning for the reachability player.
- If a position for the reachability player does not have successor positions, then it is winning for the safety player.

We then used the following arguments to identify positions that are winning for the non-safety player (who is also called the *reachability player*):

- If a position v of the reachability player has any successor position that is winning for the reachability player, then so is v.

- If all successor positions of a position v of the safety player are winning for the reachability player, then so is v.

In the first of these cases, the reachability player can win from position v as she can simply take the transition that is already known to be winning for her. By following the winning strategy from the successor position, she can then also win a game starting in v. In the second of the cases, the reachability player wins as she knows that regardless of which transition the safety player takes, the play always enters a position from which she has a winning strategy. So she definitely wins the play.

Note that applying the above four rules until no new positions are found to be winning for either of the players is *sound*, which can easily be shown by induction. Even more, if a position v is winning for the reachability player, then there exists a bound b on the number of steps in the play until a deadlock position for the safety player is reached.

Applying these rules is however not necessarily *complete*, as when stopping using the above rules, there may be positions that have neither been found to be winning for the reachability player, nor known to be winning for the safety player at that stage.

There is however a simple way to complete the rules to a game solving procedure: if we apply the rules from above to obtain a set of positions W that are winning for the reachability player, and no rule from above can be applied to identify any more positions as being winning for the reachability player, we can claim that $(V_0 \uplus V_1) \backslash W$ is winning for the safety player.

Now why is that? Let a play be in a position $(V_0 \uplus V_1) \backslash W$ and player 1 be the safety player. If the position is in V_1, then it is the safety player's move to select a transition. There is at least one transition available to a state in $(V_0 \uplus V_1) \backslash W$. Taking this transition means that the play continues. We know that v cannot be a dead-end because otherwise it would be in W. Now let us assume that $v \in V_0$. Since the second non-dead-end-rule from above does not apply, we know that all successor positions must be in $(V_0 \uplus V_1) \backslash W$, or v is a reachability-player dead-end. By induction over the length of the play, we find out that either the play goes on forever (so the safety player wins), or the play eventually reaches a dead end for the reachability player, so that the safety player wins again. Thus, the strategy that we sketched in this paragraph is winning for the safety player.

So in order to find a strategy for the safety player, it suffices to apply the rules from above until no more positions are found to be losing for the safety player, and then construct a strategy to always stay in the set of positions that is winning for the safety player.

There is a concise and illustrative style of notation to represent game solving algorithms such as the one that we sketched with the rules above. Let W be a set of positions that are already found to be winning for the reachability player (which we assume to be player 0). We can state that we extend W to a position set W' by computing

$$W' = W \cup (V_0 \cap \Diamond W) \cup (V_1 \cap \Box W) \tag{8.1}$$

for the following definitions of $\Diamond$ and $\Box$ (for all state sets $U \subseteq V_0 \uplus V_1$):

$$\begin{aligned} \Diamond U &= \{v \in V_0 \uplus V_1 \mid \exists v' \in U.(v, v') \in E\} \\ \Box U &= \{v \in V_0 \uplus V_1 \mid \forall v' \in (V_0 \cup V_1).(v, v') \in E \rightarrow v' \in U\} \end{aligned}$$

The $\Diamond$ and $\Box$ operators correspond exactly to the two types of checks proposed above. Computing W' from W means applying the above rules to every position in the game at the same time. Note that dead ends of player 1, which are winning for the player 0, are found by the $(V_1 \cap \Box W)$ part of the equation. This is simply because all successors of dead ends are in W. Applying the equation to update a set W twice can yield an even larger position set, as positions may only be found to be winning for the reachability player after their successor positions have been found to be winning. Thus, the update step needs to be applied until no more positions are found in order to find all of them, starting from a set of positions that are surely winning for the reachability player. Since all dead ends of the safety player are found by Equation 8.1, we can simply start with the empty set of positions.

