An Introduction to Artificial Intelligence

— Lecture Notes for 2025 —

Prof. Dr. Karl Stroetmann

March 17, 2025

These lecture notes, their LaTeX sources, and the programs discussed in these lecture notes are all available at

https://github.com/karlstroetmann/Artificial-Intelligence.

In particular, the lecture notes are found in the directory Lecture-Notes in the file artificial-intelligence.pdf. The lecture notes are subject to continuous change. Provided the program `git` is installed on your computer, the command

```
git clone https://github.com/karlstroetmann/Artificial-Intelligence.git
```

clones the repository containing the lecture notes and stores them on your local drive. Once you have cloned the repository, the command

```
git pull
```

can be used to load the current version of these lecture notes from `Github`. As artificial intelligence is a very active area of research, these lecture notes will always be incomplete and hence will change from time to time. If you find any typos, errors, or inconsistencies, please contact me via discord or, if that is not possible, email me at:

`karl.stroetmann@dhbw-mannheim.de`

You are also welcome to send a pull request on GitHub.

Contents

Chapter 1

Introduction

Artificial Intelligence has evolved through two primary approaches. The first, known as symbolic AI, focuses on symbolic logic. This approach led to the creation of automatic theorem provers, symbolic integration systems, and chess-playing programs like Deep Blue. Initially, symbolic AI was the predominant paradigm in the field.

The second approach, machine learning, was defined by Arthur Samuel as "the field of study that enables computers to learn without explicit programming" [Sam59]. This approach has primarily fueled the recent hype in AI.

1.1 Overview

This lecture discusses only symbolic AI, as machine learning is part of the module data mining. It emphasizes declarative programming. The core principle of declarative programming involves starting with a formal problem specification, a succinct description of the issue at hand. This specification is then processed by a problem solver to produce a solution. Originally, declarative programming adopted a broad approach to problem-solving, where problems were framed as logical formulas and tackled using automated theorem provers. The programming language Prolog is based on this paradigm. However, this approach has proven to be less effective as a universal problem-solving framework for two reasons:

1. It is often challenging to fully articulate practical problems within a logical framework.

2. In cases where it is possible to completely define a problem using logical formulas, automatic theorem proving generally lacks the capability to autonomously find solutions.

Despite these limitations, declarative programming has proven valuable in several domains, which we will explore, demonstrating its application in solving various types of problems:

1. Search problems, where the objective is to find a path within a graph. A classic instance is the fifteen puzzle. We will examine several advanced algorithms designed to resolve such search problems.

2. Constraint satisfaction problems hold significant practical relevance. Currently, highly efficient constraint solvers exist, capable of addressing various practical constraint satisfaction problems. We will delve into different strategies for solving these problems and discuss Z3, a leading-edge automatic theorem prover and constraint solver developed by Microsoft.

3. Games, such as chess or checkers, can be defined using a declarative approach. We will cover several techniques enabling computers to devise optimal strategies for these adversarial games.

4. Finally, we discuss automatic theorem proving. Having previously covered resolution theorem proving in our lecture on logic, we will now turn our attention to equational theorem proving in the final chapter of this first part.

1.2 Literature

My main sources for these lecture notes were the following:

1. A specialized course on artificial intelligence available through the EDX platform. All relevant course materials can be accessed at http://ai.berkeley.edu/home.html.
2. The book titled *Introduction to Artificial Intelligence*, authored by Stuart Russell and Peter Norvig [RN20].
3. The *Intro to Artificial Intelligence* course provided by the Udacity platform.

For exam preparation, a thorough understanding of the material covered in these lecture notes should suffice. Therefore, purchasing additional books or enrolling in other courses is certainly not necessary.

Remark: The programs presented in these lecture notes are expected to run with the *Python* version 3.12. I have created the Python environment that I am using for these lecture notes via the shell commands shown in Figure 1.1 on page 4.

```
conda create -y -n ai python=3.12
conda activate ai
pip install --upgrade pip
pip install notebook jupyterlab
pip install nbclassic
conda install -c anaconda -y graphviz ply seaborn scikit-learn
conda install -c conda-forge -y python-graphviz matplotlib ipycanvas
pip install nb_mypy
pip install z3-solver
pip install git+https://github.com/reclinarka/problem_visuals
pip install git+https://github.com/reclinarka/chess-problem-visuals
```

Figure 1.1: Bash commands to set up an Anaconda environment for Python.

When starting *Jupyter notebooks* you should take care to use the following command:

```
jupyter nbclassic
```

This command uses the classic version of *Jupyter notebooks*. There are lots of incompatibilites with tne new *Jupyter notebooks* of version $7.x$ and I have found that new version $7.x$ do not work for me.

Chapter 2

Search

This chapter is the first of three chapters where we will solve problems by making use of declarative programming. The idea of declarative programming is that rather than developing a program to solve a specific problem, we implement an algorithm that can solve a whole class of problems. Then, in order to solve a problem that falls within this class, we just have to specify the problem, which is usually much easier than writing a program that solves the given problem. In this chapter, this idea is illustrated via search problems. First, we define the notion of a search problem formally. This notion is then illustrated with two examples. We start with the missionaries and cannibals puzzle. Next, we use the sliding puzzle as our running example. After that, we introduce various algorithms for solving search problems. In particular, we present

1. breadth first search,
2. depth first search,
3. iterative deepening,
4. bidirectional breadth first search.

   The algorithms mentioned work on any search problem. If we have a heuristic that estimates how many steps it takes to solve the given search problem, then a solution can be found much faster. The following algorithms make use of a heuristic:

5. A* search and bidirectional A* search,
6. iterative deepening A* search, and
7. A*-IDA* search.

We proceed to define the notion of a search problem.

Definition 1 (Search Problem) A search problem is a tuple of the form

$$\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$$

where

1. Q is the set of states, also known as the state space.
2. `next_states` is a function taking a state as input and returning the set of those states that can be reached from the given state in one step, i.e. we have

   $$\texttt{next\_states} : Q \to 2^Q.$$

The function `next_states` gives rise to the transition relation R, which is a binary relation on Q, i.e. we have $R \subseteq Q \times Q$. This relation is defined as follows:

$$R := \big\{\langle s_1, s_2\rangle \in Q \times Q \mid s_2 \in \texttt{next\_states}(s_1)\big\}.$$

If either $\langle s_1, s_2\rangle \in R$ or $\langle s_2, s_1\rangle \in R$, then s_1 and s_2 are called neighboring states.

3. `start` is the start state, hence $\texttt{start} \in Q$.

4. `goal` is the goal state, hence $\texttt{goal} \in Q$.

   Sometimes, instead of a single state `goal` there is a set of states `Goals`.

A path is a list $[s_1, \cdots, s_n]$ such that $s_{i+1} \in \texttt{next\_states}(s_i)$ for all $i \in \{1, \cdots, n-1\}$. The length of this path is defined as the length of this list minus 1, i.e. the path $[s_1, \cdots, s_n]$ has length $n-1$. The reason for defining the length of this path as $n-1$ and not n is that the path consists of $n-1$ edges of the form $\langle s_i, s_{i+1}\rangle$ where $i \in \{1, \cdots, n-1\}$. A path $[s_1, \cdots, s_n]$ is a solution to the search problem $\mathcal{P}$ iff the following conditions are satisfied:

1. $s_1 = \texttt{start}$, i.e. the first element of the path is the start state.

2. $s_n = \texttt{goal}$, i.e. the last element of the path is the goal state.

   If instead of a single `goal` we have a set of `Goals`, then the last condition is changed into

   $$s_n \in \texttt{Goals}.$$

A path $p = [s_1, \cdots, s_n]$ is a minimal solution to the search problem $\mathcal{P}$ iff it is a solution and, furthermore, the length of p is minimal among all other solutions. ◇

Remark: In the literature, a state is often called a node. In these lecture notes, I will also sometimes refer to states as nodes. ◇

Example: We illustrate the notion of a search problem with the following example, which is also known as the missionaries and cannibals puzzle: Three missionaries and three infidels have to cross a river that runs from the north to the south. Initially, both the missionaries and the infidels are on the western shore. There is just one small boat that can carry at most two passengers. Both the missionaries and the infidels can steer the boat. However, if at any time the missionaries are confronted with a majority of infidels on either shore of the river, then the missionaries have a problem. Figure 2.1 shows an artist's rendition[1] of the problem.
Figure 2.2 shows a formalization of the missionaries and cannibals puzzle as a search problem. We discuss this formalization line by line.

1. Line 1 defines the auxiliary function `no_problem`.

   If m is the number of missionaries on the western shore and i is the number of infidels on that shore, then the expression $\texttt{no\_problem}(m, i)$ is `True`, if there is no problem for the missionaries on either shore. There are three cases where there is no problem:

   (a) all missionaries are on the left shore, i.e. $m = 3$, or

   (b) all missionaries are on the right shore, i.e. $m = 0$, or

   (c) the number of missionaries is the same as the number of infidels, i.e. $m = i$.

[1] Thanks to Marcel Vilas for providing this beautiful painting as well as an animation of this problem.

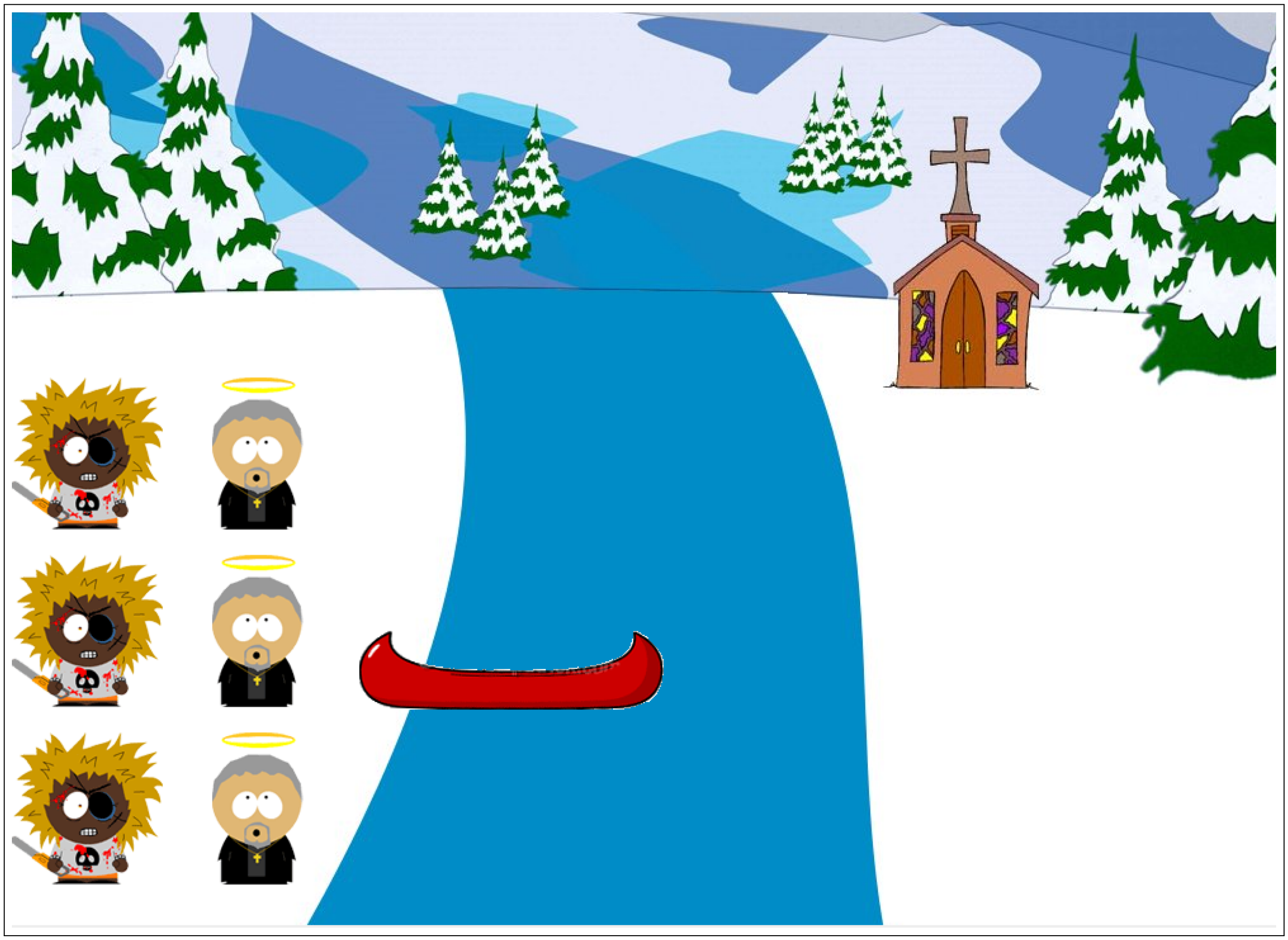


Figure 2.1: Start state of the missionaries-and-infidels problem.

2. Lines 4 to 17 define the function `next_states`. A state s is represented as a triple of the form

$$s = (m, i, b) \quad \text{where } m \in \{0, 1, 2, 3\},\ i \in \{0, 1, 2, 3\}, \text{ and } b \in \{0, 1\}.$$

   Here m, i, and b are, respectively, the number of missionaries, the number of infidels, and the number of boats on the western shore.

   (a) Line 5 extracts the components m, i, and b from the state s.
   (b) Line 6 checks whether the boat is on the western shore.
   (c) If this is the case, then the states reachable from the given state s are those states where `mb` missionaries and `ib` infidels cross the river. After `mb` missionaries and `ib` infidels have crossed the river and reached the eastern shore, `m`$-$`mb` missionaries and `i`$-$`ib` infidels remain on the western shore. Of course, after the crossing, the boat is no longer on the western shore. Therefore, the new state has the form

```
(m - mb, i - ib, 0).
```

   This explains line 10.
   (d) Since the number `mb` of missionaries leaving the western shore can not be greater than the number m of all missionaries on the western shore, we have the condition

```python
def no_problem(m: int, i: int) -> bool:
    return m == 0 or m == 3 or m == i

def next_states(state):
    m, i, b = state
    if b == 1:
        return { (m - mb, i - ib, 0) for mb in range(m+1)
                                     for ib in range(i+1)
                                     if 1 <= mb + ib <= 2 and
                                        no_problem(m - mb, i - ib)
               }
    else:
        return { (m + mb, i + ib, 1) for mb in range(3-m+1)
                                     for ib in range(3-i+1)
                                     if 1 <= mb + ib <= 2 and
                                        no_problem(m + mb, i + ib)
               }

start = (3, 3, 1)
goal  = (0, 0, 0)
```

Figure 2.2: The missionary and cannibals problem coded as a search problem.

$$\texttt{mb} \in \{0, \cdots, \mathrm{m}\},$$

which is implemented by the line

`for mb in range(m+1).`

There is a similar condition for the number of infidels crossing:

$$\texttt{ib} \in \{0, \cdots, \mathrm{i}\}$$

which is implemented by

`for ib in range(i+1).`

(e) Furthermore, we have to check that the number of persons crossing the river is at least 1 and at most 2. This explains the condition

`1 <= mb + ib <= 2.`

(f) Finally, there should be no problem in the new state on either shore. This is checked using the expression

`noProblem(m - mb, i - ib).`

3. If the boat is on the eastern shore instead, then the missionaries and the infidels will be crossing the river from the eastern shore to the western shore. Therefore, the number of missionaries and infidels on the western shore is now increased. Hence, in this case the new state has the form

`(m + mb, i + ib, 1).`

Here, `mb` is the number of missionaries arriving on the western shore and `ib` is the number of arriving infidels. As the number of missionaries on the eastern shore is $3 - \mathrm{m}$ and the number of infidels on the eastern shore is $3 - \mathrm{i}$, `mb` has to be a member of the set $\{0, \cdots, 3 - \mathtt{m}\}$, while `ib` has to be a member of the set $\{0, \cdots, 3 - \mathtt{i}\}$.

4. Finally, the start state and the goal state are defined in line 22 and line 23.

The code in Figure 2.2 does not define the set of states Q of the search problem. The reason is that, in order to solve the problem, we do not need to define this set. If we wanted to, we could define the set of states as follows:

```
States = { (m, i, b) for m in {0, 1, 2, 3}
                     for i in {0, 1, 2, 3}
                     for b in {0, 1}
                     if no_problem(m, i)
         }
```

However, in general the set of states is not needed by the algorithms solving search problems and in many cases this set is so big that it would be impossible to store it. Hence, in practice the set of states is only an abstract notion that is needed in order to specify the function `next_states`, but it is not implemented.

Figure 2.3 shows a graphical representation of the transition relation of the missionaries and cannibals puzzle. In that figure, for every state, both the western and the eastern shore are shown. The start state is covered with a blue ellipse, while the goal state is covered with a green ellipse. The figure clearly shows that the problem is solvable and that there is a solution involving just 11 crossings of the river. ◇

Exercise 1: The *Three Thieves Puzzle* is similar to the *Missionaries and Cannibals Puzzle*. Three greedy thieves have cross a river. Each of the thieves has a bag of gold coins.

1. Ariel has 1,000 gold coins.
2. Benjamin has 700 gold coins.
3. Claude has 300 gold coins.

There is a boat available that can carry either two people or one person along with a bag of gold coins. The boat can transport two entities at a time, meaning either two thieves or one thief and a bag can cross together. A problem arises if a thief, or a pair of thieves, is left with a quantity of gold greater than their own, since then they will take the money and run away with it.

Write a program that formulates this puzzle as a search problem. You should do this by augenting the notebook `Three-Tieves.ipynb` that can be found in the github repository

http://github.com/karlstroetmann/Artificial-Intelligence

in the directory Python/1 Search/02-Three-Thieves.ipynb. ◇

2.1 The Sliding Puzzle

The missionaries and cannibals puzzle is rather small and therefore it is not useful when we want to compare the efficiency of various algorithms for solving search problems. Therefore, we will now present a problem that has a greater complexity: The 3×3 sliding puzzle uses a square board, where each side has a length of 3. This board is subdivided into $3 \times 3 = 9$ squares of length 1. Of these 9

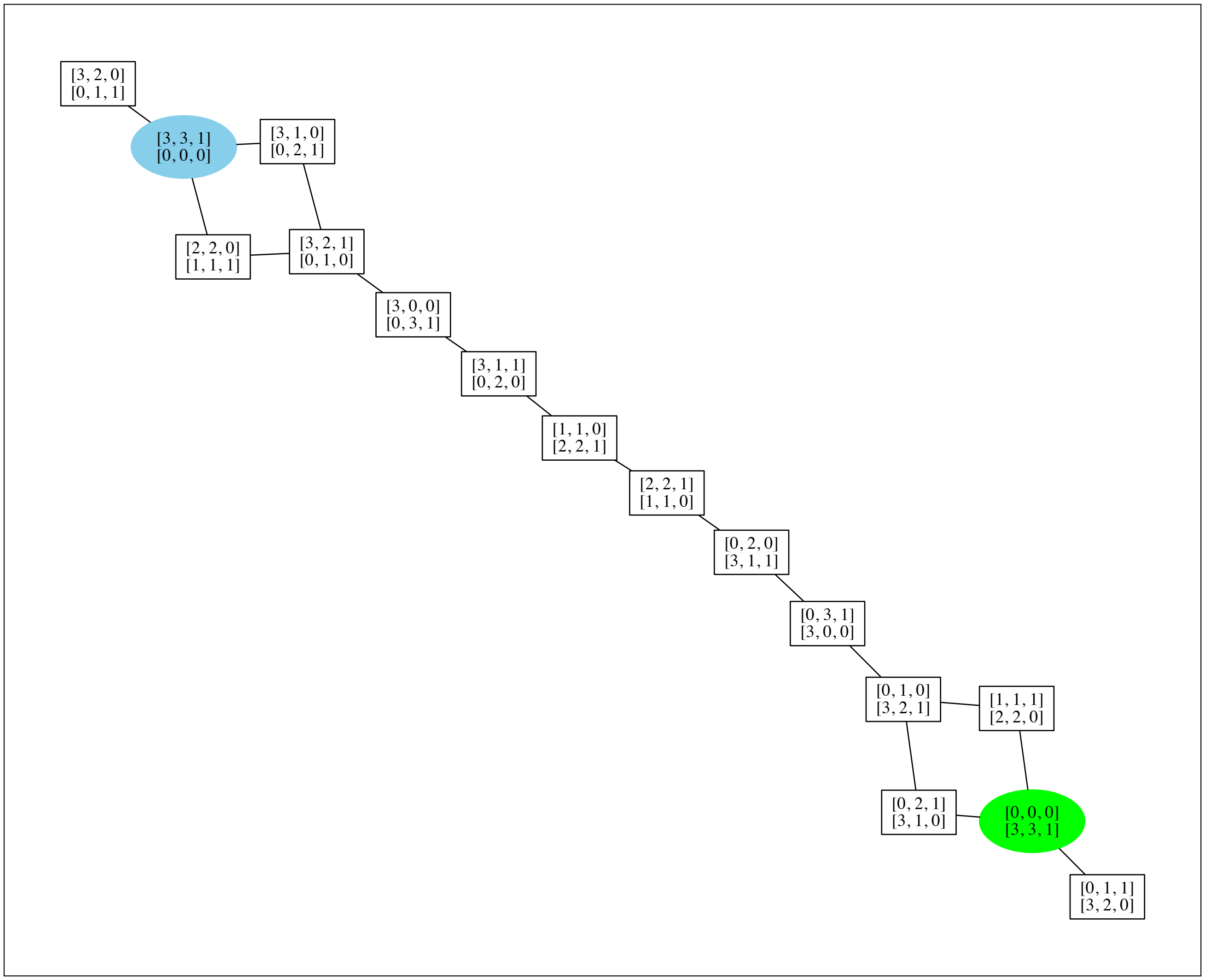


Figure 2.3: A graphical representation of the missionaries and cannibals puzzle.

squares, 8 are occupied with square tiles that are numbered from 1 to 8. One square remains empty. Figure 2.4 on page 10 shows two possible states of this sliding puzzle. The 4×4 sliding puzzle is similar to the 3×3 sliding puzzle, but uses a square board of size 4 instead. The 4×4 sliding puzzle is also known as the 15 puzzle, while the 3×3 puzzle is called the 8 puzzle.

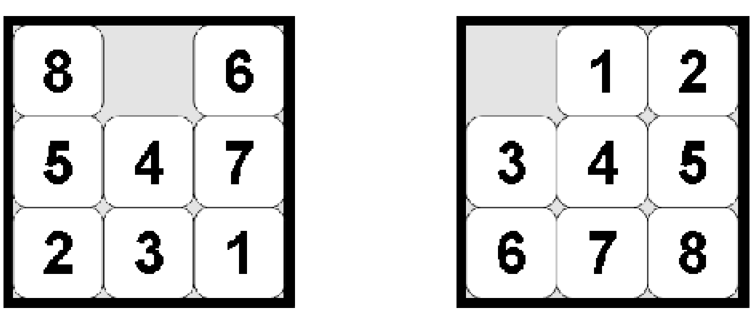


Figure 2.4: The 3×3 sliding puzzle.

In order to solve the 3×3 sliding puzzle shown in Figure 2.4 we have to transform the state shown on the left of Figure 2.4 into the state shown on the right of this figure. The following operations are permitted when transforming a state of the sliding puzzle:

1. If a tile is to the left of the free square, this tile can be moved to the right.
2. If a tile is to the right of the free square, this tile can be moved to the left.
3. If a tile is above the free square, this tile can be moved down.
4. If a tile is below the free square, this tile can be moved up.

In order to get a feeling for the complexity of the sliding puzzle, you can check the page

https://www.helpfulgames.com/subjects/brain-training/sliding-puzzle.html.

The sliding puzzle is much more complex than the missionaries and cannibals puzzle because the state space is much larger. For the case of the 3×3 sliding puzzle, there are 9 squares that can be positioned in 9! different ways. It turns out that only half of these positions are reachable from a given start state. Therefore, the effective number of states for the 3×3 sliding puzzle is

$$9!/2 = 181,440.$$

This is already a big number, but 181,440 states can easily be stored on a modern computer. However, the 4×4 sliding puzzle has

$$16!/2 = 10,461,394,944,000$$

different states reachable from a given start state. If a state is represented as a matrix containing 16 numbers and we store every number using just 4 bits, we still need $16 \cdot 4 = 64$ bits or 8 bytes for every state. Hence, we would need a total of

$$(16!/2) \cdot 8 = 83,691,159,552,000$$

bytes to store every state. We would thus need about 84 terabytes to store the set of all states. As few computers are equipped with this kind of memory, it is obvious that we won't be able to store the entire state space in memory.

Figure 2.5 shows how the 3×3 sliding puzzle can be formulated as a search problem. In order to discuss the program, we first have to understand that states are represented as tuples of tuples. For example, the state shown above on the left side in Figure 2.4 is represented as the tuple:

```
( (8, 0, 6),
  (5, 4, 7),
  (2, 3, 1)
)
```

Here, we have represented the empty tile as 0. If states are represented as tuples of tuples, given a state s, the expression $s[r][c]$ returns the tile in the row r and column c, where the counting of rows and columns starts from 0. We have to represent states as tuples of tuples rather than lists of lists since tuples are immutable while lists are mutable and we need to store states in sets later. In *Python*, sets can only store immutable objects. However, we also have to manipulate the states. To this end, we have to first transform the states to lists of lists, which can be manipulated. After the manipulation, these lists of lists have to be transformed back to tuples of tuples. We proceed to discuss the program shown in Figure 2.5 line by line.

1. The function `to_list` transforms a tuple of tuples into a list of lists.

```
def to_list(State):
    return [list(row) for row in State]
def to_tuple(State):
    return tuple(tuple(row) for row in State)
def find_tile(tile, State):
    n = len(State)
    for row in range(n):
        for col in range(n):
            if State[row][col] == tile:
                return row, col

def move_dir(State, row, col, dx, dy):
    State = to_list(State)
    State[row     ][col     ] = State[row + dx][col + dy]
    State[row + dx][col + dy] = 0
    return to_tuple(State)

def next_states(State):
    n          = len(State)
    row, col   = find_tile(0, State)
    New_States = set()
    Directions = [ (1, 0), (-1, 0), (0, 1), (0, -1) ]
    for dx, dy in Directions:
        if row + dx in range(n) and col + dy in range(n):
            New_States.add(move_dir(State, row, col, dx, dy))
    return New_States

start = ( (8, 0, 6),
          (5, 4, 7),
          (2, 3, 1)
        )

goal = ( (0, 1, 2),
         (3, 4, 5),
         (6, 7, 8)
       )
```

Figure 2.5: The 3×3 sliding puzzle.

2. The function `to_tuple` transforms a list of lists into a tuple of tuples.

3. `find_tile` is an auxiliary function that is needed to implement the function `next_states`. It is called with a `number` and a `State` and returns the row and column where the tile labelled with `number` can be found.

4. `move_dir` takes a `State`, the `row` and the `column` where to find the empty square and a direction in which the empty square should be moved. This direction is specified via the two variables `dx`

and dy. The tile at the position $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ is moved into the position $\langle \texttt{row}, \texttt{col} \rangle$, while the tile at position $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ becomes empty.

5. Given a `State`, the function `next_states` computes the set of all states that can be reached in one step from `State`. The basic idea is to find the position of the empty tile and then try to move the empty tile in all possible directions. If the empty tile is found at position $\langle \texttt{row}, \texttt{col} \rangle$ and the direction of the movement is given as $\langle \texttt{dx}, \texttt{dy} \rangle$, then in order to ensure that the empty tile can be moved to the position $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$, we have to ensure that both

$$\texttt{row} + \texttt{dx} \in \{0, \cdots, n-1\} \quad \text{and} \quad \texttt{col} + \texttt{dy} \in \{0, \cdots, n-1\}$$

hold, where n is the size of the board. ◇

Next, we want to develop an algorithm that can solve puzzles of the kind described so far. The most basic algorithm to solve search problems is breadth first search. We discuss this algorithm next.

2.2 Breadth First Search

Informally, breadth first search, abbreviated as BFS, works as follows:

1. Given a search problem $\langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$, we initialize a set `Frontier` to contain the state `start`.

   In general, `Frontier` contains those states that have just been discovered and whose successors have not yet been seen.

2. As long as the set `Frontier` does not contain the state `goal`, we recompute this set by adding all states to it that can be reached in one step from a state in `Frontier`. Then, the states that had been previously present in `Frontier` are removed. These old states are then added to the set `Visited`.

In order to avoid going around in circles, an implementation of breadth first search keeps track of those states that have been visited in the set `Visited`. Once a state has been added to the set `Visited`, it will never be revisited again. Furthermore, in order to keep track of the path leading to the goal, we utilize a dictionary called `Parent`. For every state s that is in `Frontier`, $\texttt{Parent}[s]$ is the state that has caused s to be added to the set `Frontier`, i.e. for all states $s \in \texttt{Frontier}$ we have

$$s \in \texttt{next\_states}(\texttt{Parent}[s]).$$

Figure 2.6 on page 14 shows an implementation of breadth first search in *Python*. The function `search` takes three arguments to solve a search problem:

(a) `start` is the start state of the search problem,

(b) `goal` is the goal state of the search problem, and

(c) `next_states` is a function with signature

$$\texttt{next\_states} : Q \to 2^Q,$$

where Q is the set of states. For every state $s \in Q$, $\texttt{next\_states}(s)$ is the set of states that can be reached from s in one step.