Fixpoint notation gives us the possibility to combine the complete computation procedure into a single line:

$$W = \mu X.X \cup (V_0 \cap \Diamond X) \cup (V_1 \cap \Box X) \tag{8.2}$$

The μ operator takes a function f from $2^{\mathcal{V}}$ to $2^{\mathcal{V}}$ for some set $\mathcal{V}$ and computes the *least fixpoint* of f, i.e., the smallest set $U \subseteq \mathcal{V}$ for which $f(U) = U$. For *monotone functions*, i.e., functions for which for all sets $A \subseteq B$, we have $f(A) \subseteq f(B)$, this set is uniquely determined, which follows from the famous Knaster-Tarski Theorem (and is quite trivial to show anyway for *finite* sets $\mathcal{V}$). We can determine the least fixpoint by computing the series $\mu^i X.f(X)$, where

$$\mu^0 X.f(X) = \emptyset \text{ and}$$
$$\mu^{i+1} X.f(X) = f(\mu^i X.f(X)) \text{ for all } i \geqslant 0.$$

We can also consider the computation of $W' = W \cup (V_0 \cap \Diamond W) \cup (V_1 \cap \Box W)$ for some set W as a function for obtaining W' from W, and then the set characterized by Equation 8.2 is exactly the result of applying the above rules until there are no opportunities for applying them any more. Hence, the evaluation of Equation 8.2 over a game structure is exactly what is needed in order to solve safety games.

8.3.2 Symbolic game solving

If a game is stored in a position-by-position fashion in memory, evaluating Equation 8.2 over the game structure is relatively easy. The games however quite often have a number of positions that is too high to allow storing the game in memory. A common approach to solve this issue is to represent E and the intermediate positions sets during the evaluation of Equation 8.2 as *binary decision diagrams* (*BDD*s).

A very short primer on BDDs Intuitively, binary decision diagrams are compact representations of *boolean functions*, i.e., functions that map some assignment of a set of variables $\mathcal{V}$ to $\{\mathbf{false}, \mathbf{true}\}$ to either **false** or **true**. While not all functions have compact representations as BDDs, it has been observed that the boolean functions that appear in the game solving process presented in this section mostly do, which gives rise to BDD-based game solving as a practical approach. We do not want to discuss how exactly BDDs operate, but treat them on an abstract level in the following.

Let $\mathcal{B}(\mathcal{V}) = \mathcal{V} \to \mathbb{B}$ denote the set of assignments of $\mathcal{V}$ to $\mathbb{B}$. A BDD f over some set of variables $\mathcal{V}$ is a map from $\mathcal{B}(\mathcal{V})$ to $\mathbb{B}$. Given BDDs A and B, we can apply the following operations:

$$A \wedge B := \{\vec{v} \mapsto A(\vec{v}) \wedge B(\vec{v}) \mid \vec{v} \in \mathcal{B}(\mathcal{V})\}$$
$$A \vee B := \{\vec{v} \mapsto A(\vec{v}) \vee B(\vec{v}) \mid \vec{v} \in \mathcal{B}(\mathcal{V})\}$$
$$\neg A := \{\vec{v} \mapsto \neg A(\vec{v}) \mid \vec{v} \in \mathcal{B}(\mathcal{V})\}$$

In addition, given a set $V \subseteq \mathcal{V}$ of variables and two equally long lists $M = m_0, \ldots, m_n$ and $M' = m'_0, \ldots, m'_n$ of variables, we define:

$$\exists V.A = \{\vec{v} \mapsto b \mid \vec{v} \in \mathcal{B}(\mathcal{V} \backslash V), b \in \mathbb{B}.b \leftrightarrow (\exists \vec{u} \in \mathcal{B}(V).A(\vec{v} \cup \vec{u}) = \mathbf{true})\}$$
$$\forall V.A = \{\vec{v} \mapsto b \mid \vec{v} \in \mathcal{B}(\mathcal{V} \backslash V), b \in \mathbb{B}.b \leftrightarrow (\forall \vec{u} \in \mathcal{B}(V).A(\vec{v} \cup \vec{u}) = \mathbf{true})\}$$
$$A[M'/M] = \{\vec{v} \mapsto A(\vec{v}') \mid \vec{v}, \vec{u} \in \mathcal{B}(\mathcal{V}).\forall i \in \{0, \ldots, n\}.\vec{u}(m_i) = \vec{v}(m'_i),$$
$$\forall x \notin \{m_0, \ldots, m_n\}.\vec{u}(x) = \vec{v}(x)\}$$

Every BDD is defined over a set of set of variables, and for a BDD A over the variables $\mathcal{V}$, computing $\exists V.A$ for some set $V \subseteq \mathcal{V}$ results in a BDD over a smaller set of variables. However, every BDD over a set of variables $\mathcal{V}$ can be seen as a BDD for a larger set of variables $\mathcal{V}' \supseteq \mathcal{V}$. In this way, computing $A \wedge \exists V.A$ is allowed for some BDD A over a set of variables $\mathcal{V}$ even though $\exists V.A$ is a BDD over a smaller set of variables (if V is non-empty). By the definitions of the operations above, we have that $(A \wedge \exists V.A) = A$.