If successful, `search` returns a path from `start` to `goal` that is a solution of the search problem

$$\langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle.$$

Next, we discuss the implementation of the function `search`:

```
def search(start, goal, next_states):
    Frontier = { start }
    Visited  = set()
    Parent   = { start: start }
    while Frontier:
        NewFrontier = set()
        for s in Frontier:
            for ns in next_states(s):
                if ns not in Visited:
                    NewFrontier.add(ns)
                    Parent[ns] = s
                    if ns == goal:
                        return path_to(goal, Parent)
        Visited |= Frontier
        Frontier = NewFrontier
```

Figure 2.6: Breadth first search.

1. `Frontier` is the set of all those states that have been encountered but whose neighbours have not yet been explored. Initially, it contains the state `start`.

   After the n^{th} iteration of the `while` loop, every state s in the set `Frontier` has a distance of n from the node `start`, i.e. there is a path of length n leading from `start` to s.

2. `Visited` is the set of all those states, all of whose neighbours have already been added to the set `Frontier` in the last iteration of the `while` loop. In order to avoid infinite loops, these states must not be visited again.

3. `Parent` is a dictionary keeping track of the predecessors of the state that have been reached. The only state with no real predecessor is the state `start`. By convention, `start` is its own predecessor.

4. As long as the set `Frontier` is not empty, we add all neighbours of states in `Frontier` that have not yet been visited to the set `NewFrontier`. When doing this, we keep track of the path leading to a new state `ns` by storing its parent in the dictionary `Parent`.

5. If the new state happens to be the state `goal`, we return a path leading from `start` to `goal` by calling the function `path_to`. This function is shown in Figure 2.7 on page 15.

6. After we have collected all successors of states in `Frontier`, the states in the set `Frontier` have been visited and are therefore added to the set `Visited`, while the set `Frontier` is updated to `NewFrontier`.

The function call `path_to(state,Parent)` constructs a path reaching from `start` to `state` in reverse by looking up the parent states. It uses the fact that only the start state is its own parent.

If we try breadth first search to solve the missionaries and cannibals puzzle, we obtain the solution shown in Figure 2.8. 15 nodes had to be expanded to find this solution. To keep this in perspective, we note that Figure 2.3 shows that the entire state space contains 16 states. Therefore, with the

```python
def path_to(state, Parent):
    p = Parent[state]
    if p == state:
        return [state]
    return path_to(p, Parent) + [state]
```

Figure 2.7: The function `path_to`.

```
MMM   KKK   B      |~~~~~|
                   >  KK >
MMM   K            |~~~~~|              KK   B
                   <  K  <
MMM   KK    B      |~~~~~|               K
                   >  KK >
MMM                |~~~~~|             KKK   B
                   <  K  <
MMM   K     B      |~~~~~|              KK
                   > MM  >
M     K            |~~~~~|        MM    KK   B
                   < M K <
MM    KK    B      |~~~~~|         M     K
                   > MM  >
      KK           |~~~~~|       MMM     K   B
                   <  K  <
      KKK   B      |~~~~~|       MMM
                   >  KK >
      K            |~~~~~|       MMM    KK   B
                   <  K  <
      KK    B      |~~~~~|       MMM     K
                   >  KK >
                   |~~~~~|       MMM   KKK   B
```

Figure 2.8: A solution of the missionaries and cannibals puzzle.

exception of one state, we have inspected all the states. If the search problem is difficult, then this is a typical behaviour of breadth first search.

Next, let us try to solve the 3×3 sliding puzzle. It takes about 1.2 seconds to solve this problem on my computer[2], while 181,439 states are touched. Again, we see that breadth first search touches nearly all the states reachable from the start state. If we measure the memory consumption, we discover that the program uses about 90 megabytes of memory.

Breadth first search has two important properties:

(a) Breadth first search is complete: If there is a solution to the given search problem, then breadth

[2] My computer is a Mac Studio from 2022. This iMac is equipped with 64 Gigabytes of main memory and an Apple M1 Max processor.

first search is going to find it.

(b) The solution found by breadth first search is optimal, i.e. it is one of the shortest possible solutions.

Proof: Both of these claims can be shown simultaneously. Consider the implementation of breadth first search shown in Figure 2.6 on page 14. We prove by induction on the number of iterations of the `while` loop that after n iterations of the `while` loop, the set `Frontier` contains exactly those states that have a distance of n to the state `start`.

Base Case: $n = 0$.

After 0 iterations of the `while` loop, i.e. before the first iteration of this loop, the set `Frontier` only contains the state `start`. As this is the only state that has a distance of 0 to the state `start`, the claim is true in this case.

Induction Step: $n \mapsto n + 1$.

In the induction step we assume the claim is true after n iterations. Then, in the next iteration all states that can be reached in one step from a state in `Frontier` are added to the new `Frontier`, provided there is no shorter path to them. By induction hypothesis, there is a shorter path to a state if this state is already a member of the set `Visited`. In this case, the state would not be added to `NewFrontier`. Otherwise, the shortest path to a state that is reached in iteration $n+1$ has the length $n+1$ and the state is added to `NewFrontier`. Hence, the claim is true after $n+1$ iterations also.

Now, if there is a path from `start` to `goal`, there must also be a shortest path. Assume this path has a length of k. Then, `goal` is reached in the k^{th} iteration and the shortest path is returned. □

The fact that breadth first search is both complete and the path returned is optimal is rather satisfying. However, breadth first search still has a big downside that makes it unusable for many problems: If the `goal` is far from the `start`, breadth first search will use a lot of memory because it will store a large part of the state space in the set `Visited`. In many cases, the state space is so big that this is impossible. For example, it is impossible to solve the more interesting cases of the 4×4 sliding puzzle using breadth first search.

2.2.1 A Queue Based Implementation of Breadth First Search

In the literature, for example in Figure 3.9 of Russell & Norvig [RN20], breadth first search is often implemented using a queue data structure.

Figure 2.9 on page 17 shows an implementation of breadth first search that uses a queue to store the set `Frontier`. Here we use the module `deque` from the package `collections`. This module implements a double-ended queue, which is implemented as a doubly linked list. Besides the constructor, our implementation uses two methods from the class `deque`:

1. Line 4 initializes the `Frontier` as a double-ended queue that contains the state `start`.
2. In line 7 we remove the oldest element in the queue `Frontier`, which is supposed to be at the left end of the queue. This is achieved via the method `popleft`.
3. In line 14 we add the states that have not been encountered previously at the right end of the queue `Frontier` using the method `append`.

Additionally, we have used the fact that the information contained in the set `Visited` is already available in the dictionary `Parent`, because when we visit a state s, we add an entry for `Parent`$[s]$. As a result, this implementation of breadth first search is slightly faster than our previous implementation. Furthermore, only 76 megabytes of memory are used for the computation.

```
from collections import deque

def search(start, goal, next_states):
    Frontier = deque([start])
    Parent   = { start: start }
    while Frontier:
        state = Frontier.popleft()
        if state == goal:
            return path_to(state, Parent)
        for ns in next_states(state):
            if ns not in Parent:
                Parent[ns] = state
                Frontier.append(ns)
```

Figure 2.9: A queue based implementation of breadth first search.

2.3 Depth First Search

To overcome the memory limitations of breadth first search, the depth first search algorithm has been developed. Depth first search is abbreviated as DFS. While BFS ensures that every state is visited by implementing the `Frontier` as a queue, DFS replaces this queue by a stack. This way, DFS tries to get as far away from the state `start` as early as possible. In order to prevent the search from looping, we still have the parent dictionary.

```
def search(start, goal, next_states):
    Stack  = [start]
    Parent = { start: start }
    while Stack:
        state = Stack.pop()
        for ns in next_states(state):
            if ns not in Parent:
                Parent[ns] = state
                if ns == goal:
                    return path_to(goal, Parent)
                Stack.append(ns)

def path_to(state, Parent):
   Path = [state]
   while state != Parent[state]:
       state = Parent[state]
       Path  = [ state ] + Path
   return Path
```

Figure 2.10: The depth first search algorithm.

Since a stack can be implemented as an ordinary *Python* list, we don't need the module `deque` anymore. The idea is that the top of the stack is at the end of this list. Therefore, when we `pop` an element from the stack, it is removed from the end of the list, while we can push an element onto the stack by using the method `append`. The resulting algorithm is shown in Figure 2.10 on page 17. Basically, in this implementation, a path is searched to its end before trying an alternative. This way, we might be able to find a `goal` that is far away from `start` without exploring the whole state space.

The implementation of `search` works as follows:

1. Any states that are encountered during the search are placed on top of the stack `Stack`.
2. In order to record the information how a state has been added to the `Stack`, we have the dictionary `Parent`. For every state s that is on `Stack`, `Parent`$[s]$ returns a state p such that $s \in$ `next_states`(p), i.e. p is the state that immediately precedes s on the path that leads from `start` to s.
3. Initially, `Stack` only contains the state `start`.
4. As long as `Stack` is not empty, the `state` on top of `Stack` is replaced by all states that can be reached in one step from `state`. However, in order to prevent depth first search from running in circles, only those states `ns` from the set `next_states(state)` are appended to `Stack` that have not been encountered previously. This is checked by testing whether `ns` is in the domain of `Parent`.
5. When the `goal` is reached, a path leading from `start` to `goal` is returned.
6. We have reimplemented the function `path_to` using a `while` loop. The reason is that the recursive implementation that we had used before is not viable when the path gets too long because the recursion limit in *Python* is set to 3000 and hence the previous implementation of `path_to` does not work if the path exceeds a length of 3000.

When we test the implementation shown above with the 3×3 sliding puzzle, it takes 264 milliseconds on my computer to find a solution. This is an improvement compared to breadth first search. The memory consumption is reduced to 3 megabytes. This is still a lot and is due to the fact that we still have to maintain the dictionary `Parent`. Fortunately, we will be able to get rid of the dictionary `Parent` when we develop a recursive implementation of depth first search in the following subsection.

However, there is also bad news: the solution that is found has a length of $17,510$ steps. As the shortest path from `start` to `goal` has only 31 steps, the solution found by depth first search is very far from being optimal.

2.3.1 A Recursive Implementation of Depth First Search

Sometimes, the depth first search algorithm is presented as a recursive algorithm, since this leads to an implementation that is slightly shorter and also easier to understand. What is more, we no longer need the dictionary `Parent` to record the parent of each node. The resulting implementation is shown in Figure 2.11 on page 19.

The only purpose of the function `search` is to call the function `dfs`, which needs two additional arguments. These arguments are called `Path` and `PathSet`. The idea is that `Path` is a path leading from the state `start` to the current `state` that is the first argument of the function `dfs`, while `PathSet` is a set containing all the elements of the path `Path`. The argument `PathSet` is only used for efficiency reasons: In order to avoid infinite loops, when we discover a node we have to check that this node does not occur already in `Path`. However, checking whether an element occurs in the list `Path` is much slower than checking whether the element occurs in the corresponding set `PathSet`. On the first

```
def search(start, goal, next_states):
    return dfs(start, goal, next_states, [start], { start })

def dfs(state, goal, next_states, Path, PathSet):
    if state == goal:
        return Path
    for ns in next_states(state):
        if ns not in PathSet:
            Result = dfs(ns, goal, next_states, Path + [ns], PathSet | {ns})
            if Result:
                return Result
```

Figure 2.11: A recursive implementation of depth first search.

invocation of `dfs`, the parameter `state` is equal to `start` and therefore `Path` is initialized as the list containing only `start`.

The implementation of `dfs` works as follows:

1. If `state` is equal to `goal`, our search is successful. Since by assumption the list `Path` is a path connecting `start` and `state` and we have checked that `state` is equal to `goal`, we can return `Path` as our solution.

2. Otherwise, `next_states(state)` is the set of states that are reachable from `state` in one step. Any of the states `ns` in this set could be the next state on a path that leads to `goal`. Therefore, we try recursively to reach `goal` from every state `ns`. Note that we have to change `Path` to the list

   ```
   Path + [ ns ]
   ```

   when we call the function `dfs` recursively. This way, we retain the invariant of `dfs` that the list `Path` is a path connecting `start` with `state`.

3. In the same spirit we have to change `PathSet` to the set

   ```
   PathSet | { ns }
   ```

   since we have to maintain the invariant that `PathSet` is the set of all nodes in `Path`.

4. We still have to avoid running in circles. In the recursive version of depth first search, this is achieved by checking that the state `ns` is not already a member of the set `PathSet`. In the non-recursive version of depth first search, we had used the dictionary `Parent` instead. The current implementation no longer has a need for the dictionary `Parent`. This is very fortunate since it reduces the memory requirements of depth first search considerably.

5. If one of the recursive calls of `dfs` returns a list, this list is a solution to our search problem and hence it is returned. However, if instead `None` is returned, the `for` loop needs to carry on and test the other successors of `state`.

6. Note that the recursive invocation of `dfs` returns `None` if the end of the `for` loop is reached and no solution has been returned so far.

Unfortunately, due to a bug in *Python* `3.12`, the *Python* kernel just dies when trying to solve the 3×3 sliding puzzle. This is due to the fact that the path gets very long and the garbage collector is not reclaiming the memory.

2.4 Iterative Deepening

The fact that the stack-based version of depth first search took less than one second to find a solution is very impressive, but the fact that this solution has a length of more than ten thousand steps is disappointing. The question is, whether it might be possible to force depth first search to find the shortest solution. The answer to this question leads to an algorithm that is known as iterative deepening. The main idea behind iterative deepening is to run depth first with a depth limit d. This limit enforces that a solution has at most a length of d. If no solution is found at a depth of d, the new depth $d+1$ can be tried next and the process can be continued until a solution is found. The program shown in Figure 2.12 on page 20 implements this strategy. There is one simplification that we can apply: As the search will always find the shortest path, there is no need to keep the dictionary `PathSet` around. Instead of checking whether a node is a member of `PathSet`, we can just check whether it is a member of the list `Path`. This works, because searching in a small list does not take much more time than searching in a small set. We proceed to discuss the details of the implementation.

```python
def search(start, goal, next_states):
    limit = 1
    while True:
        Path = dls(start, goal, next_states, [start], limit)
        if Path is not None:
            return Path
        limit += 1

def dls(state, goal, next_states, Path, limit):
    if state == goal:
        return Path
    if len(Path) == limit:
        return None
    for ns in next_states(state):
        if ns not in Path:
            Result = dls(ns, goal, next_states, Path + [ns], limit)
            if Result:
                return Result
```

Figure 2.12: Iterative deepening implemented in *Python*.

1. The function `search` initializes the variable `limit` to 1 and tries to find a solution to the search problem that has a length that is less than or equal to `limit`. If a solution is found, it is returned. Otherwise, the variable `limit` is incremented by one and a new instance of depth first search is started. This process continues until either a solution is found or the sun rises in the west.
2. The function `dls` implements a recursive version of depth first search but takes care to compute only those paths that have a length of at most `limit`. The name `dls` is short for depth limited

search. If the `Path` has reached a length of `limit` but does not end in `goal`, the function returns `None` instead of trying to extend this `Path`. Otherwise, the implementation is the same as the recursive implementation of depth first search that was shown in Figure 2.11 on page 19 and that has been discussed in the previous section. The only difference is that we no longer need to use the set `PathSet`.

The nice thing about the program presented in this section is the fact that it uses only 136 kilobytes of memory. The reason is that the `Path` can never have a size that is longer than `limit`. However, when we run this program to solve the 3×3 sliding puzzle, the algorithm takes about 7 minutes. There are two reasons for the long computation time:

1. First, it is quite wasteful to run the search for a depth limit of 1, 2, 3, $\cdots$ all the way up to 32. Essentially, all the computations done with a limit less than 32 are wasted. However, this process is not as wasteful as we might first expect. To see this, assume that the number of states reached is doubled[3] after every iteration. Then the number of states to explore when searching with a depth limit of d is roughly 2^d. Hence, when we run depth limited search up to depth d, the number of states visited is

$$1 + 2^1 + 2^2 + \cdots + 2^d = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1.$$

   Therefore, if the solution is found at a depth of $d+1$, we will explore at most 2^{d+1} states to find the solution if we would do depth first search with a depth limit of $d+1$. If, instead, we use iterative deepening, we have wastefully explored an additional number of $2^{d+1}-1$ states. Hence, we will visit only about twice the number of states with iterative deepening than we would have visited with depth limited search with the correct depth limit.

2. Given a state s that is reachable from the `start`, there often are a huge number of different paths that lead from `start` to s. The version of iterative deepening presented in this section tries all of these paths and hence needs a large amount of time.

To check what is really going on, we can change the initial value of `limit` that is set to 1 in line 2 of Figure 2.12 on page 20. If we set this value to 31, which is one less that the value that is needed, the program needs about 5 minutes to compute the solution. However, if this value is set to 32, then the program is able to find the solution in less than two minutes. The reason is that in the case that `limit` has the value 31, the program has to check all possible lists `Path` that have a length of at most 31. Unfortunately, there is no such list, so all possible states that have a distance of at most 30 from `start` have to be explored. However, if `limit` has the value 32, it is sufficient to find any `Path` of length 32 that leads to the `goal` and if that `Path` has been found, the program can return immediately. The following exercise digs deeper into this observation.

Exercise 2: Assume the set of states Q is defined as

$$Q := \bigl\{\langle a, b\rangle \mid a \in \mathbb{N} \wedge b \in \mathbb{N}\bigr\}.$$

Furthermore, the states `start` and `goal` are defined as

$$\texttt{start} := \langle 0, 0\rangle \quad \text{and} \quad \texttt{goal} := \langle n, n\rangle \text{ where } n \in \mathbb{N}.$$

Next, the function `next_states` is defined as

$$\texttt{next\_states}\bigl(\langle a, b\rangle\bigr) := \bigl\{\langle a+1, b\rangle, \langle a, b+1\rangle\bigr\}.$$

[3] When we run breadth first search for the sliding puzzle, we can observe that at least at the beginning of the search, the number of states is roughly doubled after each step. This observation holds true for the first 16 steps.

Finally, the search problem $\mathcal{P}$ is defined as

$$\mathcal{P} := \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle.$$

Given a natural number n, compute the number of different solutions of this search problem and prove your claim. The Figure 2.13 on page 22 shows possible solutions in a graph.

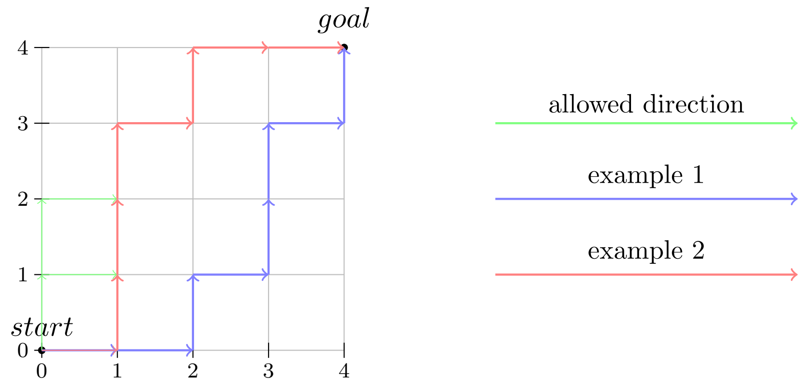


Figure 2.13: Example for possible paths in a graph

Hint: The expression giving the number of different solutions contains factorials. In order to get a better feeling for the asymptotic growth of this expression we can use Stirling's approximation of the factorial. Stirling's approximation of $n!$ is given as follows:

$$n! \sim \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n.$$ ◇

Exercise 3: If there is no solution, the implementation of iterative deepening that is shown in Figure 2.12 does not terminate. The reason is that the function `dls` does not distinguish between the following two reasons for failure:

(a) It can fail to find a solution because the depth limit is reached.

(b) It can also fail because it has exhausted all possible paths without hitting the depth limit.

Improve the implementation of iterative deepening so that it will always terminate eventually, provided the state space is finite. ◇

2.5 Bidirectional Breadth First Search

Breadth first search first visits all states that have a distance of 1 from start, then all states that have a distance of 2, then of 3 and so on until finally the goal is found. If the length of the shortest path from `start` to `goal` is d, then all states that have a distance of at most d will be visited. In many search problems, the number of states grows exponentially with the distance, i.e. there is a branching factor b such that the set of all states that have a distance of at most d from `start` is roughly

$$1 + b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} = \mathcal{O}\big(b^d\big).$$

At least this is true in the beginning of the search. As the size of the memory that is needed is the most constraining factor when searching, it is important to cut down this size. If the search problem is symmetrical, i.e. if we have

$$x \in \texttt{next\_states}(y) \;\Leftrightarrow\; y \in \texttt{next\_states}(x),$$

then a simple idea is to start searching both from the node `start` and the node `goal` simultaneously. This approach is known as bidirectional search. All of the search problems that we have encountered so far are symmetrical.

The justification for bidirectional search is that the path starting from `start` and the path starting from `goal` will meet in the middle and hence they will both have a size of approximately $d/2$. If this is the case, only

$$2 \cdot (1 + b + \cdots + b^{\frac{d}{2}}) = 2 \cdot \frac{b^{\frac{d}{2}+1} - 1}{b - 1}$$

nodes need to be explored and even for modest values of b this number is much smaller than

$$\frac{b^{d+1} - 1}{b - 1}$$

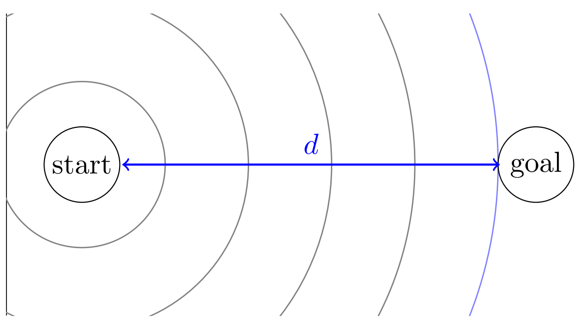
which is the number of nodes expanded in breadth first search. For example, assume that the branching factor $b = 2$ and that the length of the shortest path leading from `start` to `goal` is $d = 40$. Then we need to explore

$$2^{41} - 1 = 2,199,023,255,551$$

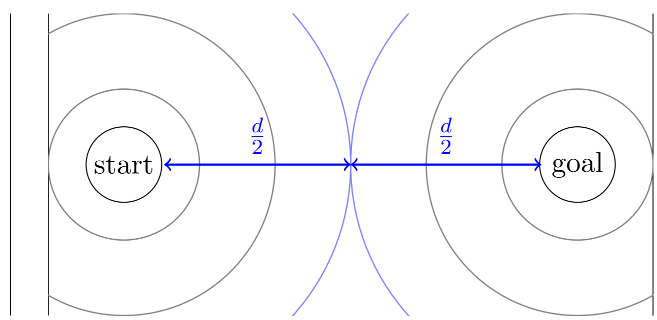
states in breadth first search, while we only have to explore

$$2 \cdot \left(2^{\frac{40}{2}+1} - 1\right) = 4,194,302$$

states with bidirectional breadth first search. While it is certainly feasible to keep four million states in memory, keeping two trillion states in memory is impossible on most devices. The Figure 2.14 on page 23 demonstrates that the conventional search algorithm has to use a lot more space than the bidirectional approach.



(a) Conventional search



(b) Bidirectional search

Figure 2.14: Example of space usage of conventional and bidrectional-BFS

Figure 2.15 on page 24 shows the implementation of bidirectional breadth first search. Essentially, we have two copies of the breadth first search program shown in Figure 2.6. However, since the information that was stored in the set `Visited` in the implementation of BFS shown in Figure 2.6 is also available in the dictionary `Parent`, we have removed the variable `Visited` in our implementation of bidirectional breadth first search.

Let us discuss the details of the implementation.

1. The variable `FrontierA` is the frontier that starts from the state `start`, while `FrontierB` is the frontier that starts from the state `goal`.

2. For every state s that is in `FrontierA`, `ParentA`$[s]$ is the state that caused s to be added to the set `FrontierA`. Similarly, for every state s that is in `FrontierB`, `ParentB`$[s]$ is the state that caused s to be added to the set `FrontierB`.

```
1  def search(start, goal, next_states):
2      FrontierA = { start }
3      ParentA   = { start: start }
4      FrontierB = { goal }
5      ParentB   = { goal: goal }
6      while FrontierA and FrontierB:
7          if Path := bfs_one_step(FrontierA, ParentA, ParentB, next_states):
8              return Path
9          if Path := bfs_one_step(FrontierB, ParentB, ParentA, next_states):
10             return Path[::-1]
```

Figure 2.15: Bidirectional breadth first search.

3. The bidirectional search keeps running for as long as both sets `FrontierA` and `FrontierB` are non-empty and a path has not yet been found.

4. In line 7, the function `bfs_one_step` tries to compute a path that connects `start` with `goal`. If such a path is found, this path which is then returned in line 8.

   The details of the function `bfs_one_step` are discussed below.

5. Similarly, the function `bfs_one_step` in line 9 tries to find a path that connects `goal` with `start` by trying to expand states in `FrontierB`.

```
def bfs_one_step(Frontier, ParentA, ParentB, next_states):
    NewFrontier = set()
    for s in Frontier:
        for ns in next_states(s):
            if ns not in ParentA:
                NewFrontier |= { ns }
                ParentA[ns]  = s
                if ns in ParentB:
                    return combinePaths(ns, ParentA, ParentB)
    Frontier.clear()
    Frontier.update(NewFrontier)
```

Figure 2.16: The function `bfs_one_step`.

The function `bfs_one_step` is shown in Figure 2.16 on page 24.

1. The function `bfs_one_step` takes four arguments:

   (a) `Frontier` is the frontier of states that result from a breadth first search that originates in a state p where p is either equal to `start` or to `goal`.

(b) `ParentA` is a dictionary. For every state q that is discovered in the breadth first search originating in p, `ParentA`$[q]$ is a state that satisfies

$$q \in \texttt{next\_states}(\texttt{ParentA}[q]),$$

i.e. `ParentA`$[q]$ is the state that lead to the state q.

(c) `ParentB` is a dictionary that is similar to `ParentA` but that instead contains as keys the states from the opposite search, i.e. if $p =$ `start`, then `ParentB` contains the states as keys that have been found while searching from `goal` and if instead $p =$ `goal`, then `ParentB` contains the states as keys that have been found while searching from `start`.

(d) For every state s, we have that `next_states`(s) is the set of states that can be reached in one step from s.

The function `bfs_one_step` either returns a path or `None`. In the latter case, the function just the set `Frontier` to contain those states that can be reached in one step from the previous version of the set `Frontier`. Hence, the function `bfs_one_step` performs one iteration of breadth first search.

2. The set `NewFrontier` is initialized as the empty set in line 2.

3. Next, we iterate over all states `s` in the set `Frontier`.

4. For every state `ns` that is reachable from the state `s` in one step and that has not already been visited, we add `ns` to the set `NewFrontier` in line 6 and record its parent in line 7.

5. If the state `ns` has already been reached in the search starting from `goal` and hence has a parent node in `ParentB`, we have found a path from start to goal. Hence, We combine the path that leads from `start` to `ns` with the path leading from `goal` to `ns` in line 9.

6. It is important to note that the function `bfs_one_step` does not only return a path, it also has a side effect: If no path has been found, then the set `Frontier` is updated to contain those states that have been found in the current iteration.

Finally, Figure 2.17 on page 25 show the function `combinePaths` that takes a `state` that is reachable from both `start` and `goal`. It computes the path from `start` to the node `state` in line 2, the path from `goal` to the node `state` in line 3 and then combines these paths by first reversing the second path and appending it to the first path. When combining the paths, we have to take care to remove the last state from the first path `Path1`, since otherwise the node `state` would occur twice in the resulting path.

```
1 def combinePaths(state, ParentA, ParentB):
2     Path1 = path_to(state, ParentA)
3     Path2 = path_to(state, ParentB)
4     return Path1[:-1] + Path2[::-1] # Path2 is reversed
```

Figure 2.17: Combining two paths.

On my computer, bidirectional breadth first search solves the 3×3 sliding puzzle in 81 milliseconds and uses 4 megabytes. However, bidirectional breadth first search is still not able to solve the more interesting cases of the 4×4 sliding puzzle since the portion of the search space that needs to be computed is still too big to fit into memory.

2.6 Best First Search

Up to now, all the search algorithms we have discussed have been essentially blind. Given a state s and all of its neighbours, they had no idea which of the neighbours they should pick because they had no conception which of these neighbours might be more promising than the other neighbours. Search algorithms that know nothing about the distance of a state to the goal are called blind. Russell and Norvig [RN20] use the name uninformed search instead of blind search.

If a human tries to solve a search problem, she will usually develop an intuition that certain states are more favourable than other states because they seem to be closer to the solution. In order to formalise this procedure, we next define the notion of a search heuristic.

Definition 2 (Search Heuristic) Given a search problem

$$\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle,$$

a search heuristic or simply heuristic is a function

$$h : Q \to \mathbb{R}$$

that computes an approximation of the distance of a given state s to the state goal. The heuristic is admissible if it never overestimates the true distance, i.e. if the function

$$d : Q \to \mathbb{N}$$

computes the true distance from a state s to the goal, then we must have

$$h(s) \leq d(s) \quad \text{for all } s \in Q.$$

Hence, the heuristic is admissible iff it is optimistic: Although it never overestimates the distance to the goal, it is free to underestimate this distance.

Finally, the heuristic h is called consistent iff we have

$$h(\texttt{goal}) = 0 \quad \text{and} \quad h(s_1) \leq 1 + h(s_2) \quad \text{for all } s_2 \in \texttt{next\_states}(s_1). \qquad \diamond$$

Let us explain the idea behind the notion of consistency. First, if we are already at the goal, the heuristic should notice this fact and therefore we need to have $h(\texttt{goal}) = 0$. Secondly, assume we are at the state s_1 and s_2 is a neighbour of s_1, i.e. we have that

$$s_2 \in \texttt{next\_states}(s_1).$$

Now if our heuristic h assumes that the distance of s_2 from the goal is $h(s_2)$, then the distance of s_1 from the goal can be at most $1 + h(s_2)$ because starting from s_1 we can first go to s_2 in one step and then from s_2 to goal in $h(s_2)$ steps for a total of $1 + h(s_2)$ steps. Of course, it is possible that there exists a shorter path from s_1 leading to the goal than the one that visits s_2 first. Hence, we have the inequality

$$h(s_1) \leq 1 + h(s_2).$$

The Figure 2.18 on page 27 demonstrates this inequality.

Theorem 3 Every consistent heuristic is an admissible heuristic.

Proof: Assume that the heuristic h is consistent. Assume further that $s \in Q$ is some state such that there is a shortest path P from s to the goal. Assume this path has the form

$$P = [s_n, s_{n-1}, \cdots, s_1, s_0], \quad \text{where } s_n = s \text{ and } s_0 = \texttt{goal}.$$

Then the length of the path p is n and we have to show that $h(s) \leq n$. In order to prove this claim,

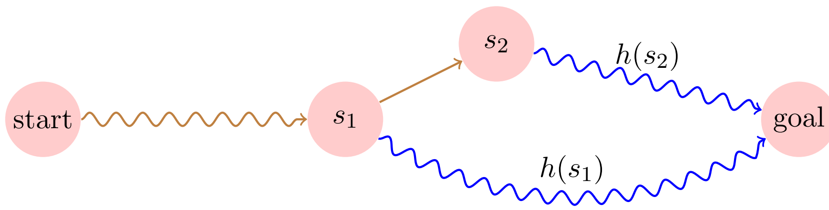


Figure 2.18: Explanation of the inequality $h(s_1) \leq 1 + h(s_2)$.

we show that we have

$$h(s_k) \leq k \quad \text{for all } k \in \{0, 1, \cdots, n\}.$$

This claim is shown by induction on k.

B.C.: $k = 0$.

We have $h(s_0) = h(\texttt{goal}) = 0 \leq 0$, because the fact that h is consistent implies $h(\texttt{goal}) = 0$.

I.S.: $k \mapsto k + 1$.

We have to show that $h(s_{k+1}) \leq k + 1$ holds. This is shown as follows:

$$\begin{array}{lcll} h(s_{k+1}) & \leq & 1 + h(s_k) & \text{because } s_k \in \texttt{next\_states}(s_{k+1}) \text{ and } h \text{ is consistent,} \\ & \leq & 1 + k & \text{because } h(s_k) \leq k \text{ by induction hypotheses.} \end{array}$$

We have shown $h(s_k) \leq k$ and since this also holds for $k = n$ we know that $h(s) = h(s_n) \leq n$. Since P is a shortest path we know that the state s has the distance n from the state `goal`. Hence the heuristic h underestimates this distance and is therefore admissible. □

It is natural to ask whether the last theorem can be reversed, i.e. whether every admissible heuristic is also consistent. The answer to this question is negative since there are some *contorted* heuristics that are admissible but that fail to be consistent. However, in practice it turns out that most admissible heuristics are also consistent. Therefore, when we construct consistent heuristics later, we will start with admissible heuristics, since these are easy to find. We will then have to check that these heuristics are also consistent.

Examples: In the following, we will discuss several heuristics for the sliding puzzle.

1. The simplest heuristic that is admissible is the function $h(s) := 0$. Since we have

   $$0 \leq 1 + 0,$$

   this heuristic is obviously consistent, but when we use this heuristic, we are back to blind search.

2. The next heuristic is the number of misplaced tiles heuristic. For a state s, this heuristic counts the number of tiles in s that are not in their final position, i.e. that are not in the same position as the corresponding tile in `goal`. For example, in Figure 2.4 on page 10 in the state depicted to the left, only the tile with the label 4 is in the same position as in the state depicted to the right. Hence, there are 7 misplaced tiles.

   As every misplaced tile must be moved at least once and every step in the sliding puzzle moves at most one tile, it is obvious that this heuristic is admissible. It is also consistent. First, the `goal` has no misplaced tiles, hence its heuristic is 0. Second, in every step of the sliding puzzle only one tile is moved. Therefore the number of misplaced tiles in two neighbouring states can differ by at most one and hence the inequality

$$h(s_1) \leq 1 + h(s_2)$$

is satisfied for any neighbouring states s_1 and s_2. Unfortunately, the number of misplaced tiles heuristic is very crude and therefore not particularly useful.

3. The Manhattan heuristic improves on the previous heuristic. For two points $\langle x_1, y_1\rangle, \langle x_2, y_2\rangle \in \mathbb{R}^2$ the Manhattan distance of these points is defined as

$$d_1\big(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle\big) := |x_2 - x_1| + |y_2 - y_1|.$$

The Manhattan distance is also called the L_1 norm of the difference vector $\langle x_2 - x_1, y_2 - y_1\rangle$. If we associate Cartesian coordinates with the tiles of the sliding puzzle such that the tile in the upper left corner has coordinates $\langle 1, 1\rangle$ and the coordinates of the tile in the lower right corner are $\langle 3, 3\rangle$, then the Manhattan distance of two positions measures how many steps it takes to move a tile from the first position to the second position if we are allowed to move the tile horizontally or vertically regardless of the fact that the intermediate positions might be blocked by other tiles. To compute the Manhattan heuristic for a state s with respect to the `goal`, we first define the function $\texttt{pos}(t, s)$ for all tiles $t \in \{1, \cdots, 8\}$ in a given state s as follows:

$$\texttt{pos}(t, s) = \langle \texttt{row}, \texttt{col}\rangle \overset{\text{def}}{\iff} s[\texttt{row}][\texttt{col}] = t,$$

i.e. given a state s, the expression $\texttt{pos}(t, s)$ computes the Cartesian coordinates of the tile t with respect to the state s. Then we can define the Manhattan heuristic h for the 3×3 puzzle as follows:

$$h(s) := \sum_{t=1}^{8} d_1\big(\texttt{pos}(t, s), \texttt{pos}(t, \texttt{goal})\big).$$

The Manhattan heuristic measures the number of moves that would be needed if we wanted to put every tile of s into its final position and if we were allowed to slide tiles over each other. Figure 2.19 on page 28 shows how the Manhattan distance can be computed. The code given in that figure works for a general $n \times n$ sliding puzzle. It takes two states `stateA` and `stateB` and computes the Manhattan distance between these states.

```
def manhattan(stateA, stateB):
    n = len(stateA)
    result = 0
    for rowA in range(n):
        for colA in range(n):
            tile = stateA[rowA][colA]
            if tile != 0:
                rowB, colB = find_tile(tile, stateB)
                result += abs(rowA - rowB) + abs(colA - colB)
    return result
```

Figure 2.19: The Manhattan distance between two states.

(a) First, the size `n` of the puzzle is computed by checking the number of rows of `stateA`.

(b) Next, the `for` loops iterates over all rows and columns of `stateA` that do not contain a blank tile. Remember that the blank tile is coded using the number 0. The tile at position $\langle$rowA, colA$\rangle$ in stateA is computed using the expression stateA[rowA][colA] and the corresponding position $\langle$rowB, colB$\rangle$ of this tile in state stateB is computed using the function find_tile.

(c) Finally, the Manhattan distance between the two positions $\langle$rowA, colA$\rangle$ and $\langle$rowB, colB$\rangle$ is added to the result.

The Manhattan heuristic is admissible. The reason is that if $s_2 \in$ next_states(s_1), then there can be only one tile t such that the position of t in s_1 is different from the position of t in s_2. Furthermore, this position differs by either one row or one column. Therefore,

$$|h(s_1) - h(s_2)| = 1$$

and hence $h(s_1) \leq 1 + h(s_2)$. $\Box$

Now we are ready to present best first search. This algorithm is derived from the stack based version of depth first search. However, instead of using a stack, the algorithm uses a priority queue. In this priority queue, the paths are ordered with respect to the estimated distance of the state at the end of the path from the goal. We always expand the path next that seems to be closest to the goal.

In *Python* the module heapq provides priority queues that are implemented as heaps. Technically, these heaps are just lists. In order to use them as priority queues, the entries of these lists will be pairs of the form (p, o), where p is the priority of the object o. Usually, the priorities are numbers and, contra-intuitively, high priorities correspond to **small** numbers, that is (p_1, o_1) has a higher priority than (p_2, o_2) if and only if $p_1 < p_2$. We need only two functions from the module heapq:

1. Given a heap H, the function heapq.heappop(H) returns and removes the pair from H that has the highest priority.

2. Given a heap H, the function heapq.heappush$\big(H, (p, o)\big)$ pushes the pair (p, o) onto the heap H. This method does not return a value. Instead, the heap H is changed in place.

```
def search(start, goal, next_states, heuristic):
    Visited   = set()
    PrioQueue = [ (heuristic(start, goal), [start]) ]
    while PrioQueue:
        _, Path = heapq.heappop(PrioQueue)
        state   = Path[-1]
        if state == goal:
            return Path
        if state in Visited:
            continue
        for ns in next_states(state):
            if ns not in Visited:
                prio = heuristic(ns, goal)
                heapq.heappush(PrioQueue, (prio, Path + [ns]))
        Visited.add(state)
```

Figure 2.20: The best first search algorithm.

The function `search` shown in Figure 2.20 on page 29 takes four parameters. The first three of these parameters are the same as in the previous search algorithms. The last parameter `heuristic` is a function that takes two states and then estimates the distance between these states. Later, when solving the sliding puzzle, we will use the Manhattan distance to serve as the parameter `heuristic`. The details of the implementation are as follows:

1. The Variable `Visited` collects all states that have been visited. A node n counts as visited when all of its neighbours have been inspected in line 11.

2. The variable `PrioQueue` serves as a priority queue. This priority queue is initialized as a list containing the pair $\langle d, [\texttt{start}]\rangle$, where d is the estimated distance of a path leading from `start` to `goal`. In `PrioQueue` we store the paths in pairs of the form

   $$\langle \texttt{estimate}, \texttt{Path}\rangle,$$

   where `Path` is a list of states starting from the node `start`. If the last node on this list is called `state`, then we have

   $$\texttt{estimate} = \texttt{heuristic}(\texttt{state}, \texttt{goal}),$$

   i.e. `estimate` is the estimated distance between this `state` and `goal` and hence an estimate of the number of steps needed to complete `Path` into a solution. This ensures, that the path whose last state is nearest to the `goal` is at the beginning of the set `PrioQueue`.

3. As long as `PrioQueue` is not empty, we take the `Path` from the top of this priority queue and remove it from the queue. This is then the path with the highest priority. If the state at the end of `Path` is named `state`, then `state` is, according to our heuristic, the state nearest to the `goal` among all states in the priority queue.

   The function `heappop(PrioQueue)` returns the smallest pair from `PrioQueue` and, furthermore, this pair is removed from `PrioQueue`.

4. If `state` is the `goal`, a solution has been found and is returned.

5. Next, we inspect all neighbouring states `ns` of `state` that have not already been visited. The paths leading to these nodes are pushed on the priority queue.

6. Finally, we mark `state` as visited by adding it to the set `Visited`.

Best first search solves the instance of the 3×3 puzzle shown in Figure 2.4 on page 10 in less than 3 millisecond and visits only 87 different states while solving the puzzle. However, the solution that is found takes 49 steps. While the length of this solution is not as ridiculous as the length of the solution found by depth first search, this length is far from being optimal. Best first search is able to solve the 4×4 puzzle shown in Figure 2.23 on page 34 in less than a second. It visits 8224 different states in order to find the solution. Unfortunately, the solution that is found has a length of 492 steps, while the optimal solution only needs 36 steps.

It should be noted that the fact that the Manhattan distance is a consistent heuristic is of no consequence for best first search. Only the A* algorithm, which is presented next, makes use of this fact.

2.7 A* Search

We have observed that best-first search can be remarkably efficient. However, most of the times the solution it provides is not optimal. This limitation arises because when best-first search considers a state s, it only takes into account the distance from s to the `goal` state. Crucially, it neglects the distance from the `start` state to s. In contrast, the A* search algorithm incorporates both the distance from `start` to s and the estimated distance from s to the `goal`. As a result, when the heuristic employed by the A* search algorithm is admissible, it is assured to find the shortest path.

Specifically, in the context of a given state s, the function $g(s)$ calculates the length of the path from `start` to s, and $h(s)$ provides a heuristic estimate of the distance from s to the `goal`. Consequently, the priority utilized by the A* search algorithm is expressed as

$$f(s) := g(s) + h(s).$$

The intricacies of the A* algorithm are detailed in Figure 2.21 on page 31. When we compare this implementation with the implementation of best first search shown in Figure 2.20 on page 29 we realize that these figure differ only in line 13 where the priority of a path is computed. With A* search, the priority of a path P ending in state s is the length of the path plus the estimated distance of the state s to the `goal`. The fact that we have to add 1 to `len(Path)` is due to the fact that `Path` only leads to `state` and we need the length of the path that leads to `ns`.

```
1  def search(start, goal, next_states, heuristic):
2      Visited   = set()
3      PrioQueue = [ (heuristic(start, goal), [start]) ]
4      while PrioQueue:
5          _, Path = heapq.heappop(PrioQueue)
6          state   = Path[-1]
7          if state in Visited:
8              continue
9          if state == goal:
10             return Path
11         for ns in next_states(state):
12             if ns not in Visited:
13                 prio = len(Path) + heuristic(ns, goal)
14                 heapq.heappush(PrioQueue, (prio, Path + [ns]))
15         Visited.add(state)
```

Figure 2.21: The A* search algorithm.

The A* search algorithm has been discovered by Hart, Nilsson, and Raphael and was first published in 1968 [HNR68]. However, there was a subtle bug in the first publication which was corrected in 1972 [HNR72].

When we run A* search on the 3×3 sliding puzzle, it takes about 0.1 seconds to solve the instance shown in Figure 2.4 on page 10 and visits 6614 states. Furthermore, the good news about A* search is that, provided the heuristic is admissible, the path which is found is optimal [HNR72].

Theorem 4 (Completeness and Optimality of A* Search)
If $\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$ is a search problem and $h : Q \to \mathbb{R}$ is a admissible heuristic for $\mathcal{P}$, then A* search is both complete and optimal, i.e. if there is a path from start to goal, then the search is successful and, furthermore, the solution that is computed is a shortest path leading from start to goal.

Proof: To simplify the notation of the proof we agree to use the following notation. If P is a path that is an element of PrioQueue and s is the last state of the path P, i.e. $P[-1] = s$, then we say that $s \in$ PrioQueue, although it really is the path P that is an element of PrioQueue. Furthermore, we denote the length of P with $g(s)$. Finally, we define

$$f(s) := g(s) + h(s) \quad \text{for all the states } s \in \texttt{PrioQueue}.$$

Therefore, $f(s)$ computes the priority of a state $s \in$ PrioQueue.

Next, the proof of our claim proceeds indirect. We assume that the path $P_1 = [s_0, s_1, \cdots, s_n]$ that is computed by A^* search is not the shortest path. Then there is a path $P_2 = [t_0, t_1, \cdots, t_m]$ leading from start to goal such that P_2 is shorter than P_1, i.e. we must have $m < n$. We claim that

$$f(t_i) \leq m \quad \text{for all } i \in \{0, \cdots, m\}.$$

The reasoning is as follows:

$$\begin{array}{lcll} f(t_i) & = & g(t_i) + h(t_i) & \\ & = & i + h(t_i) & \text{since } g(t_i) = \texttt{len}([t_0, \cdots, t_i]) = i \\ & \leq & i + (m - i) & \text{since } h(t_i) \leq \texttt{len}([t_i, \cdots, t_m]) = m - i \\ & = & m & \end{array}$$

However, for the path P_1 we know that

$$f(s_n) = g(s_n) + h(s_n) = n + 0 = n > m.$$

Since PrioQueue is a priority queue, we only remove the path P_1 from PrioQueue when all paths with a higher priority have already been removed and the corresponding end notes have been expanded. But as $n > m$ this means that all paths

$$[t_0],\ [t_0, t_1],\ \cdots,\ [t_0, \cdots, t_m]$$

have already been removed from PrioQueue before P_1 is removed. But then A^* search would have already found the shortest path from start to goal and hence the path P_1 would never be removed from PrioQueue. This shows that A^* search can't return a path that is not a shortest path. □

2.8 Bidirectional A* Search

When we refined breadth first search into bidirectional breadth first search were able to increase the power of breadth first search. We can try to do something similar with the A* algorithm and develop a bidirectional variant of this algorithm. Figure 2.22 on page 33 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.21 on page 31 as the algorithm for bidirectional search shown in Figure 2.15 on page 24 relates to breadth first search shown in Figure 2.6 on page 14. The only new idea is that we alternate between the A* search starting from start and the A* search starting from goal depending on the estimated total distance:

(a) As long as the search starting from start is more promising, we remove states from FrontierA.

(b) Once the total estimated distance of a path starting from goal is less than the best total estimated distance of a path starting from start, we switch and remove states from FrontierB.

There is one more twist, as the computation of the priority is a bit more involved. This is necessary to guarantee that the shortest path is computed.

When we run bidirectional A* search for the 3×3 sliding puzzle shown in Figure 2.4 on page 10, the program takes 150 milliseconds and uses $10,554$ states. I have also used bidirectional A^* search to solve the 4×4 sliding puzzle shown in Figure 2.23 on page 34. A solution of 36 steps was found

```python
def search(start, goal, next_states, heuristic):
    VisitedA   = {}
    VisitedB   = {}
    PrioQueueA = [ (heuristic(start, goal), [start]) ]
    PrioQueueB = [ (heuristic(goal, start), [goal ]) ]
    while PrioQueueA and PrioQueueB:
        a, PathA = PrioQueueA[0]
        b, PathB = PrioQueueB[0]
        if a <= b:
            heapq.heappop(PrioQueueA)
            for Result in search_os(PrioQueueA, PathA, goal,
                                    VisitedA, VisitedB, next_states, heuristic):
                return Result
        else:
            heapq.heappop(PrioQueueB)
            for Result in search_os(PrioQueueB, PathB, start,
                                    VisitedB, VisitedA, next_states, heuristic):
                return Result[::-1]

def search_os(PQ, Path, goal, VisitedA, VisitedB, next_states, heuristic):
    state = Path[-1]
    if state in VisitedA:
        return None
    if state in VisitedB:
        return Path[:-1] + VisitedB[state][::-1]
    for ns in next_states(state):
        if ns not in VisitedA:
            prio1 = len(Path) + heuristic(ns, goal)
            prio2 = 2 * len(Path)
            prio  = max(prio1, prio2)
            heapq.heappush(PQ, (prio, Path + [ns]))
    VisitedA[state] = Path
```

Figure 2.22: Bidirectional A* search.

in 2 seconds. A total 77,870 states had to be processed to compute this solution. This shows that, counter-intuitively, bidirectional A^* search uses more memory than unidirectional A^* search. Hence, in general, it not worth the trouble and we should stick with unidirectional A^* search.

```
start = ( (  0,  1,  2,  3 ),
          (  4,  5,  6,  8 ),
          ( 14,  7, 11, 10 ),
          (  9, 15, 12, 13 )
        )
goal  = ( (  0,  1,  2,  3 ),
          (  4,  5,  6,  7 ),
          (  8,  9, 10, 11 ),
          ( 12, 13, 14, 15 )
        )
```

Figure 2.23: A start state and a goal state for the 4×4 sliding puzzle.

2.9 Iterative Deepening A* Search

So far, we have combined A* search with bidirectional search and achieved good results. When memory space is too limited for bidirectional A* search to be possible, we can instead combine A* search with iterative deepening. The resulting search technique is known as iterative deepening A* search and is commonly abbreviated as IDA search. It has been invented by Richard Korf [Kor85]. Figure 2.24 on page 35 shows an implementation of IDA search. We proceed to discuss this program.

1. As in the A* search algorithm, the function `search` takes four parameters.
   (a) `start` is a state. This state represents the start state of the search problem.
   (b) `goal` is the goal state.
   (c) `next_states` is a function that takes a state s as a parameter and computes the set of all those states that can be reached from s in a single step.
   (d) `heuristic` is a function that takes two parameters s_1 and s_2, where s_1 and s_2 are states. The expression

   `heuristic`(s_1, s_2)

   computes an estimate of the distance between s_1 and s_2. In IDA search it is required that this estimate is optimistic, i.e. the `heuristic` has to be admissible.

2. The function `search` initializes `limit` to be an estimate of the distance between `start` and `goal`. As we assume that the function `heuristic` is optimistic, we know that there is no path from `start` to `goal` that is shorter than `limit`. Hence, we start our search by assuming that we might find a path that has a length of `limit`.

3. Next, we start a `while` loop. In this loop, we call the function `dl_search` (depth limited search) to compute a path from `start` to `goal` that has a length of at most `limit`. The function `dl_search` is described in detail below. When the function `dl_search` returns, there are two cases:
   (a) `dl_search` does find a path. In this case, this path is returned in the variable `Path` and the value of this variable is a list. This list is returned as the solution to the search problem.
   (b) `dl_search` is not able to find a path within the given `limit`. In this case, `dl_search` will not return a list representing a path, but instead it will return a number. This number

```
def search(start, goal, next_states, heuristic):
    limit = heuristic(start, goal)
    while True:
        Path = dl_search([start], goal, next_states, limit, heuristic)
        if isinstance(Path, list):
            return Path
        limit = Path

def dl_search(Path, goal, next_states, limit, heuristic):
    state    = Path[-1]
    distance = len(Path) - 1
    total    = distance + heuristic(state, goal)
    if total > limit:
        return total
    if state == goal:
        return Path
    smallest = float("Inf")  # infinity
    for ns in next_states(state):
        if ns not in Path:
            Solution = dl_search(Path+[ns], goal, next_states, limit, heuristic)
            if isinstance(Solution, list):
                return Solution
            smallest = min(smallest, Solution)
    return smallest
```

Figure 2.24: Iterative deepening A* search.

will specify the minimal length that any path leading from `start` to `goal` needs to have. This number is then used to update the `limit` which is used for the next invocation of `dl_search`.

Note that the fact that `dl_search` is able to compute this new `limit` is a significant enhancement over iterative deepening. While we had to test every single possible length in iterative deepening, now the fact that we can intelligently update the `limit` results in a considerable saving of computation time.

We proceed to discuss the function `dl_search`. This function takes 5 parameters, which we describe next.

1. `Path` is a list of states. This list starts with the state `start`. If `state` is the last state on this list, then `Path` represents a path leading from `start` to `state`.

2. `goal` is another state. The purpose of the recursive invocations of `dl_search` is to find a path from `state` to `goal`, where `state` is the last element of the list `Path`.

3. `next_states` is a function that takes a state s as input and computes the set of states that are reachable from s in one step.

4. `limit` is the upper bound for the length of the path from `start` to `goal`. If the function `dl_search` is not able to find a path from `start` to `goal` that has a length of at most `limit`, then the search is unsuccessful. In that case, instead of a path the function `dl_search` returns a new estimate for the distance between `start` and `goal`. Of course, this new estimate will be bigger than `limit`.

5. `heuristic` is a function taking two states as arguments. The invocation `heuristic`(s_1, s_2) computes an estimate of the distance between the states s_1 and s_2. It is assumed that this estimate is optimistic, i.e. the value returned by `heuristic`(s_1, s_2) is less than or equal to the true distance between s_1 and s_2.

We proceed to describe the implementation of the function `dl_search`

1. The variable `state` is assigned the last element of `Path`. Hence, `Path` connects `start` and `state`.

2. The length of the path connecting `start` and `state` is stored in `distance`.

3. Since `heuristic` is assumed to be optimistic, if we want to extend `Path`, then the best we can hope for is to find a path from `start` to `goal` that has a length of

   $$\texttt{distance} + \texttt{heuristic}(\texttt{state}, \texttt{goal}).$$

   This length is computed and saved in the variable `total`.

4. If `total` is bigger than `limit`, it is not possible to find a path from `start` to `goal` passing through `state` that has a length of at most `limit`. Hence, in this case we return `total` to communicate that the limit needs to be increased to have at least a value of `total`.

5. If we are lucky and `state` is equal to `goal`, the search is successful and `Path` is returned.

6. Otherwise, we iterate over all nodes `ns` reachable from `state` that have not already been visited by `Path`. If `ns` is a node of this kind, we extend the `Path` so that this node is visited next. The resulting path has the form

   `Path + [ns]`.

   Next, we recursively start a new search starting from the node `ns`. If this search is successful, the resulting path is returned. Otherwise, the search returns the minimum distance that is needed to reach the state `goal` from the state `start` on a path via the state `ns`. If this distance is smaller than the distance `smallest` that is returned from visiting previous neighbouring nodes, the variable `smallest` is updated accordingly. This way, if the `for` loop is not able to return a path leading to `goal`, the variable `smallest` contains a lower bound for the distance that is needed to reach `goal` by a path that extends the given `Path`.

   Note: At this point, a natural question is to ask whether the `for` loop should collect all paths leading to `goal` and then only return the path that is shortest. However, this is not necessary: Every time the function `dl_search` is invoked it is already guaranteed that there is no path that is shorter than the parameter `limit`. Therefore, if `dl_search` is able to find a path that has a length of at most `limit`, this path is known to be optimal.

Iterative deepening A* is a complete search algorithm that does find an optimal path, provided that the employed heuristic is optimistic. On the instance of the 3×3 sliding puzzle shown on Figure 2.4 on page 10, this algorithm takes about 1.2 seconds to solve the puzzle. Only about 170 kilobytes of memory are necessary for this search. For the 4×4 sliding puzzle shown in Figure 2.23, the algorithm takes about 1.6 seconds and uses 184 kilobytes. Although this is more than the time needed by bidirectional A* search, the good news is that the IDA* algorithm does not need much memory since basically only the path discovered so far is stored in memory. Hence, IDA* is a viable alternative if the available memory is not sufficient to support the bidirectional A* algorithm.

Exercise 4: The eight queens puzzle is the problem of placing eight chess queens on a chessboard so that no two queens can attack each other. In chess, a queen can attack by moving horizontally, vertically, or diagonally.

(a) Reformulate the eight queens puzzle as a search problem.

(b) Compute an upper bound for the number of states.

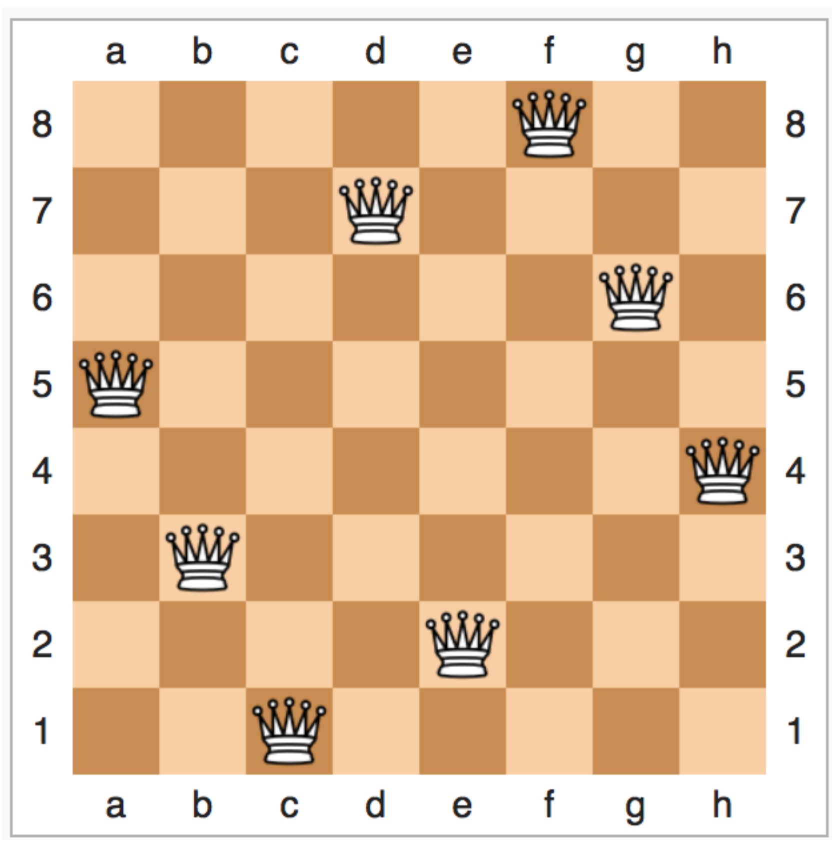
(c) Which of the algorithms we have discusses are suitable to solve this problem?