BDDs for game solving A BDD can be used to encode a set of positions in a game and the transition relation of a game, so that Equation 8.2 can be evaluated symbolically. Given position sets V_0 and V_1, we need a set of variables $\mathcal{V}$ that is large enough such that every position in $V_0 \uplus V_1$ can be represented as a different assignment to $\mathcal{V}$. A BDD B over $\mathcal{V}$ then represents the set of positions p for which for their encoding $\vec{p}$, we have $B(\vec{p}) = \mathbf{true}$. We need at least $\lceil \log_2(|V_0| + |V_1|) \rceil$ variables for this. For the simplicity of presentation, we assume that $|V_0| + |V_1|$ is exactly $2^{|\mathcal{V}|}$ in the following.

For every position p, let $f(p)$ be the encoding of p into $\mathcal{V}$. For many games, there exist relatively straightforward encodings f such that the BDDs A_0 and A_1 that map $f(p)$ to **true** for exactly the positions in V_0 and V_1, respectively, can be stored in a memory-efficient way. In order to encode the edge relation E in a game, which can have up to $(|V_0| + |V_1|)^2$ different elements, the set of variables $\mathcal{V}$ for encoding the positions is insufficient. We solve this difficulty by introducing an additional variable set $\mathcal{V}'$ so that the BDD Y that encodes the edge set E of the game can range over $\mathcal{V} \cup \mathcal{V}'$, which is sufficient for encoding E. To do so, we must set an order $M = m_0 \ldots m_n$ for the variables $\mathcal{V}$ and a corresponding order $M' = m'_0 \ldots m'_n$ for the variables $\mathcal{V}'$. An edge from some positions p to p' then exists in Y if and only if $Y(f(p) \cup f(p')[m'_0/m_0, \ldots, m'_n/m_n]) = \textbf{true}$.

With the BDDs Y, A_0, and A_1 prepared, we can now reformulate Equation 8.2 for obtaining the set of winning positions for the reachability player as an fixpoint equation over BDD operations:

$$W = \mu X.\, X \vee \underbrace{\underbrace{(A_1 \wedge \exists \mathcal{V}'.(Y \wedge X[M'/M]))}_{\text{Reachability player positions}} \vee \underbrace{(A_0 \wedge \forall \mathcal{V}'.(\neg Y \vee X[M'/M]))}_{\text{Safety player positions}}}_{\psi} \tag{8.3}$$

Again, the equation is relatively short. Note that ψ is actually a function from BDDs over $\mathcal{V}$ to BDDs over $\mathcal{V}$, and it is monotone, so that a least fixpoint over it is well-defined. Evaluating this formula by repetitive evaluation of ψ amounts to successively computing the set of positions for which either (1) the position is a reachability player position and it has a successor that is winning for the player, or (2) the position belongs to the safety player and all successors are winning for the reachability player. While the usage of the $\forall$ operator in the formula looks uncommon at first, it is rooted in the fact that by default, the operator considers all valuations of the variables quantified out. Taking $\neg Y \vee X[M'/M])$ ensures that only successor positions that are actually reachable by E are required to be contained in X.

By defining

$$\mathit{EnfPre}(X) = (A_1 \wedge \exists \mathcal{V}'.(Y \wedge X[M'/M])) \vee (A_0 \wedge \forall \mathcal{V}'.(\neg Y \vee X[M'/M])),$$

we can simplify Equation 8.3 to

$$W = \mu X.X \vee \mathit{EnfPre}(X)$$

The *enforceable predecessor* operator is probably the most important individual operation in game solving.

8.3.3 Büchi games

In addition to the safety games defined earlier in this chapter, there also exist more sophisticated winning conditions for infinite plays. An example is the *Büchi winning condition*, where one of the player tries to visit a set of *goal positions* infinitely often. Games with this winning condition are called *Büchi games* and have applications in *reactive synthesis*, which we discuss in the next section.

Let player 1 be the Büchi player in the following. If player 1 can make sure that either eventually a dead-end for the non-Büchi player is reached, or infinitely often goal states are visited, then it can surely enforce that at least n times (for some $n \in \mathbb{N}$), goal states are visited unless a dead end is reached beforehand.

For $n = 1$, this is exactly a reachability game, we can compute:

$$W_1 = \mu X.X \vee \mathit{EnfPre}(X \vee G)$$

In this equation, G is the set of goal states. We only count when the goal state is reached after the first move in the game. This choice is motivated by the idea that we can then characterize W_2 as follows:

$$W_2 = \mu X.X \vee \mathit{EnfPre}(X \vee (G \wedge W_1))$$

The idea is that in order to visit goal states at least two times, the Büchi player has to be able to reach such a state a single time, and from there it must be able to reach a goal state one more time. The argument can be generalized for all $n > 1$ as follows:

$$W_n = \mu X.X \vee EnfPre(X \vee (G \wedge W_{n-1}))$$

It follows directly from the definition that for every $n > 1$, we have $W_{n-1} \supseteq W_n$. It can be shown that if $W_n = W_{n-1}$, the position set W_n is exactly the one from which the Büchi player can win the game.

Again, iterating through a process until no more change can be observed can be modelled as a fixpoint equation. This time, it is a *greatest fixpoint*, denoted by the operator ν. It takes a monotone function f mapping from 2^S to 2^S for some set S, and applies f to 2^S until no more change occurs.

The overall game solving process can thus be summarized as:

$$W = \nu Y.\mu X.X \vee EnfPre(X \vee (G \wedge Y))$$

8.4 Application: Reactive synthesis

One of the prime applications of game solving is *reactive synthesis*. In reactive synthesis, we want to automatically compute a controller that in every clock cycle reads the values of some input signals, produces suitable output signals, and updates its internal state. The controller's computation does not terminate, i.e., it continues operation until it is eventually switched off. Computations of such controllers are assumed to be infinite as there is no predefined time of going out of service. We want the controller to only produce correct implementations, i.e., those that lie in a given *language* over infinite words that represent the computations of the controller.

The difficulty of the problem is that the controller has no control over its inputs and also does not know the input signal valuations ahead of time. Thus, it is has to make strategically wise decisions that allow it to satisfy the specification regardless of how the input to the controller continues.

Reactive synthesis is typically reduced to game solving: if we give one of the players the task to select the output variable values and the other player can select the input variable values, then by encoding the specification as a winning condition in the game, we have a 1-to-1 mapping between winning strategies for the system player in the game and correct controllers.

Let us consider an example for reactive synthesis that shows that the Büchi winning condition is useful. In fact, the following example uses a *generalized Büchi winning condition*, where the Büchi player (which is the environment player here) has to visit multiple goal sets infinitely often in order to win on an infinite play. The `slugs` tool that we already used in Chapter 8.2.2 supports such winning conditions, so that we can use it again for the following example.

We want to synthesize a controller for an elevator. The elevator has to service all four floors of a building by request. We store into the game positions what is the current floor the elevator is at, whether the elevator door is open, and which requests are outstanding. The environment (non-Büchi) player gives requests for floors, and the system player chooses the motion of the elevator between the floors. The elevator can only move with closed doors, and serving a floor only counts if the door has opened.

To model the scenario in `slugs`, we start by defining the variable sets of the two players. As in Section 8.2.2, we assign the task to keep track of the state of the environment to the system player, who is also the generalized Büchi player in `slugs`.

```
[INPUT]
req0
req1
req2
req3

[OUTPUT]
```

```
doorOpen
floor:0...3
outReq0
outReq1
outReq2
outReq3
```

The Variables `req0` to `req3` model requests by the users of the elevator, whereas `outReq0` to `outReq3` models whether there are outstanding requests for floors 0 to 3. We next model the initial values of these variables:

```
[ENV_INIT]
!req0 & !req1 & !req2 & !req3

[SYS_INIT]
floor=0
!doorOpen
!outReq0 & !outReq1 & !outReq2 & !outReq3
```

The environment player is completely free in when requests are issued, so there is no `[ENV_TRANS]` part necessary in the specification. We next model that the elevator can only move with the door closed.

```
[SYS_TRANS]
floor!=floor' -> (!doorOpen & !doorOpen')
```