(d) Compute all 92 solutions of the eight queens puzzle.

Hint: It is easiest to encode states as lists. For example, the solution of the eight queens puzzle that is shown in Figure 2.25 would be represented as the list

$$[6, 4, 7, 1, 8, 2, 5, 3]$$

because the queen in the first row is positioned in column 6, the queen in the second row is positioned in column 4, and so on. The start state would then be the empty list and given a state L, all states from the set `nextState`(L) would be lists of the form $L + [c]$. If $\#L = k$, then the state $l + [c]$ has $k + 1$ queens, where the queen in row $k + 1$ has been placed in column c. A frame for solving this problem is available at Artificial-Intelligence/blob/master/Python/1 Search/14-N-Queens.ipynb.



,

Figure 2.25: A solution of the eight queens puzzle.

Exercise 5: The founder of Taoism, the Chinese philosopher Laozi once said:

> *"A journey of a thousand miles begins but with a single step"*.

This proverb is the foundation of taoistic search. The idea is, instead of trying to reach the goal directly, we rather define some intermediate states which are easier to reach than the goal state and that are nearer to the goal than the start state. To make this idea more precise, consider the following instance of the 15-puzzle, where the states `Start` and `Goal` are given as follows:

```
Start :=  +---+---+---+---+    Goal :=  +---+---+---+---+
          |   |15 |14 | 8 |             |   | 1 | 2 | 3 |
          +---+---+---+---+             +---+---+---+---+
          |12 |10 |11 |13 |             | 4 | 5 | 6 | 7 |
          +---+---+---+---+             +---+---+---+---+
          | 9 | 6 | 2 | 5 |             | 8 | 9 |10 |11 |
          +---+---+---+---+             +---+---+---+---+
          | 1 | 3 | 4 | 7 |             |12 |13 |14 |15 |
          +---+---+---+---+             +---+---+---+---+
```

In order to solve this instance of the 15-puzzle, we could try to first move the tiles numbered with 14 and 15 into the lower right corner. The resulting state would have the following form:

```
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * |14 |15 |
+---+---+---+---+
```

Here, the character "`*`" is used as a wildcard character, i.e. we do not care about the actual character in the state, for we only want to ensure that the first two tiles are positioned correctly. Once we have reached a state specified by the pattern given above, we could then proceed to reach a state that is described by the following pattern:

```
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
|12 |13 |14 |15 |
+---+---+---+---+
```

We have now solved the bottom line of the puzzle. In a similar way, we can try to solve the line above the bottom line. After that, the next step would be to reach a goal of the form

```
+---+---+---+---+
| * | * | 2 | 3 |
+---+---+---+---+
| * | * | 6 | 7 |
+---+---+---+---+
| 8 | 9 |10 |11 |
+---+---+---+---+
|12 |13 |14 |15 |
+---+---+---+---+
```

The final step would then solve the puzzle. I have prepared a framework for taoistic search. The file

`Python/1 Search/15-Taoistic-Search.ipynb`

from my github repository at `https://github.com/karlstroetmann/Artificial-Intelligence` contains a framework to solve the sliding puzzle using taoistic search where some functions are left unimplemented. Your task is to implement the missing functions in this file and thereby solve the puzzle. ◇

Chapter 3

Solving Constraint Satisfaction Problems

In this chapter, we delve into a variety of algorithms designed for solving constraint satisfaction problems. In a constraint satisfaction problem, we are presented with a set of formulas, and our objective is to find values that can be assigned to the variables in these formulas, ensuring all formulas evaluate to true. Constraint satisfaction problems represent a more refined version of the search problems addressed in the previous chapter. Unlike search problems, where states are abstract and lack exploitable structure for guiding the search, constraint satisfaction problems feature structured states in the form of variable assignments. This structure can be leveraged to direct the search process more effectively.

This chapter is organized as follows:

(a) The initial section introduces the concept of a constraint satisfaction problem. To elucidate this concept, we explore two examples: map colouring and the eight queens puzzle. Subsequently, we examine various applications of constraint satisfaction problems.

(b) The most basic algorithm for addressing a constraint satisfaction problem is brute force search. The principle of *brute force search* involves examining every possible variable assignment.

(c) Often, the search space is so vast that enumerating all variable assignments becomes impractical. Backtracking search enhances brute force search by intertwining the generation of variable assignments with the evaluation of constraints, significantly boosting the efficiency of the search.

(d) Backtracking search can be further refined through the integration of constraint propagation and the application of the most restricted variable heuristic.

(e) Additionally, verifying the consistency of values assigned to different variables can substantially reduce the search space.

(f) Local search presents an alternative methodology for solving constraint satisfaction problems, particularly beneficial for large but uncomplicated problems.

(g) Lastly, we discuss the `Z3` theorem prover, an industrial-grade constraint solver. Here, a *constraint solver* is defined as software that accepts a *constraint satisfaction problem* as input and produces a *solution* for the problem.

Upon concluding our exploration of constraint satisfaction problems, we will have developed a constraint solver capable of resolving the most challenging Sudoku puzzles in mere seconds.

3.1 Formal Definition of Constraint Satisfaction Problems

Formally, a constraint satisfaction problem is defined as a triple:

$$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$$

where

(a) `Vars` represents a set of strings, functioning as variables.

(b) `Values` denotes a set of values that can be assigned to the variables in `Vars`.

(c) `Constraints` is a collection of formulas derived from first order logic, each termed a constraint of $\mathcal{P}$. To evaluate these formulas, an interpretation of the function and predicate symbols appearing in these constraints is essential. To avoid excessive formalization, we presume these interpretations are implicitly understood. In the provided examples, these interpretations will be given through functions implemented in *Python*.

In subsequent sections, the abbreviation CSP refers to constraint satisfaction problem. Given a CSP

$$\mathcal{P} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle,$$

a variable assignment for $\mathcal{P}$ is a function

$$A : \texttt{Vars} \to \texttt{Values}$$

that maps variables to values. A variable assignment A is a solution to $\mathcal{P}$ if, under A, all constraints are fulfilled, that is:

$$\texttt{eval}(f, A) = \texttt{true} \quad \text{for every } f \in \texttt{Constraints}.$$

Moreover, a partial variable assignment B for $\mathcal{P}$ is a function

$$B : \texttt{Vars} \to \texttt{Values} \cup \{\Omega\}, \text{ with } \Omega \text{ symbolizing the undefined value.}$$

Therefore, a partial variable assignment does not assign values to all variables, but only to a subset of `Vars`. The domain $\texttt{dom}(B)$ of a partial variable assignment B is defined as the set of variables assigned a value different from Ω, namely:

$$\texttt{dom}(B) := \bigl\{ x \in \texttt{Vars} \mid B(x) \neq \Omega \bigr\}.$$

The concepts delineated thus far will be explained through three examples.

3.1.1 Example: Map Colouring

In map colouring a map showing different states and their borders is given and the task is to colour the different states such that no two states that have a common border share the same colour. Figure 3.1 on page 42 shows a map of Australia. There are seven different states in Australia:

1. Western Australia, abbreviated as WA,
2. Northern Territory, abbreviated as NT,
3. South Australia, abbreviated as SA,
4. Queensland, abbreviated as Q,
5. New South Wales, abbreviated as NSW,
6. Victoria, abbreviated as V, and

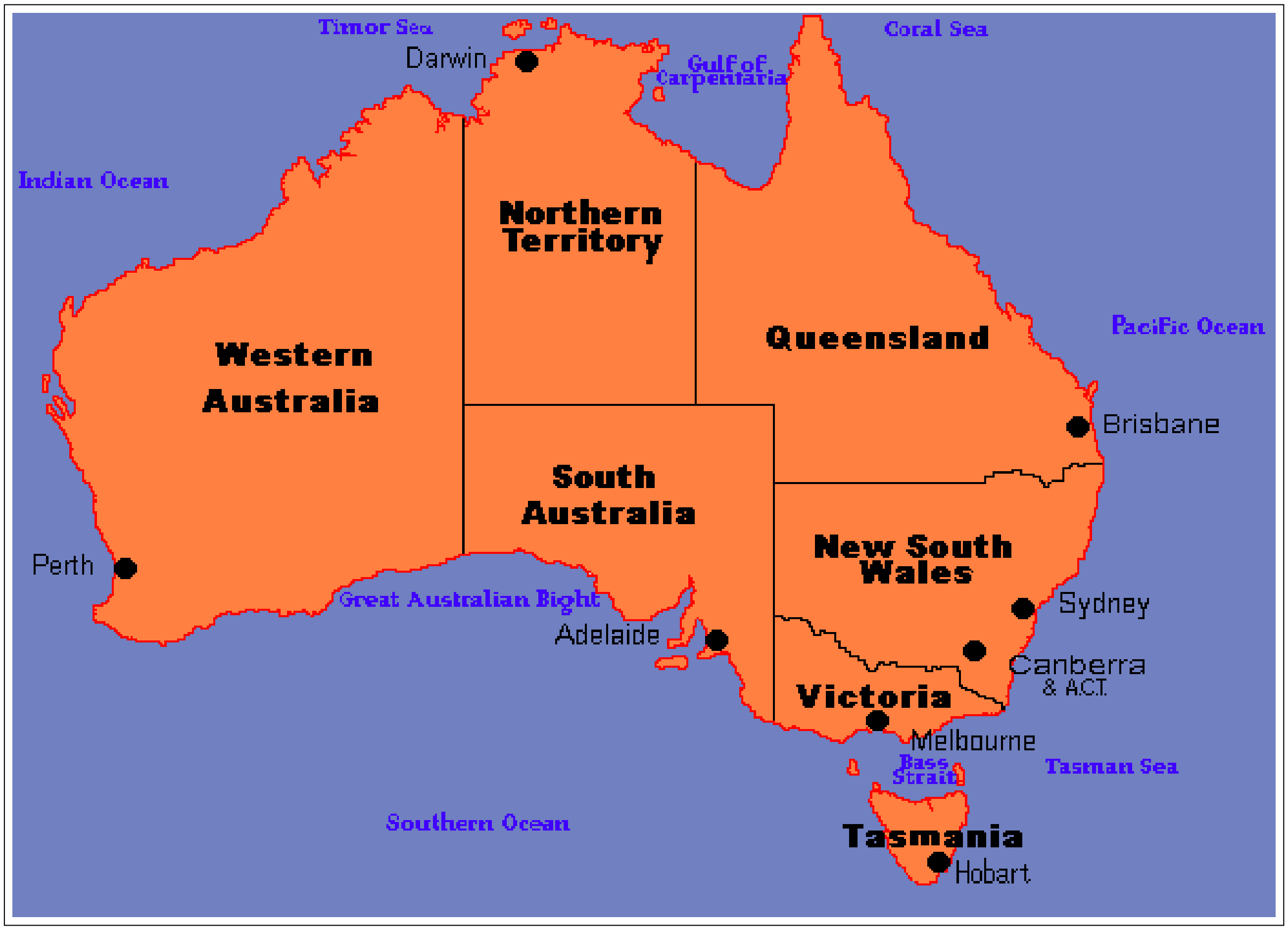


Figure 3.1: A map of Australia.

7. Tasmania, abbreviated as T.

Figure 3.1 would certainly look better if different states that share a common border had been coloured with different colours. For the purpose of this example let us assume that we only have the three colours red, green, and blue available. The task is then to colour the different states in a way that no two neighbouring states share the same colour. This task can be formalized as a constraint satisfaction problem. To this end we define:

1. $\texttt{Vars} := \{\mathrm{WA}, \mathrm{NT}, \mathrm{SA}, \mathrm{Q}, \mathrm{NSW}, \mathrm{V}, \mathrm{T}\}$,
2. $\texttt{Values} := \{\texttt{red}, \texttt{green}, \texttt{blue}\}$,
3. $\texttt{Constraints} :=$

$$\{\texttt{WA} \neq \texttt{NT},\ \texttt{WA} \neq \texttt{SA},\ \texttt{SA} \neq \texttt{Q},\ \texttt{NT} \neq \texttt{Q},\ \texttt{SA} \neq \texttt{Q},\ \texttt{SA} \neq \texttt{NSW},\ \texttt{SA} \neq \texttt{V},\ \texttt{Q} \neq \texttt{NSW},\ \texttt{NSW} \neq \texttt{V}\}.$$

Then $\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$ is a constraint satisfaction problem. If we define the assignment A such that

(a) $A(\mathrm{WA}) = \texttt{blue}$,

(b) $A(\mathrm{NT}) = \texttt{red}$,

(c) $A(\mathrm{SA}) = \texttt{green}$,

(d) $A(\mathrm{Q}) = \texttt{blue}$,

(e) $A(\mathrm{NSW}) = \texttt{red}$,

(f) $A(\mathrm{V}) = \texttt{blue}$,

(g) $A(\mathrm{T}) = \texttt{red}$,

then it is straightforward to check that this assignment is indeed a solution to the constraint satisfaction problem $\mathcal{P}$.

3.1.2 Example: The Eight Queens Puzzle

The eight queens puzzle asks to put 8 queens on a chessboard such that no queen can attack another queen. In chess, a queen can attack all pieces that are either in the same row, the same column, or the same diagonal. If we want to put 8 queens on a chessboard such that no two queens can attack each other, we have to put exactly one queen in every row: If we would put more than one queen in a row, the queens in that row could attack each other. If we would leave a row empty, then, given that the other rows contain at most one queen, there would be less than 8 queens on the board. Therefore, in order to model the eight queens problem as a constraint satisfaction problem, we will use the following set of variables:

$$\texttt{Vars} := \{\texttt{V}_1, \texttt{V}_2, \texttt{V}_3, \texttt{V}_4, \texttt{V}_5, \texttt{V}_6, \texttt{V}_7, \texttt{V}_8\},$$

where for $i \in \{1, \cdots, 8\}$ the variable $\texttt{V}_i$ specifies the column of the queen that is placed in row i. As the column numbers run from 1 up to 8, we define the set `Values` as

$$\texttt{Values} := \{1, 2, 3, 4, 5, 6, 7, 8\}.$$

Next, let us define the constraints. There are two different types of constraints.

1. We need constraints that express that no two queens that are positioned in different rows share the same column. To capture these constraints, we define

   $$\texttt{DifferentCols} := \big\{\texttt{V}_i \neq \texttt{V}_j \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i\big\}.$$

   Here the condition $j < i$ ensures that, for example, while we have the constraint $\texttt{V}_2 \neq \texttt{V}_1$ we do not also have the constraint $\texttt{V}_1 \neq \texttt{V}_2$, as the latter constraint would be redundant if the former constraint had already been established.

2. We need constraints that express that no two queens positioned in different rows share the same diagonal. The queens in row i and row j share the same diagonal iff the equation

   $$|i - j| = |V_i - V_j|$$

   holds. The expression $|i - j|$ is the absolute value of the difference of the rows of the queens in row i and row j, while the expression $|V_i - V_j|$ is the absolute value of the difference of the columns of these queens. To capture these constraints, we define

   $$\texttt{DifferentDiags} := \big\{|i - j| \neq |\texttt{V}_i - \texttt{V}_j| \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i\big\}.$$

   For a fixed pair of values $\langle j, V_j \rangle$ the equations

   $$V_i = V_j - j + i \quad \text{and} \quad V_i = V_j + j - i$$

   are the linear equations for the straight lines with slope 1 and -1 that pass through $\langle j, V_i \rangle$.

Then, the set of constraints is defined as

$$\texttt{Constraints} := \texttt{DifferentCols} \cup \texttt{DifferentDiags}$$

and the eight queens problem can be stated as the constraint satisfaction problem

$$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle.$$

If we define the assignment A such that

$$A(V_1) := 4,\ A(V_2) := 7,\ A(V_3) := 5,\ A(V_4) := 2,\ A(V_5) := 6,\ A(V_6) := 1,$$
$$A(V_7) := 3,\ A(V_8) := 8,$$

then it is easy to see that this assignment is a solution of the eight queens problem. This solution is shown in Figure 3.2 on page 44. In this figure, we have numbered the rows from bottom to top, i.e. the topmost row is row number 8 and therefore the column of the queen in the first row is determined by the variable V_8.

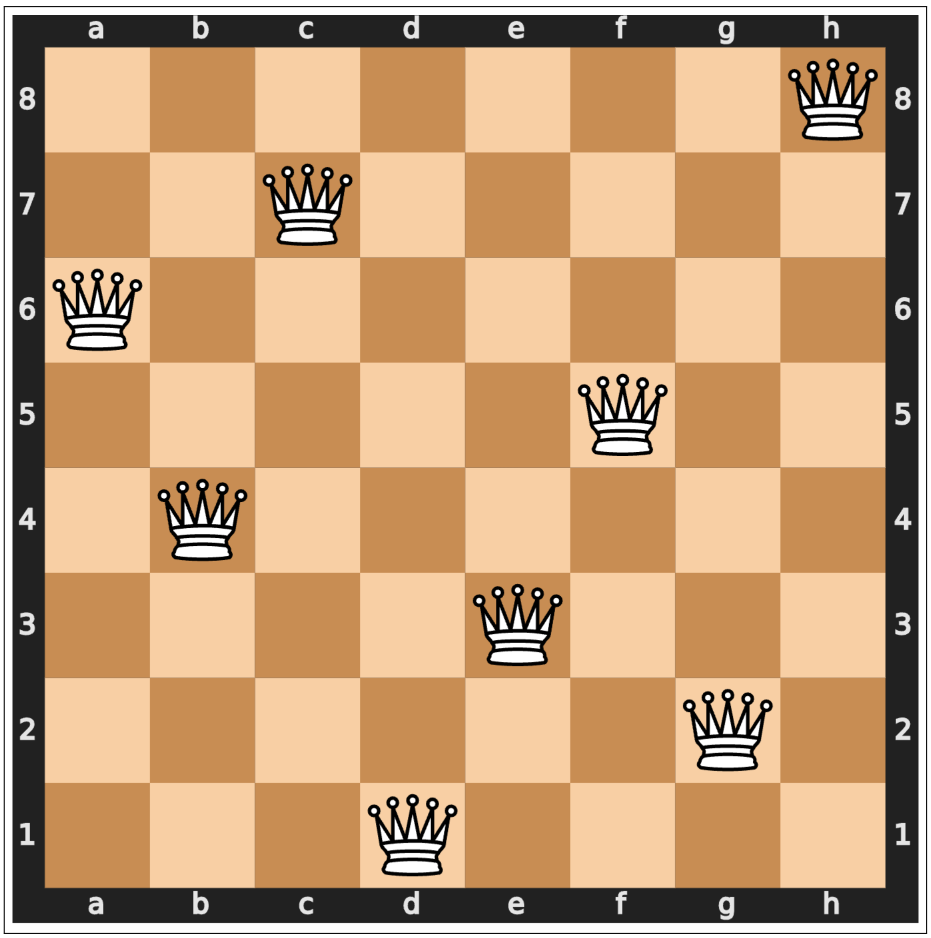


Figure 3.2: A solution of the eight queens puzzle.

Later, when we develop algorithms to solve CSPs, we will represent variable assignments and partial variable assignments as *Python* dictionaries. For example, A would then be represented as the dictionary

$$A := \{\mathrm{V}_1 : 4, \mathrm{V}_2 : 7, \mathrm{V}_3 : 5, \mathrm{V}_4 : 2, \mathrm{V}_5 : 6, \mathrm{V}_6 : 1, \mathrm{V}_7 : 3, \mathrm{V}_8 : 8\}.$$

If we define

$$B := \{\mathrm{V}_1 : 4, \mathrm{V}_2 : 7, \mathrm{V}_3 : 3\},$$

then B is a partial variable assignment and $\texttt{dom}(B) = \{V_1, V_2, V_3\}$. This partial variable assignment is shown in Figure 3.3 on page 45. Note that the bottom-most row is the row number 1.

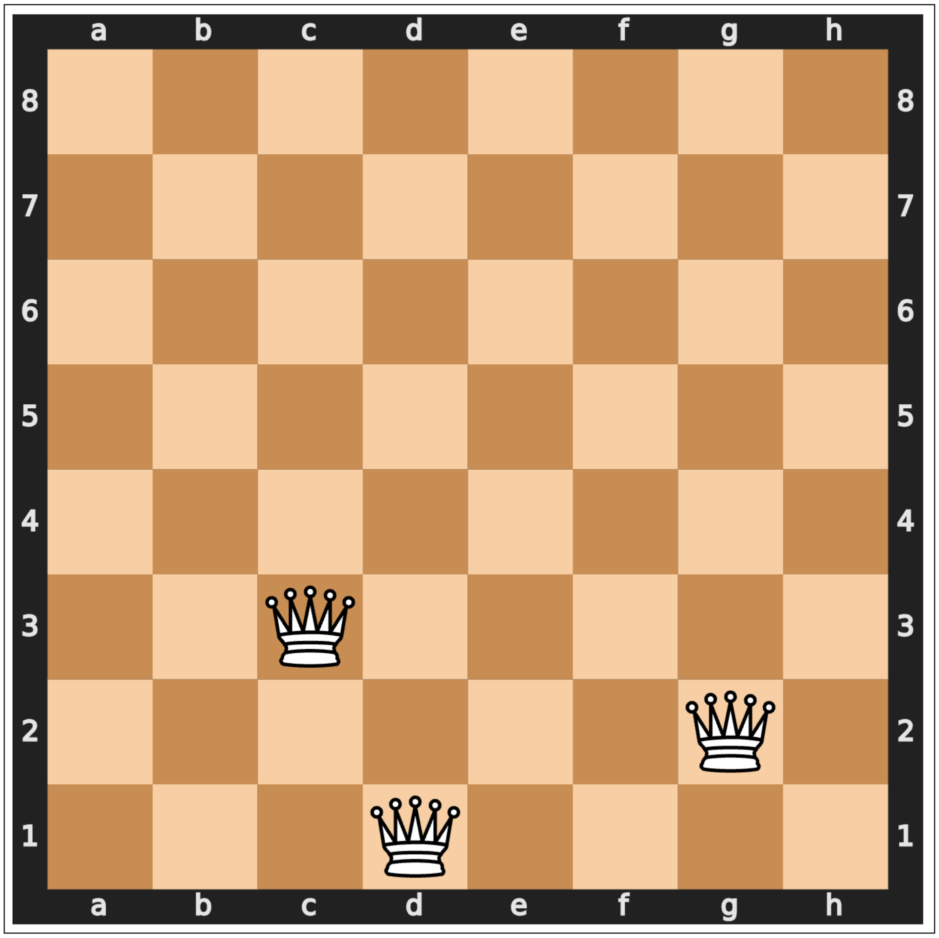


Figure 3.3: A partial solution of the eight queens puzzle.

```python
def create_csp(n):
    S              = range(1, n+1)
    Variables      = { f'V{i}' for i in S }
    Values         = set(S)
    DifferentCols  = { f'V{i} != V{j}' for i in S
                                       for j in S
                                       if  i < j
                     }
    DifferentDiags = { f'abs(V{j} - V{i}) != {j - i}' for i in S
                                                      for j in S
                                                      if  i < j
                     }
    return Variables, Values, DifferentCols | DifferentDiags
```

Figure 3.4: The n queens problem formulated as a CSP.

Figure 3.4 on page 46 shows a *Python* program that can be used to create a CSP that encodes the eight queens puzzle. The code shown in this figure is more general than necessary. Given a natural number n, the function call `create_csp`(n) creates a constraint satisfaction problem $\mathcal{P}$ that generalizes the eight queens problem to the problem of placing n queens on a board of size n times n such that no queen can capture another queen. The fact that the n-queen problem is parameterized by the number of queens n gives us the ability to check how the running time of the algorithms for solving CSPs scales with the size of the problem.

The beauty of constraint programming is the fact that we will be able to develop a so called constraint solver that takes as input a CSP like the one produced by the program shown in Figure 3.4 and that is capable of computing a solution automatically. In effect, this enables us to use declarative programming: Instead of developing an algorithm that solves a given problem we confine ourselves to specifying the problem precisely and then let a general purpose problem solver do the job of computing the solution. This approach of declarative programming was one of the main ideas incorporated in the programming language Prolog. While Prolog could not live up to its promises as a viable general purpose programming language, constraint programming has proved to be very useful in a number of domains.

3.1.3 Example: The Zebra Problem

The following puzzle is known as the Zebra Puzzle and was featured in the magazine *Life International* on December 17, 1962. It presents a series of clues pertaining to the occupants of five distinct houses, challenging the solver to deduce specific details about them. The puzzle is structured as follows:

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.

6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.
16. Each of the five houses is painted a unique color.
17. The residents of the five houses each have a distinct nationality.
18. A different pet is kept in each house.
19. The beverages consumed in the various houses are all unique.
20. A distinct brand of cigarettes is smoked in each house.

The objective of the Zebra Puzzle is to answer the following two questions:

1. Who drinks water?
2. Who owns the zebra?

Next, we formulate the zebra puzzle as a constraint satisfaction problem. To this end, we first define the set of variables:

```
Nations = { 'English', 'Spanish', 'Ukrainian', 'Norwegian', 'Japanese' }
Drinks  = { 'Coffee' , 'Tea', 'Milk', 'OrangeJuice', 'Water' }
Pets    = { 'Dog', 'Snails', 'Horse', 'Fox', 'Zebra' }
Brands  = { 'LuckyStrike', 'Parliaments', 'Kools', 'Chesterfields', 'OldGold' }
Colours = { 'Red', 'Green', 'Ivory', 'Yellow', 'Blue' }
Variables = Nations | Drinks | Pets | Brands | Colours
```

Then, we define the set of values as:

```
Values = { 1, 2, 3, 4, 5 }
```

The interpretation of the variables and their values should be obvious: For example, if the variable `English` has the value 1, the this would imply that the Englishman lives in the first house.

In order to write down the last 5 constraints we implement the auxiliary function `allDifferent`(V) which takes a set of variables V as input and returns a set of formulas expressing that the values of the variables in V are all different.

```
def allDifferent(Variables: set[str]) -> set[str]:
    return { f'{x} != {y}' for x in Variables
                           for y in Variables
                           if x < y
           }
```

For example, using the function `allDifferent` we can express the fact that all five houses are painted in a different color via the formula

`allDifferent(Nations)`.

Using the function `allDifferent`, we can define the set of all constraints as follows:

```
Constraints = { 'English       == Red',
                'Spanish       == Dog',
                'Coffee        == Green',
                'Ukrainian     == Tea',
                'Green         == Ivory + 1',
                'OldGold       == Snails',
                'Kools         == Yellow',
                'Milk          == 3',
                'Norwegian     == 1',
                'abs(Chesterfields - Fox) == 1',
                'abs(Kools - Horse) == 1',
                'LuckyStrike   == OrangeJuice',
                'Japanese      == Parliaments',
                'abs(Norwegian - Blue) == 1'
              }
Constraints |= allDifferent(Nations)
Constraints |= allDifferent(Drinks)
Constraints |= allDifferent(Pets)
Constraints |= allDifferent(Brands)
Constraints |= allDifferent(Colours)
```

We will soon develop a solver that is able to solve the resulting constraint satisfaction problem.

Exercise 6: In an oncology ward, five patients are lying in adjacent rooms. Except for one of the patients, each has smoked exactly one brand of cigarette. The patient who did not smoke cigarettes smoked a pipe. Each patient drives exactly one car and is diagnosed with exactly one type of cancer. Additionally, we have the following information:

1. In the room next to Michael, Camel is being smoked.
2. The Trabant driver smokes Harvest 23 and is in the room next to the tongue cancer patient.
3. Rolf is in the last room and has laryngeal cancer.
4. The West smoker is in the first room.
5. The Mazda driver has tongue cancer and is next to the Trabant driver.

6. The Nissan driver is next to the tongue cancer patient.
7. Rudolf is desparately begging for euthanasia and his room is between the room of the Camel smoker and the room of the Trabant driver.
8. Tomorrow is the last birthday of the Seat driver.
9. The Luckies smoker is next to the patient with lung cancer.
10. The Camel smoker is next to the patient with intestinal cancer.
11. The Nissan driver is next to the Mazda driver.
12. The Mercedes driver smokes a pipe and is next to the Camel smoker.
13. Jens is next to the Luckies smoker.
14. Yesterday, the patient with testicular cancer flushed his balls down the toilet.