Now, we have to ensure that the variables `outReq0` to `outReq3` are set by requests, and reset to **false** whenever a floor has been reached and the door is open. Just as most real elevators, we let it ignore requests to floors at which the elevator is currently located with open doors.

```
outReq0' <-> (!(floor'=0 & doorOpen) & (outReq0 | req0'))
outReq1' <-> (!(floor'=1 & doorOpen) & (outReq1 | req1'))
outReq2' <-> (!(floor'=2 & doorOpen) & (outReq2 | req2'))
outReq3' <-> (!(floor'=3 & doorOpen) & (outReq3 | req3'))
```

Also of relevance is the observation that the elevator can only move one floor at a time:

```
floor'+1 >= floor
floor+1 >= floor'
```

These two constaints together ensure that $|\texttt{floor} - \texttt{floor}'| \leqslant 1$. Finally, we must state that there is never an outstanding request that stays outstanding forever. This amounts to saying that every outstanding request variable must always eventually have a **false** value. We can state this as follows in a `slugs` specification:

```
[SYS_LIVENESS]
! outReq0
! outReq1
! outReq2
! outReq3
```

This completes the specification, and by using the specification in the `slugs` simulator for realizable specifications (i.e., those in which the system player has a strategy to win the game), we can convince ourselves that the elevator controller works as expected.

The `slugs` tool can also compute a circuit that implements the controller if desired, so that it can be used in actual hardware.

8.5 Conclusion

In this section, we discussed games of infinite duration as a modelling framework for difficult computational problems in which we want to find solutions to problems without a fixed quantifier alternation depth. We have seen two applications (robust logistical planning and reactive synthesis) that can make use of the generality of the the model. We have also glimpsed at how solving games actually works, namely by fixpoint iteration on position sets represented using symbolic data structures such as BDDs.

There are many extensions of the basic game types (safety, Büchi) that we considered. In some of them, BDDs are no longer applicable, which leads to severe computational restrictions. But even with BDDs, due to the higher difficulty of the game solving problem compared to SAT or QBF solving, scalability has often been observed to be a problem for practical game solving in the reactive synthesis application (see, e.g., Jacobs et al., 2016). As an example, the `slugs` tools has been observed to hardly scale above 50 boolean input and output variables for specification files of medium complexity, while other game solver/reactive synthesis tools for more sophisticated winning conditions max out even earlier. Yet, game solving is slowly becoming an integral part of today's computational toolset as it allows to deal with problems that are not easily encoded into a fixed quantifier structure.

Chapter 9

Conclusion

The aim of this book is to give the reader a broad overview about what was called "computational engines" throughout the book. It should be noted that this is not a common term in literature. Rather, the term was introduced to characterize algorithms that solve computational problems that are broadly enough defined to allow encoding many practical problems into them while at the same time being narrowly enough defined to allow efficient specialized solution algorithms for them.

Computational engines help with gluing the theory and practice of algorithm engineering together. They allow to solve many problem from the practice of computer science without constructing specialized solution algorithms. It suffices to find a suitable encoding that makes use of the engine's properties. For this reason, we not only discussed the applications, but also developed some insight into how the engines are working that helps with modelling practical problems in a way that makes best use of the engine's properties.

A good overview about a variety of computational engines is necessary in order to apply them effectively and efficiently in practice. Most of the engines discussed in this book have been developed by different scientific communities, and knowledge about them dissipates only quite slowly into other communities. It is an aim of this book to equip the reader with a good enough overview of computational engines to identify situations in the practice or research of computer science in which it makes sense to apply them. Also, our treatment of the engines shows that one does not need to be afraid of NP-hard or even harder problems in practice: as long as there are relatively direct encodings into an efficient computational engine, it makes sense to try solving a hard practical problem anyway, as algorithmic advances in the computational engines have led to a surprising level of scalability.

The computational engines discussed in this book have been quite diverse. We started with SAT solvers (that tackle an NP-complete problem) and then worked our way to QBF solving (which is PSPACE-complete). In LP solving, exponential worst-case algorithms are often applied even though the problem is in PTIME. The network flow problem was also in PTIME and was mainly discussed due to its numerous practical applications. The SMT solving problem is so heterogeneous that no concrete complexity class can be given, and while the game solving problems discussed in the previous chapter (Büchi games and safety games) can be solved in polynomial time, due to the compactness of BDD representations that are used for large games, the problem raises to EXPTIME-completeness when using them).