Given this information, the task is to answer the following questions:

1. What does the intestinal cancer patient smoke?
2. What car does Kurt drive?

Your task is to formulate this puzzle as a constraint satisfaction problem. ◇

3.1.4 Applications

Besides the toy problems discussed so far, there are a number of industrial applications of constraint satisfaction problems. The most important application seem to be variants of scheduling problems. A simple example of a scheduling problem is the problem of generating a timetable for a school. A school has various teachers, each of which can teach some subjects but not others. Furthermore, there are a number of classes that must be taught in different subjects. The problem is then to assign teachers to classes and to create a timetable. A special case of scheduling problems is crew scheduling. For example, airlines have to solve a crew scheduling problem in order to efficiently assign crews of pilots and crews of stewards to their aircraft. Stewards and pilots work in different crews as they have different required resting times.

3.2 Brute Force Search

The most straightforward algorithm to solve a CSP is to test all possible combinations of assigning values to variables. This approach is known as brute-force search. If there are n different values that can be assigned to k variables, this approach amounts to checking at most n^k different variable assignments. For example, for the eight queens problem there are 8 variables and 8 possible values and hence there are at most

$$8^8 = 2^{24} = 16,777,216$$

different assignments that need to be tested. Given the clock rate of modern computers, checking a million assignments per second is plausible. Hence, this approach is able to solve the eight queens problem in about 30 seconds. An implementation of brute force search is shown in Figure 3.5 on page 50.

```python
def solve(P):
    return brute_force_search({}, P)

def brute_force_search(Assignment, csp):
    Variables, Values, Constraints = csp
    if len(Assignment) == len(Variables): # all variables have been assigned
        if check_all_constraints(Assignment, Constraints):
            return Assignment
        else:
            return None
    var = arb(Variables - set(Assignment.keys()))
    for value in Values:
        NewAss = Assignment.copy()
        NewAss[var] = value
        result = brute_force_search(NewAss, csp)
        if result != None:
            return result
    return None
```

Figure 3.5: Solving a Csp via brute force search.

The function `solve` takes a constraint satisfaction problem P as its input. This `CSP` is given as a triple of the form

$$P = (\texttt{Variables}, \texttt{Values}, \texttt{Constraints}).$$

The sole purpose of the function `solve` is to call the function `brute_force_search`, which needs an additional argument. This argument is a partial variable assignment that is initially empty. Every recursive iteration of the function `brute_force_search` assigns one additional variable.

1. `Assignment` is a partial variable assignment. Initially, this assignment will be the empty dictionary. Every recursive call of `brute_force_search` adds the assignment of one variable to the given assignment.

2. `csp` is a triple of the form

   $$\texttt{csp} = (\texttt{Variables}, \texttt{Values}, \texttt{Constraints}).$$

   Here, `Constraints` is a set of Boolean expressions that are given as strings. These strings have to follow the syntax of *Python* so that they can be evaluated using the *Python* function `eval`.

The implementation of `brute_force_search` works as follows:

1. If all variables have been assigned a value, the dictionary `Assignment` will have the same number of entries as the set `Variables` has elements. Hence, in that case `Assignment` is a complete assignment of all variables and we now have to test whether all constraints are satisfied. This is done using the auxiliary function `check_all_constraints` that is shown in Figure 3.6 on page 51. If the current `Assignment` does indeed satisfy all constraints, it is a solution to the given Csp and is therefore returned.

   If, instead, some constraint is violated, then `brute_force_search` returns the value `None`.

2. If the assignment is not yet complete, we arbitrarily pick a variable `var` from the set of those `Variables` that still have no value assigned. Then, for every possible `value` in the set `Values`, we extend the current partial `Assignment` to a new assignment `NewAss` that satisfies

   `NewAss[var] = value`.

   Next, the algorithm recursively tries to find a solution for this new partial assignment. If this recursive call succeeds, the solution it has computed is returned. Otherwise, the next value for the given variable `var` is tried.

3. If none of the values work for `var`, the function returns `None`.

```python
def check_all_constraints(Assignment, Constraints):
    A = Assignment.copy()
    return all(eval(f, A) for f in Constraints)
```

Figure 3.6: Auxiliary functions for brute force search.

The function `check_all_constraints` takes a complete variable `Assignment` as its first input. The second input is the set `Constraints` which is a set of *Python* expressions. For all expressions `f` from the set `Constraints`, the function `check_all_constraints` checks whether `f` yields `True` under the given variable assignment. This check is done using the function `eval`, which is a predefined function. This function takes two arguments:

(a) The first argument is a *Python* expression `f`.

(b) The second argument is a variable assignment `A`, that is represented as a dictionary.

The function `eval` evaluates the expression `f`. In order to do this, any variables occurring in `f` are assigned values according to the variable assignment `A`. As a side effect, the function `eval` changes the dictionary `A` that is used as its second argument. This is the reason we have to make a copy of the `Assignment` that is given as the first argument of the function `check_all_constraints`.

When I tested the program discussed above with the eight queens problem, it took about 30 seconds to compute a solution. In contrast, the seven queens problem took about 1.7 second. As we have

$$\frac{8^8}{7^7} \approx 20 \quad \text{and} \quad 30/1.7 \approx 18$$

this shows that the computation time does indeed roughly grow with the number of possible assignments that have to be checked. However, the correspondence is not exact. The reason is that we stop our search as soon as a solution is found. If we are lucky and the given CSP is easy to solve, this might happen when we have checked only a small portion of the set of all possible assignments.

3.3 Backtracking Search

For the n queens problem the number of possible variable assignments growth as fast as n^n. This growth is super-exponential and this is what usually happens when we scale a CSP up. The reason is that the number of all variable assignments is given as

$$\mathrm{card}(\texttt{Values})^{\mathrm{card}(\texttt{Vars})},$$

where for a set M, the expression $\mathrm{card}(M)$ returns the number of elements of M. For this reason, brute force search is only viable for small problems. One approach to solve a CSP that is both conceptually simple and at least more efficient than brute force search is backtracking. The idea is to try to evaluate constraints as soon as possible: If C is a constraint and B is a partial assignment such that all the variables occurring in C have already been assigned a value in B and the evaluation of C fails, then there is no point in trying to complete the variable assignment B. Hence, in backtracking we evaluate a constraint C as soon as all of its variables have been assigned a value. If C is not valid, we discard the current partial variable assignment. This approach can result in huge time savings when compared to the baseline of brute force search.

Figure 3.7 on page 52 shows a simple CSP solver that employs the backtracking strategy. We discuss this program next. The function `solve` takes a constraint satisfaction problem `P` as input and tries to find a solution.

```
def solve(P):
    Variables, Values, Constraints = P
    csp = (Variables, Values, [(f, collect_variables(f)) for f in Constraints])
    try:
        return backtrack_search({}, csp)
    except Backtrack:
        return None
```

Figure 3.7: A backtracking CSP solver.

1. First, the CSP `P` is split into its components.

2. Next, for every constraint `f` of the given CSP, we compute the set of variables that are used in `f`. This is done using the function `collect_variables` that is shown in Figure 3.10 on page 55. These variables are then stored together with the constraint `f` and the correspondingly modified data structure is stored in the variable `csp` and is called an augmented CSP.

   The reason to compute and store these variables is efficiency: When we later check whether a constraint `f` is satisfied for a partial variable assignment `Assignment` where `Assignment` is stored as a dictionary, we only need to check the constraint `f` iff all of the variables occurring in `f` are elements of the domain of `Assignment`. It would be wasteful to compute these variables every time that a partial variable assignment is extended.

3. Next, we call the function `backtrack_search` to compute a solution of `CSP`. This function is enclosed in a `try-except`-block that catches exceptions of class `Backtrack`. This class is defined as follows:

   ```
   class Backtrack(Exception):
       pass
   ```

   Its only purpose is to create a name for the special kind of exceptions used to administer backtracking. The reason for enclosing the call to `backtrack_search` in a `try-except`-block is that the function `backtrack_search` either returns a solution or, if it is not able to find a solution, it raises an exception of class `Backtrack`. The `try-except`-block ensures that this exception is silently discarded.

```
1  def backtrack_search(Assignment, P):
2      Variables, Values, Constraints = P
3      if len(Assignment) == len(Variables):
4          return Assignment
5      var = arb(Variables - Assignment.keys())
6      for value in Values:
7          try:
8              if is_consistent(var, value, Assignment, Constraints):
9                  NewAss = Assignment.copy()
10                 NewAss[var] = value
11                 return backtrack_search(NewAss, P)
12         except Backtrack:
13             continue
14     raise Backtrack()
```

Figure 3.8: A backtracking CSP solver: The function `backtrack_search`.

Next, we discuss the implementation of the function `backtrack_search` that is shown in Figure 3.8 on page 53. This function receives a partial assignment `Assignment` as input together with an augmented CSP `P`. This partial assignment is consistent with `P`: If `f` is a constraint of `CSP` such that all the variables occurring in `f` are members of `dom(Assignment)`, then evaluating `f` using `Assignment` yields `true`. Initially, this partial assignment is empty and hence trivially consistent. The idea is to extend this partial assignment until it is a complete variable assignment. We take care to ensure that this partial variable assignment remains consistent when it is extended. This way, once this assignment is complete it has to satisfy all the constraints of the given `CSP`.

1. First, the augmented CSP `P` is split into its components.

2. Next, if `Assignment` is already a complete variable assignment, i.e. if the dictionary `Assignment` has as many elements as there are variables, then it must be a solution of the `CSP` and, therefore, it is returned. The reason is that the function `backtrack_search` is only called with a consistent partial assignment.

3. Otherwise, we have to extend the partial `Assignment`. In order to do so, we first have to select a variable `var` that has not yet been assigned a value in `Assignment` so far. This is done in line 5 using the function `arb` that selects an arbitrary variable from its input set.

4. Next, we try to assign a `value` to the selected variable `var`. After assigning a `value` to `var`, we immediately check whether this assignment would be consistent with the constraints using the function `is_consistent`. If the partial `Assignment` turns out to be consistent, the partial variable `Assignment` is extended to the new partial assignment `NewAss` that satisfies

   `NewAss[var] = value`.

   Then, the function `backtrack_search` is called recursively to complete this new partial assignment. If this is successful, the resulting assignment is a solution that is returned. Otherwise, the recursive call of `backtrack_search` will raise an exception. This exception is muted by the `try-except`-block that surrounds the call to `backtrack_search`. In that case, the `for`-loop generates a new `value` that can be assigned to the variable `var`. If all possible values have been tried and none was successful, the `for`-loop ends and we have to backtrack, i.e. we have to reassign one of the variables that have been assigned earlier. This is done by raising a `Backtrack` exception. This exception is then caught by one of the prior invocations of `backtrack_search`. If all variable assignments have been tried and none is successful, then the `Backtrack` exception propagates back to the function `solve`, which will return `None` in that case.

```python
def is_consistent(var, value, Assignment, Constraints):
    NewAssign      = Assignment.copy()
    NewAssign[var] = value
    return all(eval(f, NewAssign) for (f, Vs) in Constraints
                                  if  var in Vs and Vs <= NewAssign.keys()
              )
```

Figure 3.9: The definition of the function `is_consistent`.

We still need to discuss the implementation of the auxiliary function `is_consistent` shown in Figure 3.9. This function takes a variable `var`, a `value`, a partial `Assignment` and a set of `Constraints` as arguments. It is assumed that `Assignment` is partially consistent with respect to the set `Constraints`, i.e. for every formula `f` occurring in `Constraints` such that

$$\texttt{vars}(\texttt{f}) \subseteq \texttt{dom}(\texttt{Assignment})$$

holds, the formula `f` evaluates to `True` given the `Assignment`. The purpose of `is_consistent` is to check, whether the extended assignment

$$\texttt{NewAssign := Assignment} \cup \{\texttt{var} \mapsto \texttt{value}\}$$

that assigns `value` to the variable `var` is still partially consistent with `Constraints`. To this end, the `for`-loop iterates over all formulas `f` in `Constraints`. However, we only have to check those formulas `f` that contain the variable `var` and, furthermore, have the property that

$$\texttt{vars}(\texttt{f}) \subseteq \texttt{dom}(\texttt{NewAssign}),$$

i.e. all variables occurring in the formula `f` need to have a value assigned in `NewAssign`. The reasoning is as follows:

1. If `var` does not occur in the formula `f`, then adding `var` to `Assignment` cannot change the result of evaluating `f` and as `Assignment` is assumed to be partially consistent with respect to `f`, `NewAssign` is also partially consistent with respect to `f`.

2. If $\texttt{dom}(\texttt{NewAssign}) \not\subseteq \texttt{vars}(\texttt{f})$, then `f` can not be evaluated anyway.

   Note that the domain of a variable assignment `A` can be computed with the expression `A.keys()` since `A` is represented as a dictionary in *Python*.

Finally, let us discuss the function `collect_variables` that is shown in Figure 3.10 on page 55. This function uses the module `extractVariables` that provides the function `extractVars`(e). This function takes a string e that can be interpreted as a *Python* expression as its argument and returns the set of all variables and function symbols occurring in the expression e. As we only want to keep the variable names, the function `collect_variables` takes care to eliminate the function symbols. This is done by making use of the fact that all function symbols that have been defined are members

```
import extractVariables as ev

def collect_variables(expr):
    return { var for var in ev.extractVars(expr)
                  if  var not in dir(__builtins__)
                  if  var not in ['and', 'or', 'not']
           }
```

Figure 3.10: The function `collectVars`.

of the list `dir(__builtins__)`. It turns out that the keyword "`and`", "`or`", and "`not`" also need to be removed since they might also be members of the set returned by `extractVars(expr)`.

If we use the program discussed in this section, we can solve the 8 queens problem in about 22 milliseconds. Hence, for the eight queens problem backtracking is more than a thousand times faster than brute force search.

Exercise 7: There are many different versions of the *zebra puzzle*. The version below is taken from *Wikipedia*. The puzzle reads as follows:

(a) There are five houses.

(b) The Englishman lives in the red house.

(c) The Spaniard owns the dog.

(d) Coffee is drunk in the green house.

(e) The Ukrainian drinks tea.

(f) The green house is immediately to the right of the ivory house.

(g) The Old Gold smoker owns snails.

(h) Kools are smoked in the yellow house.

(i) Milk is drunk in the middle house.

(j) The Norwegian lives in the first house.

(k) The man who smokes Chesterfields lives in the house next to the man with the fox.

(l) Kools are smoked in the house next to the house where the horse is kept.

(m) The Lucky Strike smoker drinks orange juice.

(n) The Japanese smokes Parliaments.

(o) The Norwegian lives next to the blue house.

(p) Who drinks water?

(q) Who owns the zebra?

In order to solve the puzzle, we also have to know the following facts:

- Each of the five houses is painted in a different colour.
- The inhabitants of the five houses are of different nationalities, and
- they own different pets, drink different beverages, and smoke different brands of cigarettes.

Formulate the zebra puzzle as a constraint satisfaction problem and solve the puzzle using the program discussed in this section. ◇

3.4 Constraint Propagation

1 Once we have chosen a value for a variable, this choice influences the values that are still available for other variables. For example, suppose that in order to solve the n queens problem we place the queen in row one in the second column, then no other queen can be placed in that column. Furthermore, due to the constraints on diagonals, the queen in row two can not be placed in any of the first three columns. Abstractly, constraint propagation works as follows.

1. Before the search is started, we create a dictionary `ValuesPerVar`. Initially, for every variable `x`, the set

   `ValuesPerVar[x]`

   contains all values `v` from the set `Values`. As soon as we discover that assigning a value `v` to the variable `x` is inconsistent with the variable assignments that have already taken place for other variables, the value `v` will be removed from the set `ValuesPerVar[x]`.

2. As long as the given CSP is not solved, we choose a variable `x` that has not been assigned a value yet. This variable is chosen using the most constrained variable heuristic: We choose a variable `x` such that the number of values in the set

   `ValuesPerVar[x]`

   is minimal. This is done because we have to find values for all variables. If the current partial variable assignment can not be completed into a solution, then we want to find out this fact as soon as possible. Therefore, we try to find the values for the most difficult variables first. A variable is more difficult to get right if it has only a few values left that can be used to instantiate it.

3. Once we have picked a variable `x`, we next iterate over all values `v` in `ValuesPerVar[x]`. Once we have assigned a value `v` to the variable `x`, we propagate the consequences of this assignment:

   (a) For every constraint `f` that mentions only the variable `x` and one other variable `y` that has not yet been instantiated, we compute the set `Legal` of those values from `ValuesPerVar[y]` that can be assigned to `y` without violating the constraint `f`.

   (b) Then, the set `ValuesPerVar[y]` is updated to the set `Legal` and we go back to step 2.

It turns out that elaborating the idea outlined above can enhance the performance of backtracking search considerably. Figure 3.11 on page 57 shows an implementation of constraint propagation. In addition to the ideas described above, this implementation takes care of unary constraints, i.e. constraints that contain only a single variable, as these constraints can be solved prior to the other constraints without backtracking.

```
def solve(P):
    Variables, Values, Constraints = P
    Annotated    = { (f, collect_variables(f)) for f in Constraints }
    ValuesPerVar = { v: Values for v in Variables }
    UnaryConstrs = { (f, V) for f, V in Annotated if len(V) == 1 }
    OtherConstrs = { (f, V) for f, V in Annotated if len(V) >= 2 }
    try:
        for f, V in UnaryConstrs:
            var = arb(V)
            ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
        return backtrack_search({}, ValuesPerVar, OtherConstrs)
    except Backtrack:
        return None
```

Figure 3.11: Constraint Propagation.

In order to implement constraint propagation, it is necessary to administer the values that can be used to instantiate the different variables separately, i.e. for every variable `x` we need to know which values are admissible for `x`. To this end, we need a dictionary `ValuesPerVar` that contains the set of possible values for every variable `x`. Initially, this dictionary assigns the set `Values` to every variable. Next, we take care of the unary constraints and shrink these sets so that the unary constraints are satisfied. Then, whenever we assign a value to a variable `x`, we inspect those constraints that mention the variable `x` and exactly one other yet unassigned variable `y` and shrink the set of values `ValuesPerVar`[`y`] that can be assigned to this variables `y`. This process is called constraint propagation and is described in more detail below when we discuss the function `propagate`.

1. The function `solve` receives a Csp `P` as its argument. The Csp `P` is first split into its three components and the constraints are annotated with the sets of variables occurring in them. These annotated constraints are stored in the set `Annotated`.

2. The most important data structure maintained by the function `solve` is the dictionary

   `ValuesPerVar`.

   Given a variable `v`, this dictionary assigns the set of values that can be used to instantiate this variable. Initially, this set is the same for all variables and is equal to `Values`.

3. In order to solve the unary constraints we first have to find them. The set `UnaryConstrs` contains all those pairs (`f`, `V`) from the set of annotated constraints such that the set of variables `V` occurring in `f` only contains a single variable.

4. Similarly, the set `OtherConstrs` contains those constraints that involve two or more variables.

5. In order to solve the unary constraints, we iterate over these constraints and shrink the set of values associated with the variable occurring in the constraint as dictated by the constraint. This is done using the function `solve_unary`.

6. Then, we start backtracking search using the function `backtrack_search`. Besides backtracking, the implementation of `backtrack_search` that we present below implements the most constraint variable heuristic and constraint propagation.

```
def solve_unary(f, x, Values):
    Legal = { value for value in Values if eval(f, { x: value }) }
    if not Legal:
        raise Backtrack()
    return Legal
```

Figure 3.12: Implementation of `solve_unary`.

The function `solve_unary` shown in Figure 3.12 on page 58 takes a unary constraint `f`, the variable `x` occurring in `f` and the set of values `Values` that can be assigned to this variable. It returns the subset of values that can be substituted for the variable `x` without violating the given constraint `f`. If this set is empty, a `Backtrack` exception is raised since in that case the given Csp is unsolvable.

```
def backtrack_search(Assignment, ValuesPerVar, Constraints):
    if len(Assignment) == len(ValuesPerVar):
        return Assignment
    x = most_constrained_variable(Assignment, ValuesPerVar)
    for v in ValuesPerVar[x]:
        try:
            NewValues = propagate(x, v, Assignment, Constraints, ValuesPerVar)
            NewAssign = Assignment.copy()
            NewAssign[x] = v
            return backtrack_search(NewAssign, NewValues, Constraints, lcv)
        except Backtrack:
            continue
    raise Backtrack()
```

Figure 3.13: Implementation of `backtrack_search`.

The function `backtrack_search` shown in Figure 3.13 on page 58 is called with a partial variable `Assignment` that is guaranteed to be consistent, a dictionary `ValuesPerVar` associating every variable with the set of values that might be substituted for this variable, and a set of annotated `Constraints`. It tries to complete `Assignment` and thereby computes a solution of the given Csp.

1. If the partial `Assignment` is already complete, i.e. if it assigns a value to every variable, then a solution to the given Csp has been found and this solution is returned. As the dictionary `ValuesPerVar` has an entry for every variable, its size is the same as the number of variables. Therefore, `Assignment` is complete iff it has the same size as `ValuesPerVar`.

2. Otherwise, we choose a variable `x` such that the number of values that can still be used to instantiate `x` is minimal. This strategy is known as the most constrained variable heuristic. The variable `x` is computed using the function `most_constrained_variable` that is shown in Figure 3.14 on page 59.

   The logic behind choosing a maximally constrained variables is that these variables are the most difficult to get right. If we have a partial assignment that is inconsistent, then we will discover this fact earlier if we try the most difficult variables first. This might save us a lot of unnecessary backtracking later.

3. Next, we try to find a value that can be assigned to the variable `x`. To this end we iterate over all values in `ValuesPerVar[x]`. Note that since `ValuesPerVar[x]` is, in general, smaller than the set `Values` of all values of the Csp, the `for`-loop in this version of backtracking search is more efficient than the corresponding `for`-loop in backtracking search discussed in the previous section.

4. If assigning the value `v` to the variable `x` is consistent, we propagate the consequences of this assignment using the function `propagate` shown in Figure 3.15 on page 60. This function updates the dictionary `ValuesPerVar` for all variables that are still unassigned.

5. Finally, the partial variable `Assignment` is updated to include the assignment of `v` to `x` and the recursive call to `backtrack_search` tries to complete this new assignment and thereby compute a solution to the given Csp.

```
def most_constrained_variable(Assignment, ValuesPerVar):
    Unassigned = { (x, len(U)) for x, U in ValuesPerVar.items()
                               if  x not in Assignment
                 }
    minSize    = min(lenU for _, lenU in Unassigned)
    return arb({ x for x, lenU in Unassigned if lenU == minSize })
```

Figure 3.14: Finding a most constrained variable.

Figure 3.14 on page 59 shows the implementation of the function `most_constrained_variable`. The function `most_constrained_variable` takes a partial `Assignment` and a dictionary `ValuesPerVar` returning for all variables `x` the set of values `ValuesPerVar[x]` that can be assigned to `x`.

1. First, this function computes the set of `Unassigned` variables. For every variable `x` that has not yet been assigned a value in `Assignment` this set contains the pair $(\texttt{x}, \texttt{len}(U))$, where U is the set of values that still might be tried for the variable `x`.

2. Next, `minSize` is the minimum size of the sets `ValuesPerVar[x]` for all unassigned variables.

3. Finally, an arbitrary variable `x` that has only `minSize` values available is returned.

The function `propagate` shown in Figure 3.15 on page 60 implements constraint propagation. It takes the following inputs:

(a) `x` is a variable and `v` is a value that is assigned to the variable `x`.

(b) `Assignment` is a partial assignment that contains assignments for those variables that are different from the variable `x`.

(c) `Constraints` is a set of annotated constraints, i.e. this set contains pairs of the form `(f, Vars)`, where `f` is a constraint and `Vars` is the set of variables occurring in `f`.

(d) `ValuesPerVar` is a dictionary assigning sets of possible values to all variables.

```
1  def propagate(x, v, Assignment, Constraints, ValuesPerVar):
2      ValuesDict    = ValuesPerVar.copy()
3      ValuesDict[x] = { v }
4      BoundVars     = set(Assignment.keys())
5      for f, Vars in Constraints:
6          if x in Vars:
7              UnboundVars = Vars - BoundVars - { x }
8              if len(UnboundVars) == 1:
9                  y = arb(UnboundVars)
10                 Legal = set()
11                 for w in ValuesDict[y]:
12                     NewAssign = Assignment.copy()
13                     NewAssign[x] = v
14                     NewAssign[y] = w
15                     if eval(f, NewAssign):
16                         Legal.add(w)
17                 if len(Legal) == 0:
18                     raise Backtrack()
19                 ValuesDict[y] = Legal
20     return ValuesDict
```

Figure 3.15: Constraint Propagation.

The purpose of the function `propagate` is to restrict the values of variables different from the variable `x` by propagating the consequences of setting `x` to `v`. To this end the function `propagate` updates the dictionary `ValuesPerVar` by taking into account the consequences of assigning the value `v` to the variable `x`. The implementation of `propagate` proceeds as follows.

1. Initially, we copy the Dictionary `ValuesPerVar` to the dictionary `ValuesDict`
2. As `x` is assigned the value `v`, the corresponding entry in the dictionary `ValuesDict` is changed accordingly.
3. `BoundVars` is the set of those variable that already have a value assigned.
4. Next, `propagate` iterates over all constraints `f` such that the variable `x` occurs in `f`.
5. `UnboundVars` is the set of those variables occurring in `f` that are different from `x` and that do not yet have a value assigned.
6. If there is exactly one unbound variable `y` in the constraint `f`, then we can test those values that satisfy `f` and recompute the set `ValuesDict[x]`.
7. As the set `UnboundVars` contains just a single variable in line 9, the function `arb` returns this variable.
8. In order to recompute the set `ValuesDict[y]`, all values `w` in `ValuesDict[y]` are tested. The set `Legal` contains all values `w` that can be assigned to the variable `y` without violating the constraint `f`.

9. If it turns out that `Legal` is the empty set, then this means that the constraint `f` is inconsistent with assigning the value `v` to the variable `x`. Hence, in this case the search has to backtrack.
10. Otherwise, the set of admissible values for `y` is updated to be the set `Legal`.
11. Finally, the dictionary `ValuesDict` is returned.

I have tested the program described in this section using the eight queens puzzle. It takes about 18 milliseconds to find a solution. I have also tested it with the Zebra Puzzle described in a previous exercise. It solves this puzzle in 21 milliseconds. To compare, the backtracking algorithm shown in the previous section takes roughly 10 seconds to solve this puzzle.

3.5 Consistency Checking*

So far, the constraints in the constraints satisfaction problems discussed are either unary constraints or binary constraints: A unary constraint is a constraint `f` such that the formula `f` contains only one variable, while a binary constraint contains two variables. If we have a constraint satisfaction problem that involves also constraints that mention more than two variables, then the constraint propagation shown in the previous section is not as effective as it is only used for a constraint `f` if all but one variable of `f` have been assigned. For example, consider the cryptarithmetic puzzle shown in Figure 3.16 on page 61. The idea is that the letters “`S`”, “`E`”, “`N`”, “`D`”, “`M`”, “`O`”, “`R`”, “`Y`” are interpreted as variables ranging over the set of decimal digits, i.e. these variables can take values in the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Then, the string “`SEND`” is interpreted as a decimal number, i.e. it is interpreted as the number

$$\mathtt{S} \cdot 10^3 + \mathtt{E} \cdot 10^2 + \mathtt{N} \cdot 10^1 + \mathtt{D} \cdot 10^0.$$

The strings “`MORE` and “`MONEY`” are interpreted similarly. To make the problem interesting, the assumption is that different variables have different values. Furthermore, the digits at the beginning of a number should be different from 0.

Figure 3.16: A cryptarithmetic puzzle

A naïve approach to solve this problem would be to code it as a constraint satisfaction problem that has, among others, the following constraint:

$$(\mathtt{S} \cdot 10^3 + \mathtt{E} \cdot 10^2 + \mathtt{N} \cdot 10 + \mathtt{D}) + (\mathtt{M} \cdot 10^3 + \mathtt{O} \cdot 10^2 + \mathtt{R} \cdot 10 + \mathtt{E}) = \mathtt{M} \cdot 10^4 + \mathtt{O} \cdot 10^3 + \mathtt{N} \cdot 10^2 + \mathtt{E} \cdot 10 + \mathtt{Y}.$$

The problem with this constraint is that it involves far too many variables. As this constraint can only be checked when all the variables have values assigned to them, the backtracking search would essentially boil down to a mere brute force search. We would have 8 variables that each could take 10 different values and hence we would have to test 10^8 possible assignments. In order to do better, we have to perform the addition shown in Figure 3.16 column by column, just as it is taught in elementary school. Figure 3.17 on page 62 shows how this can be implemented in *Python*.