There are also many computational engines that we left out. Apart from additional variants to the linear programming problem (such as quadratic programming and semidefinite programming), the SMT solving problem has many more facets that we had to omit to foster conciseness. The overview given in this book should however be a good starting point for more focussed additional reading on such extensions of the computational problems considered.

Bibliography

Dimitris Achlioptas. *Random Satisfiability*, chapter 8, pages 245–270. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona, Catalonia, Spain, July 17-18, 2011.*, 2011.

D. Avis and V. Chvátal. Notes on Bland's pivoting rule. In M.L. Balinski and A.J. Hoffman, editors, *Polyhedral Combinatorics*, volume 8 of *Mathematical Programming Studies*, pages 24–34. Springer Berlin Heidelberg, 1978. ISBN 978-3-642-00789-7. doi: 10.1007/BFb0121192. URL `http://dx.doi.org/10.1007/BFb0121192`.

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011. doi: 10.1007/978-3-642-22110-1_14.

Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 221–234, 2014. doi: 10.1145/2535838.2535860.

Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008. URL `http://jsat.ewi.tudelft.nl/content/volume4/JSAT4_5_Biere.pdf`.

Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. ISBN 978-1-58603-929-5.

Magnus Bjork. Successful SAT encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation Addendum Paper*, 2009.

Marius Bozga and Radu Iosif. On decidability within the arithmetic of addition and divisibility. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 425–439, 2005. doi: 10.1007/978-3-540-31982-5_27. URL `http://dx.doi.org/10.1007/978-3-540-31982-5_27`.

Martin Brain, Liana Hadarean, Daniel Kroening, and Ruben Martins. Automatic generation of propagation complete SAT encodings. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 536–556, 2016. doi: 10.1007/978-3-662-49122-5_26.

Daniel Bundala and Jakub Zavodny. Optimal sorting networks. In Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2014. ISBN 978-3-319-04920-5. doi: 10.1007/978-3-319-04921-2_19. URL `http://dx.doi.org/10.1007/978-3-319-04921-2_19`.

Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. ISSN 00029327. doi: 10.2307/2371045. URL `http://dx.doi.org/10.2307/2371045`.

Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In Patrice Godefroid, editor, *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005. ISBN 3-540-28195-9. doi: 10.1007/11537328_9. URL `http://dx.doi.org/10.1007/11537328_9`.

Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047.

G. B. Dantzig. Maximization of linear function of variables subject to linear inequalities. In T. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, page 339–347. John Wiley, 1951.

Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi: 10.1145/368273.368557. URL `http://doi.acm.org/10.1145/368273.368557`.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 45–52, 2009. doi: 10.1109/FMCAD.2009.5351142.

Jian Ding, Allan Sly, and Nike Sun. Proof of the satisfiability conjecture for large k. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 59–68, 2015. doi: 10.1145/2746539.2746619. URL `http://doi.acm.org/10.1145/2746539.2746619`.

E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972. doi: 10.1145/321694.321699. URL `http://doi.acm.org/10.1145/321694.321699`.

Niklas Eén and Niklas Sörensson. Minisat v.2.2.0, 2010. Available at `http://minisat.se/MiniSat.html`.

Rüdiger Ehlers and Bernd Finkbeiner. On the virtue of patience: Minimizing Büchi automata. In *SPIN*, pages 129–145, 2010.

John Franco and John Martin. *A History of Satisfiability*, chapter 1, pages 3–74. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *J. Autom. Reasoning*, 35(1-3):143–179, 2005. doi: 10.1007/s10817-005-9011-0.

M. Gebser, B. Kaufmann, and T. Schaub. `Clasp` answer-set programming solver, 2012a. Available at `http://www.cs.uni-potsdam.de/clasp/`, retrieved on the 17*th* May 2016; The techniques used in CLASP are given in the paper by Gebser et al. (2012b).

M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012b.

Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. *Reasoning with Quantified Boolean Formulas*, chapter 24, pages 761–780. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. doi: 10.1145/505145.505149. URL `http://doi.acm.org/10.1145/505145.505149`.

Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition (syntcomp 2014). *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2016. ISSN 1433-2787. doi: 10.1007/s10009-016-0416-3.

Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012. ISBN 978-3-642-31611-1. doi: 10.1007/978-3-642-31612-8_10. URL `http://dx.doi.org/10.1007/978-3-642-31612-8_10`.

Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4): 373–396, 1984. doi: 10.1007/BF02579150. URL `http://dx.doi.org/10.1007/BF02579150`.

Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1368–1373. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X. URL `http://www.aaai.org/Library/AAAI/2005/aaai05-217.php`.

V. Klee and G. J. Minty. How Good is the Simplex Algorithm? In Oved Shisha, editor, *Inequalities III*, pages 159–175. Academic Press Inc., 1972.

Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education Inc., 2006.

Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching, Second edition*. Addison-Wesley, 1998.

Donald E. Knuth. *The Art of Computer Programming, Volume 4, Pre-Fascicle 6a: A (partial draft) of Section 7.2.2.2: Satisfiability*. Addison-Wesley, 2015.

Donald E. Knuth, Jon Bentley, Dave Walden, Charles Leiserson, Dennis Shasha, Mark Taub, Radia Perlman, Tony Gaddis, Robert Sedgewick, Barbara Steele, Silvio Levy, Peter Gordon, Udi Manber, Al Aho, Guy Steele, Robert Tarjan, Frank Ruskey, Andrew Binstock, Jeffrey O. Shallit, Scott Aaronson, and J. H. Quick. Twenty questions for Donald Knuth. *InformIT*, May 2014. Available online at `http://www.informit.com/articles/article.aspx?p=2213858`, retrived on April 7, 2016.

Daniel Kroening and Ofer Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer, 2008.

Krzysztof Kuchcinski and Radoslaw Szymanek. Java constraint solver (JaCoP), 2010. URL `http://www.jacop.eu`.

Leonid A. Levin. Universal sorting problems. *Problems on Information Transmission*, 9:265–266, 1973.

Florian Lonsing. DepQBF 5.0, 2015. Available at `http://lonsing.github.io/depqbf/`, retrieved on the 17*th* May 2016; Some advanced techniques used in the solver have been described by Lonsing et al. (2015).

Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing search-based QBF solving by dynamic blocked clause elimination. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 418–433, 2015. doi: 10.1007/978-3-662-48899-7_29.

Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Kim Marriott and Peter J. Stuckey. A minizinc tutorial. Technical report, NICTA, 2014.

Peter Marwedel. *Embedded System Design*. Kluwer, 2003. ISBN 978-1-4020-7690-9.

David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 459–465, 1992. URL `http://www.aaai.org/Library/AAAI/1992/aaai92-071.php`.

Dimitri J. Papageorgiou, George L. Nemhauser, Joel S. Sokol, Myun-Seok Cheon, and Ahmet B. Keha. Mirplib - A library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014. doi: 10.1016/j.ejor.2013.12.013. URL `http://dx.doi.org/10.1016/j.ejor.2013.12.013`.

Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing, 2015. doi: 10.1007/978-3-319-21810-6.

Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi: 10.1016/j.artint.2010.10.002. URL `http://dx.doi.org/10.1016/j.artint.2010.10.002`.

Steven Prestwich. *CNF Encodings*, chapter 2, pages 75–97. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Jussi Rintanen. *Planning and SAT*, chapter 15, pages 483–504. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

C. Roos. An exponential example for Terlaky's pivoting rule for the criss-cross simplex method. *Mathematical Programming*, 46(1-3):79–84, 1990. ISSN 0025-5610. doi: 10.1007/BF01585729. URL `http://dx.doi.org/10.1007/BF01585729`.

Karem A. Sakallah. *Symmetry and Satisfiability*, chapter 10, pages 289–338. Volume 185 of Biere et al. (2009), February 2009. ISBN 978-1-58603-929-5.

Ron Shamir. Probabilistic analysis in linear programming. *Statistical Science*, 8(1):57–64, 1993.

Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978. doi: 10.1145/359545.359570.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415, 2006. doi: 10.1145/1168857.1168907.

Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 244–257, 2009. doi: 10.1007/978-3-642-02777-2_24.

Tamás Terlaky and Shuzhong Zhang. Pivot rules for linear programming: A survey on recent theoretical developments. *Annals of Operations Research*, 46-47(1):203–233, 1993. ISSN 0254-5330. doi: 10.1007/BF02096264. URL `http://dx.doi.org/10.1007/BF02096264`.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_49. URL `http://dx.doi.org/10.1007/978-3-540-71209-1_49`.

Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987. doi: 10.1145/7531.8928. URL `http://doi.acm.org/10.1145/7531.8928`.

Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. Technical report, The University of Tokyo, 2002. Available at `http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf`.