Notice that we have introduced three additional variables “`C1`”, “`C2`”, “`C3`”. These variables serve as the carry digits. For example, “`C1`” is the carry digit that we get when we add the final digits of

```
def crypto_csp():
    Digits       = { 'S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y' }
    Variables    = Digits | { 'C1', 'C2', 'C3' }
    Values       = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    Constraints  = allDifferent(Digits)
    Constraints |= { '(D + E)      % 10 == Y', '(D + E)      // 10 == C1',
                     '(N + R + C1) % 10 == E', '(N + R + C1) // 10 == C2',
                     '(E + O + C2) % 10 == N', '(E + O + C2) // 10 == C3',
                     '(S + M + C3) % 10 == O', '(S + M + C3) // 10 == M'
                   }
    Constraints |= { 'S != 0', 'M != 0' }
    Constraints |= { 'C1 < 2', 'C2 < 2', 'C3 < 2' }
    return Variables, Values, Constraints

def allDifferent(Variables):
    return { f'{x} != {y}' for x in Variables
                           for y in Variables
                           if x < y
           }
```

Figure 3.17: Formulating “$\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$” as a CSP.

the two numbers, i.e. we have

$$\texttt{D} + \texttt{E} = \texttt{C1} \cdot 10 + \texttt{Y}.$$

This equation still contains four variables. We can split this equation into two smaller equations that each involve only three variables with the help of the modulo operator “`%`” and the operator for integer division “`//`“ as follows:

$$(\texttt{D} + \texttt{E})\ \texttt{\%}\ 10 = \texttt{Y} \quad \text{and} \quad (\texttt{D} + \texttt{E})\ \texttt{//}\ 10 = \texttt{C1}.$$

If we solve the cryptarithmetic puzzle as coded in Figure 3.17 on page 62 using the constraint solver developed, then solving the puzzle takes about a second on my computer. The reason is that most constraints involve either three or four variables and therefore the effects of constraint propagation kick only in when many variables have already been initialized. However, we can solve the problem in less than 50 milliseconds if we add the following constraints for the variables “`C1`”, “`C2`”, “`C3`”:

`"C1 < 2", "C2 < 2", "C3 < 2"`.

Although these constraints are certainly true, the problem with this approach is that we would prefer our constraint solver to figure out these constraints by itself. After all, since `D` and `E` are both less than 10, their sum is obviously less than 20 and hence the carry `C1` has to be less than 2. This line of reasoning is known as consistency maintenance: Assume that the formula f is a constraint and the set of variables occurring in f has the form

$$\texttt{Var}(f) = \{x\} \cup R \quad \text{where } x \notin R,$$

i.e. the variable x occurs in the constraint f and, furthermore, $R = \{y_1, \cdots, y_n\}$ is the set of all variables occurring in f that are different from x. In addition, assume that we have a dictionary

`ValuesPerVar` such that for every variable y, the dictionary entry `ValuesPerVar`$[y]$ is the set of values that can be substituted for the variable y. The formal definition follows.

Definition 5 (Consistent Value for a Variable) A value v is consistent for the variable x with respect to the constraint f iff the partial assignment $\{x \mapsto v\}$ can be extended to an assignment A satisfying the constraint f, i.e. for every variable y_i that is different from x we have to find a value $w_i \in$ `ValuesPerVar`$[y_i]$ such that the resulting assignment $A = \{x \mapsto v, y_1 \mapsto w_1, \cdots, y_n \mapsto w_n\}$ satisfies the equations

$$\texttt{eval}(f, A) = \texttt{True}.$$

Here, the function `eval` takes a formula f and a variable assignment A and evaluates f using this assignment. ◇

Given a CSP $\mathcal{P} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$, the algorithm for consistency maintenance is shown below.

1. The dictionary `ValuesPerVar` is initialized as follows:

   $$\texttt{ValuesPerVar}[x] := \texttt{Values} \quad \text{for all } x \in \texttt{Variables},$$

   i.e. initially every variable x can take any value from the set of `Values`.

2. Next, the set `UncheckedVariables` is initialized to the set of all `Variables`:

   $$\texttt{UncheckedVariables} := \texttt{Variables}.$$

3. As long as the set `UncheckedVariables` is not empty, we remove one variable x from this set:

   $$x := \texttt{UncheckedVariables.pop}()$$

4. We iterate over all constraints f such that x occurs in f.

   (a) For every value $v \in$ `ValuesPerVar`$[x]$ we check whether v is consistent with f.

   (b) If the value v is not consistent with f, then v is removed from `ValuesPerVar`$[x]$. Furthermore, all variables connected to x are added to the set of `UncheckedVariables`. Here we define y variable $y \neq x$ to be connected to x if there is some constraint f such that both x and y occur in f. The reason is that some of their values might have become inconsistent by removing the value v from `ValuesPerVar`$[x]$.

5. Once the set `UncheckedVariables` is empty, the algorithm terminates. Otherwise, we jump back to step 3 and remove the next variable from the set `UncheckedVariables`.

The algorithm terminates as every iteration removes either a variable from the set `UncheckedVariables` or it removes a value from one of the sets `ValuesPerVar`$[y]$ for some variable y. Although the set `UncheckedVariables` can grow during the algorithm, the union

$$\bigcup_{x \in \texttt{Vars}} \texttt{ValuesPerVar}[x]$$

can never grow: Every time the set `UncheckedVariables` grows, for some variable x the set

$$\texttt{ValuesPerVar}[x]$$

shrinks. As the sets `ValuesPerVar`$[x]$ are finite for all variables x, the set `UncheckedVariables` can only grow a finite number of times. Once the set `UncheckedVariables` does not grow any more, every iteration of the algorithm removes one variable from this set and hence the algorithm terminates eventually.

```python
def enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected):
    UncheckedVars = set(Var2Formulas.keys())
    while UncheckedVars:
        variable    = UncheckedVars.pop()
        RemovedVals = set()
        for f in Var2Formulas[variable]:
            OtherVars = Annotated[f] - { variable }
            for value in ValuesPerVar[variable]:
                if not exists_values(variable,value,f,OtherVars,ValuesPerVar):
                    RemovedVals   |= { value }
                    UncheckedVars |= Connected[variable]
        ValuesPerVar[variable] -= RemovedVals
        if len(ValuesPerVar[variable]) == 0: # the problem is unsolvable
            raise Backtrack()
```

Figure 3.18: Consistency maintenance in *Python*.

Figure 3.18 on page 64 shows how consistency maintenance can be implemented in *Python*. The function `enforce_consistency` takes four arguments.

(a) `ValuesPerVar` is a dictionary associating the set of possible values with each variable.

(b) `Var2Formulas` is a dictionary. For every variable x, `Var2Formulas`$[x]$ is the set of those constraints f such that x occurs in f.

(c) `Annotated` is a dictionary mapping constraints to the set of variables occurring in them, i.e. if f is a constraint, then `Annotated`$[f]$ is the set of variables occurring in f.

(d) `Connected` is a dictionary that takes a variable x and returns the set of all variables that are connected to x via a common constraint f, i.e. we have $y \in$ `Connected`$[x]$ iff there exists a constraint f such that both x and y occur in f and, furthermore, $x \neq y$.

The function `enforce_consistency` modifies the dictionary `ValuesPerVar` so that once the function has terminated, for every variable x the values in the set `ValuesPerVar`$[x]$ are consistent with the constraints for x. The implementation works as follows:

1. Initially, all variables need to be checked for consistency. Therefore, `UncheckedVars` is defined to be the set of all variables that occur in any of the constraints.

2. The `while`-loop iterates as long as there are still variables x left in `UncheckedVars` such that the consistency of `ValuesPerVar`$[x]$ has not been established.

3. Next, a variable `variable` is selected and removed from `UncheckedVars`.

4. `RemovedVals` is the subset of those values that are found to be inconsistent with some constraint for `variable`.

5. We iterate over all constraints `f` $\in$ `Var2Formulas[variable]`.

6. `OtherVars` is the set of variables occurring in `f` that are different from the chosen variable `variable`.

7. We iterate over all `value` ∈ `ValuesPerVar[variable]` that can be substituted for the variable `variable` and check whether `value` is consistent with `f`. To this end, we need to find values that can be assigned to the variables in the set `OtherVars` such that `f` evaluates as `True`. This is checked using the function `exists_values`.

8. If we do not find such values, then `value` is inconsistent for the variable `variable` w.r.t. `f` and needs to be removed from the set `ValuesPerVar`[`variable`]. Furthermore, all variables that are connected to `variable` have to be added to the set `UncheckedVars`. The reason is that once a value is removed for the variable `var`, the value assigned to another variable y occurring in a constraint that mentions both `var` and y might now become inconsistent.

9. The set of values that are known to be consistent for `variable` is stored as `ValuesPerVar`[`variable`].

10. If there are no consistent values for `variable` left, the problem is unsolvable and an exception is raised.

```python
def exists_values(var, val, f, Vars, ValuesPerVar):
    Assignments = all_assignments(Vars, ValuesPerVar)
    return any(eval(f, extend(A, var, val)) for A in Assignments)

def extend(A, x, v):
    B = A.copy()
    B[x] = v
    return B

def all_assignments(Variables, ValuesPerVar):
    Variables = set(Variables) # turn frozenset into a set
    if not Variables:
        return [ {} ]  # list containing empty assignment
    var         = Variables.pop()
    Assignments = all_assignments(Variables, ValuesPerVar)
    return [ extend(A, var, val) for A in Assignments
                                 for val in ValuesPerVar[var]
           ]
```

Figure 3.19: The implementation of `exists_value`.

Figure 3.19 on page 65 shows the implementation of the function `exists_values` that is used in the implementation of `enforce_consistency`. This function is called with five arguments.

(a) `var` is variable.

(b) `val` is a value that is to be assigned to `var`.

(c) `f` is a constraint such that the variable `var` occurs in `f`

(d) `Vars` is the set of all those other variables occurring in `f`, i.e. the set of those variables that occur in `f` but that are different from `var`.

(e) `ValuesPerVar` is a dictionary associating the set of possible values with each variable.

The function checks whether the partial assignment $\{$`var` $\mapsto$ `val`$\}$ can be extended so that the constraint `f` is satisfied. To this end it needs to create the set of all possible assignments. This set is generated using the function `all_assignments`. This function gets a set of variables `Vars` and a dictionary that assigns to every variable `var` in `Vars` the set of values that might be assigned to `var`. It returns a list containing all possible variable assignments. The implementation proceeds as follows:

1. As the argument `Variables` is a `frozenset` but we need to modify this set for the recursive call of `all_assignments`, we transform the `frozenset` into a `set`.
2. If the set of variables `Vars` is empty, the empty dictionary can serve as a mapping that assigns a value to every variable in `Vars`.
3. Otherwise, we remove a variable `var` from `Vars` and get the set of `Values` that can be assigned to `var`.
4. Recursively, we create the set of all `Assignments` that associate values with the remaining variables.
5. Finally, the set of all possible assignments is the set of all combinations of assigning a value `val` $\in$ `Values` to `var` and assigning the remaining variables according to an assignment `A` $\in$ `Assignments`. Here we have to make use of the function `extend` that takes a dictionary `A`, a key `x` not occurring in `A` and a value `v` and returns a new dictionary that maps `x` to `v` and otherwise coincides with `A`.

On one hand, consistency checking is a pre-processing step that creates a lot of overhead.[1] Therefore, it might actually slow down the solution of some constraint satisfaction problems that are easy to solve using just backtracking and constraint propagation. On the other hand, many difficult constraint satisfaction problems can not be solved without consistency checking.

Figure 3.20 on page 67 shows how consistency checking is integrated into a constraint solver as a pre-processing step. The procedure `solve`(P) takes a constraint satisfaction problem P as input. The function `solve` converts the CSP P into an augmented CSP where every constraint f is annotated with the variables occurring in f. Furthermore, the function solve maintains the following data structures:

1. `VarsInConstrs` is the set of all variables occurring in any constraint.
2. `ValuesPerVar` is a dictionary mapping variables to sets of values. For every variable x occurring in a constraint of P, the expression `ValuesPerVar`(x) is the set of values that can be used to instantiate the variable x. Initially, `ValuesPerVar`(x) is set to `Values`, but as the search for a solution proceeds, the sets `ValuesPerVar`(x) are reduced by removing any values that cannot be part of a solution.
3. `Annotated` is a dictionary. For every constraint f we have that `Annotated`$[f]$ is the set of all variables occurring in f.
4. `UnaryConstrs` is a set of pairs of the form (f, V) where f is a constraint containing only a single variable and V is the set containing just this variable.
5. `OtherConstrs` is a set of pairs of the form (f, V) where f is a constraint containing more than one variable and V is the set of all variables occurring in f.

[1] To be fair, the implementation shown in this section is far from optimal. In particular, by remembering which combinations of variables and values work for a given formula, the overhead can be reduced significantly. I have refrained from implementing this optimization because I did not want the code to get too complex.

```
def solve(P):
    Variables, Values, Constraints = P
    VarsInConstrs  = union([ collect_variables(f) for f in Constraints ])
    MisspelledVars = (VarsInConstrs - Variables) | (Variables - VarsInConstrs)
    if MisspelledVars:
        print("Did you misspell any of the following Variables?")
        for v in MisspelledVars:
            print(v)
    ValuesPerVar = { x: Values.copy() for x in Variables }
    Annotated    = { f: collect_variables(f) for f in Constraints }
    UnaryConstrs = { (f, V) for f, V in Annotated.items()
                            if  len(V) == 1
                   }
    OtherConstrs = { (f, V) for f, V in Annotated.items()
                            if  len(V) >= 2
                   }
    Connected    = {}
    Var2Formulas = variables_2_formulas(OtherConstrs)
    for x in Variables:
        Connected[x] = union([ V for f, V in Annotated.items()
                                 if  x in V
                             ]) - { x }
    try:
        for f, V in UnaryConstrs:
            var               = arb(V)
            ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
        enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected)
        for x, Values in ValuesPerVar.items():
            print(f'{x}: {Values}')
        return backtrack_search({}, ValuesPerVar, OtherConstrs)
    except Backtrack:
        return None
```

Figure 3.20: A constraint solver with consistency checking as a preprocessing step.

6. `Connected` is a dictionary mapping variables to sets of variables. If x is a variable, then `Connected`$[x]$ is the set of those variables y such that there is a constraint f that mentions both the variable x and the variable y.

7. `Var2Formulas` is a dictionary mapping variables to sets of formulas. For every variable x, `Var2Formulas`$[x]$ is the set of all those non-unary constraints f such that x occurs in f.

After initializing these data structures, the unary constraints are immediately solved. Then the function `enforce_consistency` performs consistency maintenance: Formally, we define: A value v is consistent for x with respect to the constraint f iff the partial assignment $\{x \mapsto v\}$ can be extended to an assignment A satisfying the constraint f, i.e. for every variable $\mathtt{y}_i$ occurring in f there is a value

$w_i \in \texttt{ValuesPerVar}[y]$ such that

$$\texttt{evaluate}\big(f, \{x \mapsto v, y_1 \mapsto w_1, \cdots, y_n \mapsto w_n\}\big) = \texttt{True}.$$

The call to `enforce_consistency` shrinks the sets $\texttt{ValuesPerVars}[x]$ until all values in $\texttt{ValuesPerVars}[x]$ are consistent with respect to all constraints.

Finally, `backtrack_search` is called to solve the remaining constraint satisfaction problem by the means of both backtracking and constraint propagation.

3.6 Local Search*

There is another approach to solve constraint satisfaction problems. This approach is known as local search. The basic idea is simple: Given as constraint satisfaction problem $\mathcal{C}$ of the form

$$\mathcal{P} := \langle \texttt{Variables}, \texttt{Values}, \texttt{Constraints} \rangle,$$

local search works as follows:

1. Use consistency checking as an optional pre-processing step.
2. Initialize the values of the variables in `Variables` randomly.
3. If all `Constraints` are satisfied, return the solution.
4. For every $x \in \texttt{Variables}$, count the number of unsatisfied constraints that involve the variable x.
5. Set `maxNum` to be the maximum of these numbers, i.e. `maxNum` is the maximal number of unsatisfied constraints for any variable.
6. Compute the set `maxVars` of those variables that have `maxNum` unsatisfied constraints.
7. Randomly choose a variable x from the set `maxVars`.
8. Find a value $d \in \texttt{Values}$ such that by assigning d to the variable x, the number of unsatisfied constraints for the variable x is minimized.

   If there is more than one value d with this property, choose the value d randomly from those values that minimize the number of unsatisfied constraints.
9. Rinse and repeat until a solution is found.

Figure 3.21 on page 69 shows the preprocessing step. The function `solve` takes a constraint satisfaction problem `P` as its argument and performs consistency checking similar to the algorithm discussed in the previous section. Following the preprocessing it calls the function `local_search` that solves the given Csp.

Figure 3.22 on page 70 shows an implementation of local search in *Python*. We proceed to discuss this program line by line.

1. The function `local_search` takes three parameters.
   (a) `Variables` is the set of all variables occurring in the given Csp.
   (b) `ValuesPerVar` is a dictionary. For every variable `x`, `ValuesPerVar[x]` is the set of values that can be used to instantiate `x`.

```
def solve(P):
    Variables, Values, Constraints = P
    VarsInConstrs  = union([ collect_variables(f) for f in Constraints ])
    MisspelledVars = (VarsInConstrs - Variables) | (Variables - VarsInConstrs)
    if MisspelledVars:
        print("Did you misspell any of the following Variables?")
        for v in MisspelledVars:
            print(v)
    ValuesPerVar = { x: Values for x in Variables }
    Annotated    = { f: collect_variables(f) for f in Constraints }
    Connected    = {}
    Var2Formulas = variables_2_formulas(Annotated)
    for x in Variables:
        Connected[x] = union([V for f, V in Annotated.items() if x in V]) - {x}
    try:
        enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected)
    except Failure:
        return None
    return local_search(Variables, ValuesPerVar, Annotated)
```

Figure 3.21: A constraint solver using local search.

(c) `Annotated` is a dictionary. For every constraint f, `Annotated`$[f]$ is the set of variables occurring in f.

If the computation is successful, `local_search` returns a dictionary that encodes a solution of the given Csp by mapping variables to values.

2. The set `Variables` is turned into a list. This is necessary because the function

   `random.choice`(L)

   that is used to select a random element from L expects its argument L to be indexable, i.e. for a number $k \in \{0, \cdots, \texttt{len}(L) - 1\}$ the expression $L[k]$ needs to be defined.

3. `Assign` is a dictionary mapping all variables from the set `Variables` to values from the set `Values`. Initially the values are assigned randomly.

4. The variable `iteration` counts the number of times that we have changed the assignment `Assign` by reassigning a variable.

5. If we have reassigned a variable x in the last iteration of the loop, then we do not want to reassign it again in the next step since otherwise the program could get stuck in an infinite loop. Therefore, the variable `lastVar` stores the variable that has been reassigned in the previous iteration. We will ensure that in the next iteration step, another variable is chosen for reassignment.

6. At the beginning of the `while` loop, we count the number of conflicts for all variables, i.e. if x is a variable that is different from the variable that has been reassigned in the last iteration, then we count the number of conflicts that x causes. This number is defined as the number of constraints f such that

```
1  def local_search(Variables, ValuesPerVar, Annotated):
2      Variables = list(Variables)
3      Assign    = { x: random.choice(list(ValuesPerVar[x])) for x in Variables }
4      iteration = 0
5      lastVar   = arb(Variables)
6      while True:
7          Conflicts = [(numConflicts(x, Assign, Annotated), x) for x in Variables
8                                                               if  x != lastVar
9                      ]
10         maxNum, _ = Set.last(cast_to_Set(Conflicts))
11         if maxNum == 0 and numConflicts(lastVar, Assign, Annotated) == 0:
12             print(f'Number of iterations: {iteration}')
13             return Assign
14         if iteration % 11 == 0:    # avoid infinite loop
15             x = random.choice(Variables)
16         else:     # choose var with max number of conflicts
17             FaultyVars = [ var for (num, var) in Conflicts if  num == maxNum ]
18             x = random.choice(FaultyVars)
19         if iteration % 13 == 0:       # avoid infinite loop
20             newVal = random.choice(list(ValuesPerVar[x]))
21         else:
22             Conflicts = [ (numConflicts(x, extend(Assign, x, v), Annotated), v)
23                           for v in ValuesPerVar[x]
24                         ]
25             minNum, _  = Set.first(cast_to_Set(Conflicts))
26             ValuesForX = [ val for (n, val) in Conflicts if n == minNum ]
27             newVal     = random.choice(ValuesForX)
28         Assign[x]  = newVal
29         lastVar    = x
30         iteration += 1
```

Figure 3.22: Implementation of local search.

(a) x occurs in f and

(b) f is not satisfied.

This is done using the function `numConflicts` shown in Figure 3.23 on page 71. The list `Conflicts` defined in line 7 contains pairs of the form (n, x) where x is a variable and n is the number of conflicts that this variable is involved in.

7. In line 10 the list `Conflicts` is turned into a set that is represented as an ordered binary set. This set is effectively a priority queue that is ordered by the number of conflicts. We pick the variable with the most conflicts from this set and store the number of conflicts in `maxNum`, i.e. `maxNum` is the maximum number of conflicts that any variable is involved in.

8. Now if `maxNum` is 0 and additionally the variable `lastVar` that is excluded from the computation of the set `Conflicts` has no conflicts, then the given CSP has been solved and the solution is

returned.

9. Otherwise, the list `FaultyVars` defined in line `17` collects those variables that have a maximal number of conflicts.

10. In line `18` we choose a random variable `x` from this list as the variable to be reassigned. However, this is only done ten out of eleven times. In order to avoid running into an infinite loop where we keep changing the same variables, every 11^{th} iteration chooses `x` randomly. This is controlled by the test `iteration % 11 == 0` in line 16.

11. Line `22` computes a list `Conflicts` that this time contains pairs of the form (n, v) where n is the number of conflicts that the variable `x` would cause if we would assign the value v to `x`.

12. Line `25` casts the list `Conflicts` into a set that is represented as an ordered binary tree. This ordered binary tree is used as a priority queue that is ordered by the number of conflicts. We pick the smallest number of conflicts that any value v causes when `x` is assigned to v.

13. `ValuesForX` is the list of those values that cause only `minNum` conflicts when assigned to `x`.

14. `newVal` is a random element from this list that is then assigned to `x`. Again, this is only done twelve out of thirteen times. The 13^{th} time a random value is assigned to `x` instead.

15. In line `29` we remember that we have reassigned `x` in this iteration so that we don't reassign `x` in the next iteration again.

```
def numConflicts(x, Assign, Annotated):
    NewAssign = Assign.copy()
    return len([ (f, V) for (f, V) in Annotated
                        if x in V and not eval(f, NewAssign)
               ])
```

Figure 3.23: The function `numConflicts`.

The function `numConficts` is shown in Figure 3.23 on page 71. If x is a variable, `Assign` is a variable assignment and `Annotated` is a list of pairs of the form (f, V) where f is a constraint and V is the set of variables occurring in f, then `numConficts`$(x,$ `Assign`$,$ `Annotated`$)$ is the number of conflicts caused by the variable x.

Using the program discussed in this section, the n queens problem can be solved for a $\mathtt{n} = 1000$ in 30 minutes. As the memory requirements for local search are small, even much higher problem sizes can be tackled if sufficient time is available. It is a fact that often large problems, which are not inherently difficult, can be solved much faster with local search than with any other algorithm. However, we have to note that local search is incomplete: If a constraint satisfaction problem $\mathcal{P}$ has no solution, then local search loops forever. Therefore, in practise a dual approach is used to solve a constraint satisfaction problem. The constraint solver starts two threads: The first search does local search, the second thread tries to solve the problem via some refinement of backtracking. The first thread that terminates wins. The resulting algorithm is complete and, for a solvable problem, will have a performance that is similar to the performance of local search. If the problem is unsolvable, this will eventually be discovered by backtracking. Note, however, that the constraint satisfaction problem is NP-complete. Hence, it is unlikely that there is an efficient algorithm that works always. However, today many practically relevant constraint satisfaction problems can be solved in a reasonably short time.

3.7 Z3

We conclude this chapter with a discussion of the solver Z3. Z3 implements most of the state-of-the-art constraint solving algorithms and is exceptionally powerful. We introduce Z3 via a series of examples.

3.7.1 A Simple Text Problem

The following is a simple text problem from my old 8^{th} grade math book.

- *I have as many brothers as I have sisters.*
- *My oldest sister has twice as many brothers as she has sisters.*
- *How many children does my father have?*

However, in order to solve this puzzle we need two additional assumptions.

1. My father has no illegitimate children.
2. All of my fathers children identify themselves as either male or female.

Strangely, in my old math book these assumptions are not mentioned.

In order to infer the number of children we first have to determine whether I am male or female. If I were female, I would have as many brothers as my sister has. Now if my sister would have twice as many brother, this could only be true if I had no brothers. But then I would not have any sisters either and this contradicts the fact that I have an oldest sister. This contradiction shows that I have to be male.

If we denote the number of boys with the variable b and the number of girls with g, the problem statements are equivalent to the following two equations:

(a) $b - 1 = g$, since I am not my own brother.

(b) $2 \cdot (g - 1) = b$ as my sister is not my own sister.

Before we can start to solve this problem, we have to install `Z3` via `pip` using the following command in the shell:

```
pip install z3-solver
```

Once we have done this and we have added the directory

```
export PATH="~/opt/anaconda3/envs/ai/bin/
```

to the environment variable `PATH`, we can use the file shown in Figure 3.24 to solve the problem. The command to invoke `Z3` has the form

`z3` *file*`.z3`

where *file*`.z3` is the name of the file that stores the Z3 specification of the problem.

(a) Line 1 and 2 declare the variables `b` and `g` as integer variables.

With Z3 we are not confined to use a finite set of values. Instead we can use integer variables and floating point variables.

The syntax of `Z3` files is similar to the syntax of the programming language lisp. Later, we will only use the *Python* API of Z3. Therefore, you do not need to worry about this syntax.

```
1 (declare-const b Int)
2 (declare-const g Int)
3
4 (assert (= (- b 1) g))
5 (assert (= (* 2 (- g 1)) b))
6
7 (check-sat)
8 (get-model)
```

Figure 3.24: Solving a simple text problem with `Z3`.

(b) Line 4 specifies the equation $b - 1 = g$ as a constraint.

Note that we have to use prefix notation for all operators.

(c) Similarly, line 5 specifies the equation $2 * (g - 1) = b$ as a constraint.

(d) Line 7 asks Z3 to check whether the problem is solvable.

(e) Line 8 prints the solution of the problem.

If we run this command with the specification shown in Figure 3.24, then we get the output shown below:

```
sat
(
  (define-fun b () Int 4)
  (define-fun g () Int 3)
)
```

The string "`sat`" tells us that the problem is solvable and the following lines show that $b = 4$ and $g = 3$ is the solution, i.e. there are 4 boys and 3 girls.
Instead of using the command line to solve CSPs we will utilize the *Python* interface of Z3. There are two reasons why this is more convenient:

1. In an interesting CSP there can easily be hundreds of variables and thousands of constraints. It would be very inconvenient if we had to write these variables and constraints manually into a file.

2. The *Python* interface allows us to extract the solution that has been computed so that we can then proceed to use the values of the solution in our own programs.

The Python program shown in Figure 3.25 solves the text problem given above via the *Python* API of `Z3`.

1. In line 1 we import the module `z3` so that we can use the Python API of Z3. The documentation of this API is available at the following address:

   https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```
1  import z3
2
3  boys  = z3.Int('boys')
4  girls = z3.Int('girls')
5
6  S = z3.Solver()
7
8  S.add(boys - 1 == girls)
9  S.add(2 * (girls - 1) == boys)
10 S.check()
11 solution = S.model()
12
13 b = solution[boys ].as_long()
14 g = solution[girls].as_long()
15
16 print(f'My father has {b + g} children.')
```

Figure 3.25: Solving a simple text problem.

2. Lines 3 and 4 creates the `Z3` variables `boys` and `girls` as integer valued variables. The function `Int` takes one argument, which has to be a string. This string is the name of the variable. We store these variables in Python variables of the same name. It would be possible to use different names for the Python variables, but that would be very confusing.
3. Line 6 creates an object of the class `Solver`. This is the constraint solver provided by `Z3`.
4. Lines 8 and 9 add the constraints expressing that the number of girls is one less than the number of boys and that my sister has twice as many brothers as she has sisters as constraints to the solver `S`.
5. In line 10 the method `check` examines whether the given set of constraints is satisfiable. In general, this method returns one of the following results:
   (a) `sat` is returned if the problem is solvable, (`sat` is short for *satisfiable*)
   (b) `unsat` is returned if the problem is unsolvable,
   (c) `unknown` is returned if `Z3` is not powerful enough to solve the given problem.
6. Since in our case the method `check` returns `sat`, we can extract the solution that is computed via the method `model` in line 11.
7. In order to extract the values that have been computed by `Z3` for the variables `boys` and `girls`, we can use dictionary syntax and write `solution[boys]` and `solution[girls]` to extract these values. However, these values are not stored as integers but rather as objects of the class `IntNumRef`, which is some internal class of `Z3` to store integers. This class provides the method `as_long` that converts its object into an integer number.

Exercise 8: Solve the following text problem using `Z3`.

(a) *A Japanese deli offers both penguins and parrots.*

(b) *A parrot and a penguin together cost 666 bucks.*

(c) *The penguin costs 600 bucks more than the parrot.*

(d) ***What is the price of the parrot?***

You may assume that the prizes of these delicacies are integer valued. ◇

Exercise 9: Solve the following text problem using `Z3`.

(a) *A train travels at a uniform speed for 360 miles.*

(b) *The train would have taken 48 minutes less to travel the same distance if it had been faster by 5 miles per hour.*

(c) ***Find the speed of the train!***

Hints:

1. As the speed is a real number you should declare this variable via the `Z3` function `Real` instead of using the function `Int`.

2. 48 minutes are four fifth of an hour. The fraction $\frac{4}{5}$ can be represented in `Z3` by the expression `Q(4, 5)`.

3. When you formulate the information given above, you will get a system of **non-linear** equations, which is equivalent to a quadratic equation. This quadratic equation has two different solutions. One of these solutions is negative. In order to exclude the negative solution you need to add a constraint stating that the speed of the train has to be greater than zero.

4. The solution will be some real number which is represented internally as an object of type `RatNumRef`. If `o` is an object of this type, then this object can be converted to a string as follows:

   ```
   o.as_decimal(17)
   ```

   Here, `17` is the number of digits following the decimal point. This string can be then converted to a float by using the function `float`. ◇

3.7.2 The Knight's Tour

In this subsection we will solve the puzzle The Knight's Tour using `Z3`. This puzzle asks whether it is possible for a knight to visit all 64 squares of the board and return to its starting square in 64 moves. The tour starts in one of the corners of the board.

In order to model this puzzle as a constraint satisfaction problem we first have to decide on the variables that we want to use. The idea is to have 65 variables that describe the position of the knight after its i^{th} move where $i = 0, 1, \cdots, 64$. However, it turns out that it is best to split the values of these positions up into a row and a column. If we do this, we end up with 130 variables of the form

$$\texttt{R}_i \text{ and } \texttt{C}_i \quad \text{for } i \in \{0, 1, \cdots, 64\}.$$

Here $\texttt{R}_i$ denotes the row of the knight after its i^{th} move, while $\texttt{C}_i$ denotes the corresponding column.

Next, we have to formulate the constraints. In this case, there are two kinds of constraints:

1. We have to specify that the move from the position $\langle \texttt{R}_i, \texttt{C}_i \rangle$ to the position $\langle \texttt{R}_{i+1}, \texttt{C}_{i+1} \rangle$ is legal move for a knight. In chess, there are two ways for a knight to move:

   (a) The knight can move two squares horizontally left or right followed by moving vertically one square up or down, or

   (b) the knight can move two squares vertically up or down followed by moving one square left or right.

   Figure 3.26 shows all legal moves of a knight that is positioned in the square `e4`. Therefore, a formula that expresses that the i^{th} move is a legal move of the knight is a disjunction of the following eight formulas that each describe one possible way for the knight to move:

   (a) $\texttt{R}_{i+1} = \texttt{R}_i + 2 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i + 1$,
   (b) $\texttt{R}_{i+1} = \texttt{R}_i + 2 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i - 1$,
   (c) $\texttt{R}_{i+1} = \texttt{R}_i - 2 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i + 1$,
   (d) $\texttt{R}_{i+1} = \texttt{R}_i - 2 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i - 1$,
   (e) $\texttt{R}_{i+1} = \texttt{R}_i + 1 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i + 2$,
   (f) $\texttt{R}_{i+1} = \texttt{R}_i + 1 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i - 2$,
   (g) $\texttt{R}_{i+1} = \texttt{R}_i - 1 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i + 2$,
   (h) $\texttt{R}_{i+1} = \texttt{R}_i - 1 \;\wedge\; \texttt{C}_{i+1} = \texttt{C}_i - 2$.

2. Furthermore, we have to specify that the position $\langle \texttt{R}_i, \texttt{C}_i \rangle$ is different from the position $\langle \texttt{R}_j, \texttt{C}_j \rangle$ if $i \neq j$.

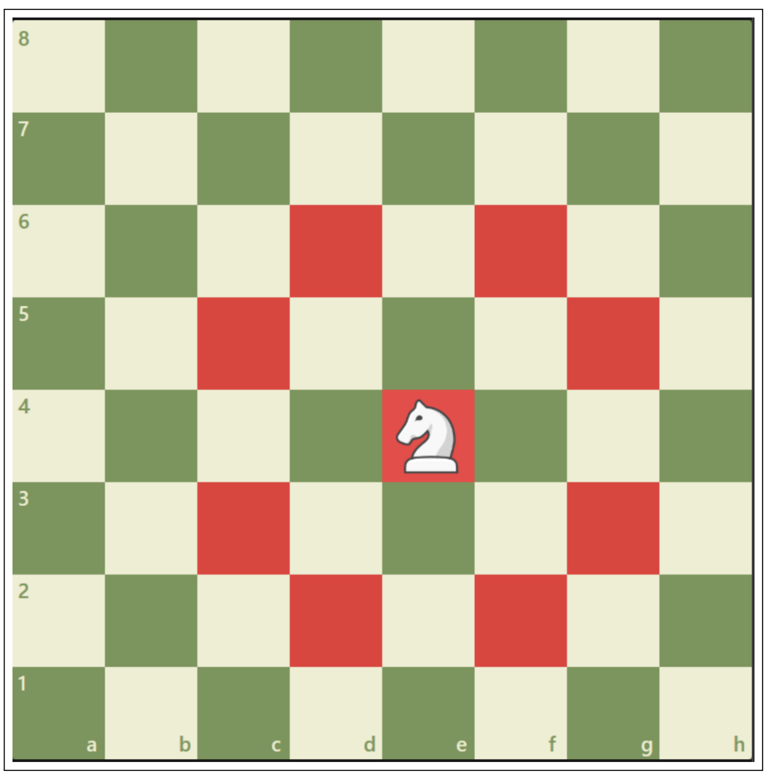


Figure 3.26: The moves of a knight, courtesy of chess.com.

Figure 3.27 shows how we can formulate the puzzle using `Z3`.

```
import * from z3
def row(i): return f'R{i}'
def col(i): return f'C{i}'

def all_variables():
    Variables = set()
    for i in range(64+1):
        Variables.add(row(i))
        Variables.add(col(i))
    return Variables

def is_knight_move(i):
    r  = row(i)
    c  = col(i)
    rX = row(i+1)
    cX = col(i+1)
    Formulas = set()
    for delta_r, delta_c in [(1, 2), (2, 1)]:
        Formulas.add(f'And({rX} == {r} + {delta_r}, {cX} == {c} + {delta_c})')
        Formulas.add(f'And({rX} == {r} + {delta_r}, {cX} + {delta_c} == {c})')
        Formulas.add(f'And({rX} + {delta_r} == {r}, {cX} == {c} + {delta_c})')
        Formulas.add(f'And({rX} + {delta_r} == {r}, {cX} + {delta_c} == {c})')
    return 'Or(' + ', '.join(Formulas) + ')'

def all_different():
    Result = set()
    for i in range(62+1):
        for j in range (i+1, 63+1):
            Result.add(f'Or({row(i)} != {row(j)}, {col(i)} != {col(j)})')
    return Result

def all_constraints():
    Constraints = all_different()
    Constraints.add(f'{row(0 )} == 0')
    Constraints.add(f'{col(0 )} == 0')
    Constraints.add(f'{row(64)} == 0')
    Constraints.add(f'{col(64)} == 0')
    for i in range(63+1):
        Constraints.add(is_knight_move(i))
    for i in range(64+1):
        Constraints.add(f'{row(i)} >= 0')
        Constraints.add(f'{col(i)} >= 0')
    return Constraints
```

Figure 3.27: The Knight's Tour: Computing the constraints.

1. In line 1 we import everything from the library `z3` so that we can write, e.g. `And`(x, y) instead of having to write `z3.And`(x, y).

   The expression `z3.And`(x, y) computes the conjunction of x and y.

2. It is not convenient to declare all of the 130 variables R_i and $\mathtt{C}_i$ for $i = 0, 1, \cdots, 64$ explicitly. Instead, we will write a function that creates and declares these variables. To implement this function, we define the auxiliary functions `row` and `col` in line 2 and 3. Given a natural number i, the expression `row`(i) returns the string `'R`i`'` and `col`(i) returns the string `'C`i`'`. These strings in turn represent the variables $\mathtt{R}_i$ and $\mathtt{C}_i$.

3. The function `all_variables` returns a set of all variable names.

4. The function `is_knight_move` checks whether the move from position i specified as $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ to the position $\langle \mathtt{R}_{i+1}, \mathtt{C}_{i+1} \rangle$ is a legal move for a knight.

5. The function `all_different` computes a set of formulas that state that the positions $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ for $i = 0, , 1 \cdots, 63$ are all different from each other.

6. The function `all_constraints` computes the set of all constraints. In addition to the constraints already discussed this function specifies that the knight starts its tour at the leftmost topmost corner of the board and that the tour also ends in this corner.

   Additionally there are constraints that the variables $\mathtt{R}_i$ and $\mathtt{C}_i$ are all non-negative. These constraints are needed as we will model the variables with bit vectors of length 4. These bit vectors store integers in two's complement representation. In two's complement representation of a bit vector of length 4 we can model integers from the set $\{-8, \cdots, 7\}$. If we add the number 1 to a 4-bit bit vector v that represents the number 7, then an overflow will occur and the result will be -8 instead of 8. This could happen in the additions that are performed in the formulas computed by the function `is_knight_move`. We can exclude these cases by adding the constraints that all variables are non-negative.

Finally, the function `solve` that is shown in Figure 3.28 on page 79 can be used to solve the puzzle. This function takes two arguments:

(a) `Constraints` is a set of strings that are interpreted as `Z3` constraints.

(b) `Variables` is a set of strings that are interpreted as variables.

The purpose of the function `solve` is to find a solution of the given CSP. If successful, it returns a dictionary that maps every variable name to the corresponding value of the solution that has been found.

1. In line 1 we define the dictionary `Environment` which will serve as the local environment for the functions `exec` and `eval` below.

2. We import everything form the package `z3` into this environment in line 3.

3. Then we declare that the strings from the set `Variable` represent `Z3` bit-vector variables of length 4.

4. We create a solver object in line 6 and add the constraints to this solver in the following two lines.

5. The function `check` tries to build a model satisfying the constraints, while the function `model` extracts this model if it exists.

```
1  def solve(Constraints, Variables):
2      Environment = {}
3      exec('import z3', Environment)
4      for v in Variables:
5          exec(f'{v} = z3.BitVec(f"{v}", 4)', Environment)
6      s = z3.Solver()
7      for c in Constraints:
8          s.add(eval(c, Environment))
9      result = str(s.check())
10     if result == 'sat':
11         m = s.model()
12         S = { v: m[eval(v, Environment)] for v in Variables }
13         return S
14     elif result == 'unsat':
15         print('The problem is not solvable.')
16     else:
17         print('Z3 cannot determine whether the problem is solvable.')
```

Figure 3.28: The function `solve`.

6. Finally, in line 12 we create a dictionary that maps all of our variables to the corresponding values that are found in the model. Note that we have to turn the variable names, that are stored as strings in the set `Variables`, into objects that represent the corresponding `Z3` variables using the function `eval`.

   This dictionary is then returned.

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

Table 3.1: A super hard sudoku from the magazine “Zeit Online”.

Exercise 10: Table 3.1 on page 79 shows a sudoku that I have taken from the Zeit Online magazine. Solve this Sudoku using `Z3`. I have written a frame for you to use that can be found at

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/2 Constraint Solver/Sudoku-Z3-Frame.ipynb ◇

3.7.3 Literature

In this chapter we could only give a glimpse of the theory of constraint satisfaction problems. For further details on the theory of CSPs, consult the book Constraint Processing by Rina Dechter [Dec03].

Chapter 4

Playing Games

One major breakthrough for the field of artificial intelligence happened in 1997 when the chess-playing computer Deep Blue was able to beat the World Chess Champion Garry Kasparov by $3^1/_2-2^1/_2$. While Deep Blue was based on special hardware, according to the computer chess rating list of the 2nd of January 2025, the best version of the chess program Stockfish runs on ordinary desktop computers and has an Elo rating of 3642. To compare, according to the Fide list of January 2025, the best human player (and former World Chess Champion) Magnus Carlsen has an Elo rating of 2831. If two players differ by more than 400 in their ELO ranking, the lower ranked player does not stand a chance to win or even draw against the higher ranked player. Hence, Magnus Carlsen wouldn't stand a chance to win or draw a game against Stockfish. In 2017, at the Future of Go Summit, the computer program AlphaGo was able to beat Ke Jie, who was at that time considered to be the best human Go player in the world. Besides Go and chess, there are many other games where today the performance of a computer exceeds the performance of human players. To name just one more example, in 2019 the program Pluribus was able to beat fifteen professional poker players in six-player no-limit Texas Hold'em poker resoundingly.[1]

This chapter is structured as follows:

(a) We define the notion of deterministic two player zero sum games in the next section.

(b) To illustrate this definition we describe the game tic-tac-toe in this framework.

(c) The minimax algorithm is a simple algorithm to play games and is described next.

(d) Alpha-beta pruning is an improvement of the minimax algorithm.

(e) Finally, we consider the case of those games that, due to memory limitations, can not be solved with the pure version of alpha-beta pruning. For these games we discuss depth-limited adversarial search.

4.1 Basic Definitions

In order to investigate how a computer can play a game we define a game $\mathcal{G}$ as a quintuple

$$\mathcal{G} = \langle \texttt{States}, s_0, \texttt{Players}, \texttt{nextStates}, \texttt{utility} \rangle$$

where the components are interpreted as follows:

1. `States` is the set of all possible states of the game.

   We will only consider games where the set `States` is finite.

[1] Well-informed circles report that all 15 professional players had to go home stark naked.

2. $s_0 \in \texttt{States}$ is the start state.

3. `Players` is the list of the players of the game. The first element in `Players` is the player to start the game and after that the players take turns. As we only consider two person games, we assume that `Players` is a list of length two.

4. `nextStates` is a function that takes a state $s \in \texttt{States}$ and a player $p \in \texttt{Players}$ and returns the set of states that can be reached if the player p has to make a move in the state s. Hence, the signature of `nextStates` is given as follows:

$$\texttt{nextStates} : \texttt{States} \times \texttt{Players} \rightarrow 2^{\texttt{States}}.$$

5. `utility` is a function that takes a state s as its argument. If the game is finished, it returns the value that the game has for the first player. Otherways, it returns the undefined value Ω. In general, the value of a game is a real number, but in all of our examples, this value will be an element from the set $\{-1, 0, +1\}$. If $\texttt{utility}(s) = -1$, then the first player has lost the game, if $\texttt{utility}(s) = 1$, then the first player has won the game, and if $\texttt{utility}(s) = 0$, then the game drawn. Hence the signature of `utility` is

$$\texttt{utility} : \texttt{States} \rightarrow \{-1, 0, +1\} \cup \{\Omega\}.$$

If $\mathcal{G} = \langle \texttt{States}, s_0, \texttt{Players}, \texttt{nextStates}, \texttt{utility} \rangle$ is a game, we define an auxiliary function `finished` that takes a state s and decides whether the games is finished. Therefore, the signature of `finished` is

$$\texttt{finished} : \texttt{States} \rightarrow \mathbb{B}.$$

Here, $\mathbb{B}$ is the set of Boolean values, i.e. we have $\mathbb{B} := \{\texttt{true}, \texttt{false}\}$. The definition of `finished` is as follows:

$$\texttt{finished}(s) := \big(\texttt{utility}(s) \neq \Omega\big).$$

Using the function `finished`, we define the set `TerminalStates` as the set of those states such that the game is finished, i.e. we define

$$\texttt{TerminalStates} := \{s \in \texttt{States} \mid \texttt{finished}(s)\}.$$

We will only consider so called two person zero sum games. This means that the list `Players` has exactly two elements. If we call these players `A` and `B`, i.e. if we have

$$\texttt{Players} = \big[\texttt{A}, \texttt{B}\big],$$

then the game is called a zero sum game if `A` has won the game if and only if `B` has lost the game and vice versa. Games like Go, chess, and Checkers are two person zero sum games. We proceed to discuss a simple example.

4.2 Tic-Tac-Toe

The game tic-tac-toe is played on a square board of size 3×3. On every turn, the first player puts an "`X`" on one of the free squares of the board when it is her turn, while the second player puts an "`O`" onto a free square when it is his turn. If the first player manages to place three `X`s in a row, column, or diagonal, she has won the game. Similarly, if the second player manages to put three `O`s in a row, column, or diagonal, he is the winner. Otherwise, the game is drawn. In this section we present two different implementations of tic-tac-toe:

1. We begin with a naive implementation of tic-tac-toe that is easy to understand but has a high memory footprint.

2. After that, we present an implementation that is based on bitboards and has only a fraction of the memory requirements of the naive implementation.

4.2.1 A Naive Implementation of Tic-Tac-Toe

```
gPlayers = [ "X", "O" ]
gStart   = tuple( tuple(" " for col in range(3)) for row in range(3))
def to_list (State): return [list(row) for row in State]
def to_tuple(State): return tuple(tuple(row) for row in State)

def empty(State):
    return [ (row, col) for row in range(3)
                        for col in range(3)
                        if  State[row][col] == ' '
           ]

def next_states(State, player):
    Empty  = empty(State)
    Result = []
    for row, col in Empty:
        NextState           = to_list(State)
        NextState[row][col] = player
        Result.append( to_tuple(NextState) )
    return Result

gAllLines = [ [ (row, col) for col in range(3) ] for row in range(3) ] \
          + [ [ (row, col) for row in range(3) ] for col in range(3) ] \
          + [ [ (idx,   idx) for idx in range(3) ] ]                   \
          + [ [ (idx, 2-idx) for idx in range(3) ] ]

def utility(State):
    for Pairs in gAllLines:
        Marks = { State[row][col] for row, col in Pairs }
        if len(Marks) == 1 and  Marks != { ' ' }:
            return 1 if Marks == { 'X' } else -1
    for row in range(3):
        for col in range(3):
            if State[row][col] == ' ': # the board is not filled
                return None
    return 0
```

Figure 4.1: A *Python* implementation of tic-tac-toe.

Figure 4.1 on page 83 shows a *Python* implementation of tic-tac-toe.

1. The variable `gPlayers` stores the list of players. Traditionally, we use the characters "`X`" and "`O`" to name the players.

2. The variable gStart stores the start state, which is an empty board. States are represented as tuples of tuples. If S is a state and $r, c \in \{0, 1, 2\}$, then $S[r][c]$ is the mark in row r and column c. To represent states we have to use immutable data types, i.e. tuples instead of lists, as we need to store states in sets later. The entries in the inner tuples are the characters "X", "O", and the blank character " ". As the state gStart is the empty board, it is represented as a tuple of three tuples containing three blanks each:

```
( (' ', ' ', ' '),
  (' ', ' ', ' '),
  (' ', ' ', ' ')
).
```

3. As we need to manipulate States, we need a function that converts them into lists of lists. This function is called to_list.

4. We also need to convert the lists of lists back into tuples of tuples. This is achieved by the function to_tuple.

5. Given a state S the function empty(S) returns the list of pairs (row, col) such that S[row][col] is a blank character. These pairs are the coordinates of the fields on the board S that are not yet occupied by either an "X" or an "O".

6. The function next_states takes a State and a player and computes the list of states that can be reached from State if player is to move next. To this end, it first computes the set of empty positions, i.e. those positions that have not yet been marked by either player. Every position is represented as a pair of the form (row, col) where row specifies the row and col specifies the column of the position. The position (row, col) is empty in State iff

   State[row][col] = " ".

   The computation of the empty position has been sourced out to the function empty. The function nextStates then iterates over these empty positions. For every empty position (row, col) it creates a new state NextState that results from the current State by putting the mark of player in this position. The resulting states are collected in the list Result and returned.

   Note that we had to turn the State into a list of list in order to manipulate it. The manipulated State is then cast back into a tuple of tuples.

7. The function utility takes a State as its argument. If the game is finished in the given State, it returns the value that this State has for the player "X". If the outcome of the game is not yet decided, the value None is returned instead.

   In order to achieve its goal, the procedure first computes the set of all sets of coordinate pairs that either specify a horizontal, vertical, or diagonal line on a 3×3 tic-tac-toe board. Concretely, the variable gAllLines has the following value:

$$\Big[\ \big[(0,0),(0,1),(0,2)\big],\ \big[(1,0),(1,1),(1,2)\big],\ \big[(2,0),(2,1),(2,2)\big],$$
$$\big[(0,0),(1,0),(2,0)\big],\ \big[(0,1),(1,1),(2,1)\big],\ \big[(0,2),(1,2),(2,2)\big],$$
$$\big[(0,0),(1,1),(2,2)\big],\ \big[(2,0),(1,1),(0,2)\big]$$
$$\Big]$$

   The first line in this expression gives the sets of pairs defining the rows, the second line defines the columns, and the last line yields the two diagonals. Given a state State and a set Pairs, the set

```
Marks = { State[row][col] :  (row, col) in Pairs }
```

is the set of all marks in the line specified by `Pairs`. For example, if

```
Pairs = { (1, 1), (2, 2), (3, 3) },
```

then `Marks` is the set of marks on the falling diagonal. The game is decided if all entries in the set `Marks` have the value "`X`" or the value "`O`". In this case, the set `Marks` has exactly one element which is different from the blank. If this element is `"X"`, then the game is won by `"X"`, otherwise the element must be `"O"` and hence the game has a value of -1 for `"X"`.

If there are any empty squares on the board, but the game has not yet been decided, then the function returns `None`. Finally, if there are no more empty squares left, the game is a draw.

The implementation shown so far has one important drawback: Every state needs 256 bytes in memory. This can be checked using the *Python* function `sys.getsizeof`. Therefore, we show a leaner implementation next.

4.2.2 A Bitboard-Based Implementation of Tic-Tac-Toe

If we have to reduce the memory requirements of the states, then we can store the states as integers. The first nine bits of these integers store the position of the `X`s, while the next nine bits store the positions of the `O`s. This kind of representation where a state is coded as a series of bit in an integer is known as a bitboard. This is much more efficient than storing states as tuples of tuples of characters. Figure 4.2 on page 86 shows an implementation of tic-tac-toe that is based on a bitboard. We proceed to discuss the details of this implementation.

1. When we use bitboards to implement tic-tac-toe it is more convenient to store the players as numbers. The first player `X` is encoded as the number 0, while the second player `O` is encoded as the number 1.

2. In the state `gStart`, no mark has been placed on the board. Hence all bits are unset and therefore this state is represented by the number 0.

3. The function `set_bits` takes a list of natural numbers as its argument `Bits`. These numbers specify the bits that should be set. It returns an integer where all bits specified in the argument `Bits` are set to 1 and all other bits are set to 0.

4. The function `set_bit` takes a natural number `n` as its argument. It returns a number where the `n`$^{\mathrm{th}}$ bit is set to 1 and all other bits are set to 0.

5. Given a `state` that is represented as a number, the function `empty(state)` returns the set of indexes of those cells such that neither player `X` nor player `O` has placed a mark in the cell.

   Note that there are 9 cells on the board. Each of these cells can hold either an `'X'` or an `'O'`. If the i^{th} cell is marked with a `'X'`, then the i^{th} bit of `state` is set. If instead the i^{th} cell is marked with an `'O'`, then the $(9+i)^{\mathrm{th}}$ bit of `state` is set. If the i^{th} cell is not yet marked, then both the i^{th} bit and the $(9+i)^{\mathrm{th}}$ bit are 0.

6. Given a `state` and the `player` who is next to move, the function `next_states` computes the set of states that can be reached from `state`. Note that player `X` is encoded as the number 0, while player `O` is encoded as the number 1.

7. The global variable `gAllLines` is a list of eight bit masks. These masks can be used to test whether there are three identical marks in a row, column, or diagonal.

```
gPlayers = [0, 1]
gStart = 0

def set_bits(Bits):
    result = 0
    for b in Bits:
        result |= 1 << b
    return result

def next_states(S: State, player: int) -> list[int]:
    Empty = { n for n in range(9)
                if S & ((1 << n) | (1 << (9 + n))) == 0
            }
    return  [ (S | (1 << (player * 9 + n))) for n in Empty ]

gAllLines = [ set_bits([0,1,2]), set_bits([3,4,5]), set_bits([6,7,8]),
              set_bits([0,3,6]), set_bits([1,4,7]), set_bits([2,5,8]),
              set_bits([0,4,8]), set_bits([2,4,6])                     ]

def utility(state):
    for mask in gAllLines:
        if state & mask == mask:
            return 1              # player 'X' has won
        if (state >> 9) & mask == mask:
            return -1             # player 'O' has won
    # 511 == 2**9 - 1 = 0b1_1111_1111
    if (state & 511) | (state >> 9) != 511: # the board is not yet filled
        return None
    return 0 # it's a draw
```

Figure 4.2: Tic-Tac-Toe implemented by a bitboard.

8. The function `utility` takes two arguments:

   (a) `state` is an integer representing the board.

   (b) `player` specifies a player. Here player `X` is encoded as the number 0, while player `O` is encoded as the number 1.

   The function returns 1 if `player` has won the game, -1 if the game is lost for `player`, 0 if it's a draw, and `None` if the game has not yet been decided.

4.3 The Minimax Algorithm

Having defined the notion of a game, our next task is to come up with an algorithm that can play a game. The algorithm that is easiest to implement is the minimax algorithm. This algorithm is based on the notion of the value of a state. Conceptually, the notion of the value of a state is an extension of the notion of the utility of a state. While the utility is only defined for terminal states, the value is defined for all states. Formally, we define a function

$$\texttt{maxValue} : \texttt{States} \rightarrow \{-1, 0, +1\}$$

that takes a state $s \in \texttt{States}$ and returns the value that the state s has for the first player, who tries to maximize the value of the state, provided that both the player p and his opponent play optimally. The easiest way to define this function is via recursion. As the `maxValue` function is an extension of the `utility` function, the base case is as follows:

$$\texttt{finished}(s) \rightarrow \texttt{maxValue}(s) = \texttt{utility}(s). \tag{1}$$

If the game is not yet finished, we define

$$\neg\texttt{finished}(s) \rightarrow \texttt{maxValue}(s) = \max\Bigl(\bigl\{\texttt{minValue}(n) \bigm| n \in \texttt{nextStates}(s, \texttt{gPlayers}[0])\bigr\}\Bigr). \tag{2}$$

The reason is that, if the game is not finished yet, the maximizing player $\texttt{gPlayers}[0]$ has to evaluate all possible moves. From these, the player will choose the move that maximizes the value of the game for herself. In order to do so, the player computes the set $\texttt{nextStates}(s, \texttt{gPlayers}[0])$ of all states that can be reached from the state s in any one move of the player $\texttt{gPlayers}[0]$. Now if n is a state that results from player $\texttt{gPlayers}[0]$ making some move, then in state n it is the turn of the other player $\texttt{gPlayers}[1]$ to make a move. However, this player is the minimizing player who tries to achieve the state with the minimal value. Hence, in order to evaluate the state n, we have to call the function `minValue` recursively as $\texttt{minValue}(n)$. The function `minValue` has the same signature as `maxValue` and is defined by the following recursive equations

1. $\texttt{finished}(s) \rightarrow \texttt{minValue}(s) = \texttt{utility}(s)$.
2. $\neg\texttt{finished}(s) \rightarrow \texttt{minValue}(s) = \min\Bigl(\bigl\{\texttt{maxValue}(n) \bigm| n \in \texttt{nextStates}(s, \texttt{gPlayers}[1])\bigr\}\Bigr)$.

In the future we will sometimes speak of the `value` function. This name is used as a synonym for the function `maxValue`.

Figure 4.3 on page 88 shows an implementation of the functions `maxValue` and `minValue`. It also shows the function `best_move`. This function takes a `State` such that `X` is to move in this state. It returns a pair (v, s) where s is a state that is optimal for the player `X` and such that s can be reached in one step from `State`. Furthermore, v is the value of this state.

(a) To this end, it first computes the set `NS` of all states that can be reached from the given `State` in one step if `X` is to move next.

(b) `bestValue` is the best value that `X` can achieve in the given `State`.

(c) `BestMoves` is the set of states that `X` can move to and that are optimal for her.

(d) The function returns randomly one of those states $\texttt{ns} \in \texttt{NS}$ such that the value of `ns` is optimal, i.e. is equal to `bestValue`. We use randomization here since we want to have more interesting games. If we would always choose the first state that achieves the best value, then our program would always make the same move in a given state. Hence, playing the program would get boring much sooner.

```
def maxValue(State):
    if finished(State):
        return utility(State)
    return max([ minValue(ns) for ns in next_states(State, gPlayers[0]) ])

def minValue(State):
    if finished(State):
        return utility(State)
    return min([ maxValue(ns) for ns in next_states(State, gPlayers[1]) ])

def best_move(State):
    NS        = next_states(State, gPlayers[0])
    bestVal   = maxValue(State)
    BestMoves = [s for s in NS if minValue(s) == bestVal]
    BestState = random.choice(BestMoves)
    return bestVal, BestState
```

Figure 4.3: The Minimax algorithm.

4.3.1 Memoization

Let us consider how many states have to be explored in the case of tic-tac-toe by the minimax algorithm described previously. We have 9 possible moves for player `X` in the start state, then the player `O` can respond with 8 moves, then there are 7 moves for player `O` and so on until in the end player `X` has only 1 move left. If we disregard the fact that some games are decided after fewer than 9 moves, the functions `maxValue` and `minValue` need to consider a total of

$$9 \cdot 8 \cdot 7 \cdot \ldots \cdot 2 \cdot 1 = 9! = 362\,880$$

different moves. However, if we count the number of possibilities of putting 5 `O`s and 4 `X`s on a 3×3 board, we see that there are only

$$\binom{9}{5} = \frac{9!}{5! \cdot 4!} = \frac{9 \cdot 8 \cdot 7 \cdot 6}{1 \cdot 2 \cdot 3 \cdot 4} = 9 \cdot 2 \cdot 7 = 126$$

possibilities, because we only have to count the number of ways there are to put 5 `O`s on 9 different positions and that number is the same as the number of subsets of five elements from a set of 9 elements. Therefore, if we disregard the fact that some games are decided after fewer than nine moves, there are a factor of $5! \cdot 4! = 2880$ less terminal states than there are possible sequences of moves!

As we have to evaluate not just terminal states but all states, the saving is actually a bit smaller than 2880. The next exercise explores this in more detail.

We can use memoization to exploit the fact that the number of states is much smaller than the number of possible game sequences. Figure 4.4 on page 89 shows how this can be implemented.

```
gCache = {}

def memoize(f):
    global gCache

    def f_memoized(*args):
        if args in gCache:
            return gCache[args]
        result = f(*args)
        gCache[args] = result
        return result

    return f_memoized

maxValue = memoize(maxValue)
minValue = memoize(minValue)
```

Figure 4.4: Memoization.

1. `gCache` is a dictionary that is initially empty. This dictionary is used as a memory cache by the function `memoize`.

2. The function `memoize` is a second order function that takes a function f as its argument. It creates a memoized version of the function f: This memoized version of f, which is called `f_memoized`, first tries to retrieve the value of f from the dictionary `gCache`. If this is successful, the cached value is returned. Otherwise, the function f is called to compute the result. This result is then stored in the dictionary `gCache` before it is returned, as the result of the function `f_memoized`.

   In turn, the function `memoize` returns the function `f_memoized`, which is the memoized version of f.

3. In order to use memoization for the minimax algorithm, all that needs to be done is to memoize both the functions `maxValue` and `minValue`. These functions can share the same dictionary `gCache` because `maxValue` is only called for states where `X` has to make the next move, while `minValue` is only called for states where `O` has to make the next move. If this wouldn't be the case, the name of the function would have to be stored in `gCache` also.

```
def play_game(canvas):
    State = gStart
    while True:
        val, State = best_move(State);
        draw(State, canvas, f'For me, the game has the value {val}.')
        if finished(State):
            final_msg(State)
            return
        IPython.display.clear_output(wait=True)
        State = get_move(State)
        draw(State, canvas, '')
        if finished(State):
            IPython.display.clear_output(wait=True)
            final_msg(State)
            return
```

Figure 4.5: The function `play_game`.

Figure 4.5 on page 90 presents the implementation of the function `play_game` that is used to play a game.

1. Initially, `State` is the `startState`.
2. As long as the game is not finished, the procedure keeps running.
3. We assume that the computer goes first.
4. The function `best_move` is used to compute the move of the computer. This resulting state is then displayed.
5. After that, it is checked whether the game is finished.
6. If the game is not yet finished, the user is asked to make his move via the function `get_move`. The state resulting from this move is then returned and displayed.
7. Next, we have to check whether the game is finished after the move of the user has been executed.

In order to better understand the reason for using memoization in the implementation of the functions `maxValue` and `minValue` we introduce the following notion.

Definition 6 (Game Tree) Assume that

$$\mathcal{G} = \langle \texttt{States}, s_0, \texttt{Players}, \texttt{nextStates}, \texttt{utility} \rangle$$

is a game. Then a play of length n is a list of states of the form $[s_0, s_1, \cdots, s_n]$ such that

$$s_o = \texttt{Start} \quad \text{and} \quad \forall i \in \{0, \cdots, n-1\} : s_{i+1} \in \texttt{nextStates}(s_i, p_i),$$

where the players p_i are defined as follows:

$$p_i := \begin{cases} \texttt{Players[0]} & \text{if } i \,\%\, 2 = 0; \\ \texttt{Players[1]} & \text{if } i \,\%\, 2 = 1. \end{cases}$$

Therefore, p_i is the first element of the list `Players` if i is even and p_i is the second element of this list if i is odd. The game tree of the game $\mathcal{G}$ is the set of all possible plays. ◇

The following exercise shows why memoization is so important.

Exercise 11: In simplified tic-tac-toe the game only ends when there are no more empty squares left. The player `X` wins if she has more rows, columns, or diagonals of three `X`s than the player `O` has rows, columns, or diagonals of three `O`s. Similarly, the player `O` wins if he has more rows, columns, or diagonals of three `O`s than the player `X` has rows, columns, or diagonals of three `X`s. Otherwise, the game is a draw.

(a) Derive a formula to compute the size of the game tree of simplified tic-tac-toe.

(b) Write a short program to evaluate the formula derived in part (a) of this exercise.

(c) Derive a formula that gives the number of all states of simplified tic-tac-toe.

Notice that this question does not ask for the number of all terminal states but rather asks for all states.

(d) Evaluate the formula derived in part (c) of this exercise.

Hint: You don't have to do the calculation in your head. ◇

4.4 Alpha-Beta Pruning

In this section we discuss α-β-pruning. This is a search technique that can prune large numbers of the search space and thereby increase the efficiency of a game playing program. Figure 4.6 gives a first idea of what α-β-pruning is about. The figure shows the game tree of a game that is finished after four moves, i.e. both players are able to make two moves, that is both players can take it in turns to make two moves. After the second player has made her second move, the game is decided. Contrary to our previous definition, the values of the game are not just elements form the set $\{-1, 0, +1\}$ but instead are natural numbers. The first player, called `Max` has the goal of achieving a big number, while the second player `Min` has the goal to achieve a small number.

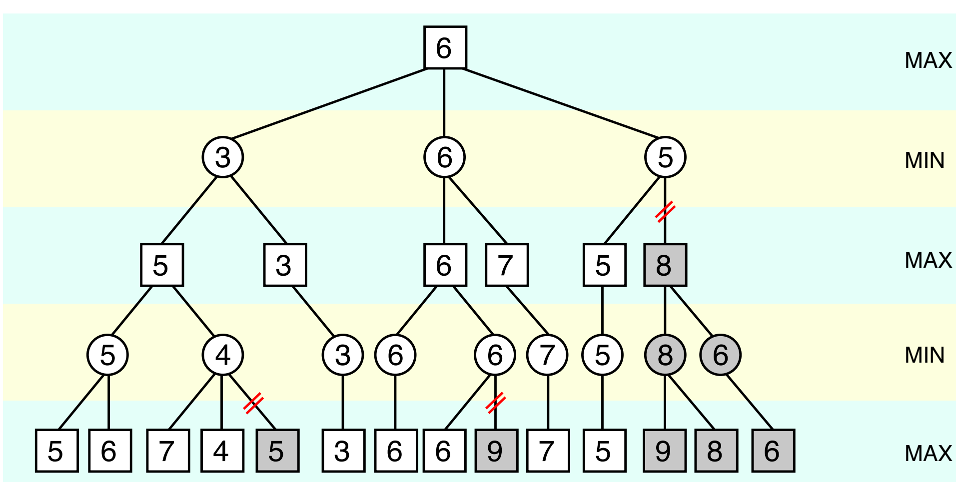


Figure 4.6: Example game tree showing α-β-Pruning. (Original: Wikipedia.)

If we look at the game tree in Figure 4.6 on page 91 we notice that some numbers are greyed out. The reason is that these numbers can not influence the result of the game. If we would replace these numbers with any arbitrary numbers, the overall result of the game would not change if both players play optimally.

1. Let us first take a look at the greyed out node that is marked with a 5 in the bottom-most line. This is the result value of a move from the player `Min`. The parent of this node is marked with a 4. In this node it is the turn of the player `Min`, who will choose the smallest possible value. If the 5 would be replaced by a 0, then `Min` would choose this zero. This would change the node of the parent node, which currently is 4, to be 0. However, this would not change the result of the game because in the parent of the node that is currently marked with a 4 it is the turn of `Max` and `Max` will choose the move leading to his left child because that already guarantees him the value 5.

   Similarly, if the value of the greyed out node was 10 instead of 5, the player `Min` would not choose this node because for her the node marked with 4 is a better choice. Hence the value of the node currently marked with a 5 is of no consequence for the overall value of the game and therefore there is no need to evaluate this node at all.

2. The situation is similar for the first node marked with a greyed out 9 in the bottom-most line, because in the grandparent of this node it is the turn of the player `Max` and `Max` is already guaranteed the value of 6 at this node when he chooses the move leading to its left child.

3. At the right part of the tree we have a whole subtree that is greyed out. The root of this subtree is marked with the number 8. Let us understand the reason this subtree does not influence the value of the overall tree.

   (a) The left sibling of the root node of the greyed out tree is marked with a 5. At the parent of this node, it is the turn of the player `Min`. As `Min` can already choose the node labelled with a 5, we know that the parent node has a value that is at most five.

   (b) However, the grandparent of this node is the root of the complete tree. In this node, it is the turn of the player `Max`. `Max` can achieve the value 6 by choosing his second child. Therefore the value of this node at least 6.

   (c) As we have already seen that the value of the rightmost child of the root node can be at most 5, `Max` will never choose this child, but will choose his second child instead.

   Therefore, the subtree whoose root is marked with a greyed out 8 does not influence the outcome of the game.

To implement α-β-pruning, the basic idea is to provide two additional arguments to the functions `maxValue` and `minValue`. Traditionally, these arguments are called α and β. In order to be able to distinguish between the old functions `maxValue` and `minValue` and its improved version, we call the improved versions `alphaBetaMax` and `alphaBetaMin`. The idea is that these functions are related by the following requirements:

1. As long as $\texttt{maxValue}(s)$ is between α and β, the function `alphaBetaMax` computes the same result as the function `maxValue`, i.e. we have

   $$\alpha \leq \texttt{maxValue}(s) \leq \beta \;\rightarrow\; \texttt{alphaBetaMax}(s, \alpha, \beta) = \texttt{maxValue}(s).$$

2. If $\texttt{maxValue}(s) < \alpha$, we require that the value returned by `alphaBetaMax` is less than or equal to α, i.e. we have

   $$\texttt{maxValue}(s) < \alpha \;\rightarrow\; \texttt{alphaBetaMax}(s, \alpha, \beta) \leq \alpha.$$

3. Similarly, if $\texttt{maxValue}(s) > \beta$, we require that the value returned by `alphaBetaMax` is bigger than or equal to β, i.e. we have

   $$\beta < \texttt{maxValue}(s) \;\rightarrow\; \beta \leq \texttt{alphaBetaMax}(s, \alpha, \beta).$$

Similar to the way that the function `maxValue` is approximated by the function `alphaBetaMax`, the function `minValue` is approximated by the function `alphaBetaMin`. We have:

1. $\alpha \leq \texttt{minValue}(s) \leq \beta \;\rightarrow\; \texttt{alphaBetaMin}(s, \alpha, \beta) = \texttt{minValue}(s)$.

2. $\texttt{minValue}(s) < \alpha \;\rightarrow\; \texttt{alphaBetaMin}(s, \alpha, \beta) \leq \alpha$.

3. $\beta < \texttt{minValue}(s) \;\rightarrow\; \beta \leq \texttt{alphaBetaMin}(s, \alpha, \beta)$.

Although $\texttt{alphaBetaMax}(s)$and $\texttt{alphaBetaMin}(s)$ are only approximations of $\texttt{maxValue}(s)$ and $\texttt{minValue}(s)$, it turns out that these approximations are all that is needed. Once the function `alphaBetaMax` is implemented, the function `maxValue` can then be computed as

$$\texttt{maxValue}(s) := \texttt{alphaBetaMax}(s, -1, +1).$$

The reason is that we already know that $-1 \leq \texttt{maxValue}(s) \leq +1$ and hence the first case of the specification of `alphaBetaMax` guarantees that the equation

$$\texttt{maxValue}(s) = \texttt{alphaBetaMax}(s, -1, +1)$$

holds. Similarly, the function `minValue` can be computed as

$$\texttt{minValue}(s) := \texttt{alphaBetaMin}(s, -1, +1).$$

Figure 4.7 on page 94 shows an implementation of the functions `alphaBetaMax` and `alphaBetaMin` that satisfies the specification given above. Since `alphaBetaMax` and `alphaBetaMin` are implemented as mutually recursive functions, the fact that the implementations of `alphaBetaMax` and `alphaBetaMin` satisfy the specifications given above can be established by computational induction. We proceed perform this proof. The first proof of the correctness of α-β-pruning has been given by Donald E. Knuth and Ronald W. Moore [KM75].
We proceed to discuss the implementation of the function `alphaBetaMax`, which is shown in Figure 4.7 on page 94.

1. If `State` is a terminal state, the function returns the `utility` of the given `State`.

2. We iterate over all successor states $\texttt{ns} \in \texttt{next\_states}(\texttt{State}, \texttt{gPlayers}[0])$.

3. We have to recursively evaluate the states `ns` with respect to the minimizing player `gPlayers[1]`. Hence we call the function `alphaBetaMin` when evaluating the state `ns`.

4. As the specification of `alphaBetaMax` asks us to compute the value of `State` only in those cases where it is less than or equal to `beta`, once we find a successor state s that has a `value` that is at least as big as `beta` we can stop any further evaluation of the successor states and return `value`.

   This shortcut results in significant savings of computation time!

5. Once we have found a successor state that has a value `value` greater than `alpha`, we can increase `alpha` to `value`. The reason is, that once we know we can achieve `value` we are no longer interested in any smaller values. This is the reason for assigning the maximum of `value` and `alpha` to `alpha`.

   After this assignment, `alpha` will be at least as big as `value` and according to the specification of `alphaBetaMax` we can therefore return `alpha`.

```
def alphaBetaMax(State, alpha, beta):
    if finished(State):
        return utility(State)
    for ns in next_states(State, gPlayers[0]):
        value = alphaBetaMin(ns, alpha, beta)
        if value >= beta:
            return value
        alpha = max(alpha, value)
    return alpha

def alphaBetaMin(State, alpha, beta):
    if finished(State):
        return utility(State)
    for ns in next_states(State, gPlayers[1]):
        value = alphaBetaMax(ns, alpha, beta)
        if value <= alpha:
            return value
        beta = min(beta, value)
    return beta
```

Figure 4.7: α-β-Pruning.

Claim: The functions `alphaBetaMax` and `alphaBetaMin` both satisfy the specification given previously.

Proof: The proof proceeds by computational induction. However, we will only show that the function `alphaBetaMax` satisfies its specification, as the proof of the correctness of the function `alphaBetaMin` is completely analogous to the proof of the correctness of `alphaBetaMax`. Furthermore, to simplify the proof we define $\texttt{A} := \texttt{gPlayers}[0]$ and $\texttt{B} := \texttt{gPlayers}[1]$.

B.C: $S \in \texttt{TerminalStates}$, that is $\texttt{finished}(S) = \texttt{True}$.

In this case we have $\texttt{alphaBetaMax}(S, \alpha, \beta) = \texttt{utility}(S)$ and since in this case we also have $\texttt{maxValue}(S) = \texttt{utility}(S)$ this implies

$$\texttt{alphaBetaMax}(s, \alpha, \beta) = \texttt{maxValue}(S). \qquad (*)$$

We have to check the three cases of our specification one by one.

(a) $\alpha \leq \texttt{maxValue}(S) \leq \beta$.
We have to show that $\texttt{alphaBetaMax}(s, \alpha, \beta) = \texttt{maxValue}(S)$ holds. This follows immediately from $(*)$.

(b) $\texttt{maxValue}(S) < \alpha$. Because of $(*)$ this implies

$$\texttt{alphaBetaMax}(s, \alpha, \beta) < \alpha$$

and therefore we also have $\texttt{alphaBetaMax}(S) \leq \alpha$.

(c) $\beta < \texttt{maxValue}(S)$. Because of $(*)$ this implies

$$\beta < \texttt{alphaBetaMax}(s, \alpha, \beta)$$

and therefore we also have $\beta \leq \texttt{alphaBetaMax}(S, \alpha, \beta)$.

This finishes the proof of the base case.

I.S.: This time $S \notin$ `TerminalStates`. As we have to check all three cases of our specification, we have to carry out a case distinction.

(a) $\alpha \leq \texttt{maxValue}(S) \leq \beta$.

We have to show that in this case $\texttt{alphaBetaMax}(S) = \texttt{maxValue}(S)$. There are two subcases.

i. If $\texttt{maxValue}(S) = \beta$ there has to be a node $\texttt{ns} \in \texttt{nextStates}(S, A)$ such that

$$\texttt{minValue}(\texttt{ns}) = \beta.$$

When the `for`-loop hits this node, the condition of the `if`-statement in line 6 is true and the value of this node, which is β, is returned. Therefore we have

$$\texttt{alphaBetaMax}(S, \alpha, \beta) = \beta = \texttt{maxValue}(S).$$

ii. If $\texttt{maxValue}(S) < \beta$, then we know that for all states $\texttt{ns} \in \texttt{nextStates}(S, \texttt{B})$ we must have

$$\texttt{minValue}(\texttt{ns}) < \beta.$$

From the specification and the induction hypothesis it the follows that

$$\texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta) < \beta.$$

Furthermore, since $\alpha \leq \texttt{maxValue}(S)$ there must exists a state $\texttt{ns} \in \texttt{nextStates}(S, \texttt{A})$ such that

$$\alpha \leq \texttt{minValue}(\texttt{ns})$$

W.l.o.g. assume that `ns` is the node with the maximal value, i.e. we have

$$\texttt{maxValue}(S) = \texttt{minValue}(\texttt{ns}).$$

As we already know that $\texttt{minValue}(\texttt{ns}) < \beta$, the induction hypothesis tells us that

$$\texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta) = \texttt{minValue}(\texttt{ns})$$

Hence the value computed in the `for`-loop is equal to $\texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta)$ and therefore we have

$$\texttt{alphaBetaMax}(S, \alpha, \beta) = \texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta) = \texttt{minValue}(\texttt{ns}) = \texttt{maxValue}(S).$$

1. $\texttt{maxValue}(S) < \alpha$

In this case we only have to show that $\texttt{alphaBetaMax}(S, \alpha, \beta) \leq \alpha$. Since $\texttt{maxValue}(S) < \alpha$ it must be the case that

$$\texttt{minValue}(\texttt{ns}) < \alpha \quad \text{for all } \texttt{ns} \in \texttt{nextStates}(S).$$

By induction hypothesis this implies

$$\texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta) \leq \alpha \quad \text{for all } \texttt{ns} \in \texttt{nextStates}(S).$$

Since the `for`-loop computes the maximum of these values it follows that

$$\texttt{alphaBetaMax}(S, \alpha, \beta) \leq \alpha.$$

2. $\beta < \texttt{maxValue}(S)$

In this case we have to show that $\beta \leq \texttt{alphaBetaMax}(s, \alpha, \beta)$. The assumption $\beta < \texttt{maxValue}(S)$ implies that there is node $\texttt{ns} \in \texttt{nextStates}(S, \texttt{A})$ such that

$$\beta < \texttt{minValue}(\texttt{ns}).$$

By induction hypothesis we can conclude that

$$\beta \le \texttt{alphaBetaMin}(\texttt{ns}, \alpha, \beta).$$

But then this value (or another one greater or equal than β) is returned by the `if`-statement in line 6 and hence we have

$$\beta \le \texttt{alphaBetaMax}(S, \alpha, \beta).$$

As the function `alphaBetaMax` satisfies its specification in all three cases, the proof is complete. □

Remark: There is a nice simulator for alpha-beta-pruning available at the following web address: https://pascscha.ch/info2/abTreePractice/.

Exercise 12: The game Nim works as follows:

(a) There are four rows of matches:

1. the first row contains 1 match,
2. the second row contains 3 matches,
3. the third row contains 5 matches, and
4. the fourth row contains 7 matches.

(b) The player whose turn it is first selects a line.

Then she removes any number of matches from this line.

(c) The player that removes the last match has won the game.

Implement this game by adapting the notebook

Artificial-Intelligence/blob/master/Python/3 Games/Nim-Frame.ipynb.

Then, test the game using the notebook

Artificial-Intelligence/blob/master/Python/3 Games/5-Alpha-Beta-Pruning.ipynb. ⋄

4.4.1 Alpha-Beta Pruning with Memoization

Adding memoization to the functions `maxValue` and `minValue` is non-trivial. If memoization is added in a naive way, then the cache might have many entries for the same state that differ only in their values for the parameters α and β. Although this is not a problem for trivial games like Tic-Tac-Toe, it becomes a problem once we try to implement more complex games like Connect Four. The reason is that for those games we are no longer able to compute the complete game tree. Instead, we need to approximate the value of a state with the help of a heuristic. Then, α and β will no longer be confined to the values from the set $\{-1, 0, 1\}$ but will rather take on a continuous set of values from the interval $[-1, +1]$. Hence there will be many function calls of the form

$$\texttt{maxValue}(s, \alpha, \beta)$$

where the state s is the same but α and β are different. If we would try to store every combination of s, α, and β we would waste a lot of memory and, furthermore, we would have only a small number of cache hits. Therefore, we will now present a more effective way to cache the functions `maxValue` and `minValue`. The method we describe here is an adaption of the method published by Marsland and Campbell [MC82]. To this end we will define a function `evaluate` that is called as follows:

$$\texttt{evaluate}(s, f, \alpha, \beta)$$

where the parameters are interpreted as follows:

(a) s is a state that is to be evaluated.

(b) f is either the function `alphaBetaMax` or the function `alphaBetaMin`. If in state s the first player has to move, then $f = $ `alphaBetaMax`, otherwise we have $f = $ `alphaBetaMin`.

(c) We interpret the parameters α and β in the same way as we did when we used them with the functions `alphaBetaMax` and `alphaBetaMin`.

The function `evaluate` encapsulates calls to the functions `alphaBetaMax` and `alphaBetaMin`. Given a state s, a function f, and the values of α and β, it first checks whether the value of $f(s, \alpha, \beta)$ has already been computed and is stored in the cache. If this is the case, the value is returned. Otherwise, the function f is called.

The function `evaluate` makes use of a global variable `gCache`. This variable is used as a cache to store the results of the function `evaluate`. This cache is implemented as a dictionary. The keys of this dictionary are the just the states, not triples of the form $\langle s, \alpha, \beta\rangle$. The values stored in the cache are pairs of the form $(\texttt{flag}, v)$, where v is a value computed by the function $\texttt{evaluate}(s, f, \alpha, \beta)$, while `flag` specifies whether v is exact, a lower bound, or an upper bound. We have

$$\texttt{flag} \in \{\text{'}\leq\text{'}, \text{'}=\text{'}, \text{'}\geq\text{'}\}.$$

The cache satisfies the following specification:

1. $\texttt{gCache}[s] = (\text{'}=\text{'}, v) \rightarrow f(s, \alpha, \beta) = v$

   If the flag is equal to '=', then the value stored in $\texttt{gCache}[s]$ is the exact value computed for the given state s by the function f.

2. $\texttt{gCache}[s] = (\text{'}\leq\text{'}, v) \rightarrow f(s, \alpha, \beta) \leq v$

   If the flag is equal to '≤', then the value stored in $\texttt{gCache}[s]$ is an upper bound for the value returned from $f(s, \alpha, \beta)$.

3. $\texttt{gCache}[s] = (\text{'}\geq\text{'}, v) \rightarrow f(s, \alpha, \beta) \geq v$

   If the flag is equal to '≥', then the value stored in `gCache[State]` is a lower bound for $f(s, \alpha, \beta)$.

If $\texttt{gCache}[s]$ is defined, then the computation of $\texttt{evaluate}(s, f, \alpha, \beta)$ proceeds according to the following case distinction:

1. If the stored value v is exact, we can return this value:

$$\texttt{gCache}[s] = (\text{'}=\text{'}, v) \rightarrow \texttt{evaluate}(s, f, \alpha, \beta) = v.$$

2. If the stored value v is an upper bound, then there are two cases.

   (a) If this upper bound v is less or equal than α, then we know that the true value of the state s is less or equal than α and hence we can also return the value v:

$$\texttt{gCache}[s] = (\text{'}\leq\text{'}, v) \wedge v \leq \alpha \rightarrow \texttt{evaluate}(s, f, \alpha, \beta) = v.$$

   (b) Otherwise we can sharpen the upper bound β by setting β to be the minimum of β and v:

$$\texttt{beta} = \texttt{min}(\texttt{beta}, v).$$

   If this leads to a reduction of β, then size of the interval $[\alpha, \beta]$ is reduced and hence Alpha-Beta pruning will be able to remove more nodes from the game tree, making the search more efficient.

3. If the stored value v is a lower bound, there are again two cases.

(a) If this lower bound is greater or equal than β, then we know that the true value is bigger or equal than β and hence we can return the value v:

$$\texttt{gCache}[s] = (\text{'}\geq\text{'}, v) \wedge \beta \leq v \rightarrow \texttt{evaluate}(s, f, \alpha, \beta) = v.$$

(b) Otherwise, we can sharpen the lower bound α by setting α to be the maximum of α and v:

$$\texttt{alpha} = \texttt{max}(\texttt{alpha}, \text{v})$$

If this leads to an increase of α, then size of the interval $[\alpha, \beta]$ is reduced and hence Alpha-Beta pruning will be able to remove more nodes from the game tree, making the search more efficient.

Finally, we call the function f on the given state with lower bound α and upper bound β and store the value that is computed in the cache via the function `store_cache`. This function has to store both the value and the appropriate flag.

```python
def evaluate(State, f, alpha=-1, beta=1):
    global gCache
    if State in gCache:
        flag, v = gCache[State]
        if flag == '=':
            return v
        if flag == '<=':
            if v <= alpha:
                return v
            else:
                beta = min(beta, v)
        if flag == '>=':
            if beta <= v:
                return v
            else:
                alpha = max(alpha, v)
    v = f(State, alpha, beta)
    store_cache(State, alpha, beta, v)
    return v

def store_cache(State, alpha, beta, v):
    global gCache
    if   v <= alpha:
        gCache[State] = ('<=', v)
    elif v <  beta: # alpha < v
        gCache[State] = ('=', v)
    else: # beta <= v
        gCache[State] = ('>=', v)
```

Figure 4.8: Implementation of the function `evaluate`.

```
def maxValue(State, alpha, beta):
    if finished(State):
        return utility(State)
    for ns in next_states(State, gPlayers[0]):
        value = evaluate(ns, minValue, alpha, beta)
        if value >= beta:
            return value
        alpha = max(alpha, value)
    return alpha

def minValue(State, alpha, beta):
    if finished(State):
        return utility(State)
    for ns in next_states(State, gPlayers[1]):
        value = evaluate(ns, maxValue, alpha, beta)
        if value <= alpha:
            return value
        beta = min(beta, value)
    return beta
```

Figure 4.9: Cached implementation of the functions `alphaBetaMax` and `alphaBetaMin`.

4.5 Progressive Deepening

In practice, most games are far too complex to be evaluated completely, i.e. the size of the set `States` is so big that even the fastest computer does not stand a chance to explore this set completely. For example, it is believed[2] that in chess there are about $4.48 \cdot 10^{44}$ different states that could occur in a game. Hence, it is impossible to explore all possible states in chess. Instead, we have to limit the exploration in a way that is similar to the way professional players evaluate their games: Usually, a player considers all variations of a game for, say, the next three moves. After a given number of moves, the value of a position is estimated using an evaluation heuristic. This function approximates the true value of a given state via a heuristic.

```
def pd_evaluate(State, limit, f=maxValue):
    for l in range(limit+1):
        value = evaluate(State, l, f)
        if value in [-1, 1]:
            return value
    return value
```

Figure 4.10: Progressive Deepening

[2]For reference, compare the wikipedia article on the so-called Shannon number. The Shannon number estimates that there are at least 10^{120} different plays in chess. However, the number of states is estimated to be about $(4.48 \pm 0.37) \cdot 10^{44}$.

In order to implement this idea, we add a parameter `limit` to the procedures `alphaBetaMax` and `alphaBetaMin` that were shown in the previous section. On every recursive invocation of the functions `alphaBetaMax` and `alphaBetaMin`, the parameter `limit` is decreased. Once the limit reaches 0, instead of invoking the function `alphaBetaMax` or `alphaBetaMin` again, we try to estimate the value of the given `State` using an evaluation heuristic. This leads to the code shown in Figure 4.11 on page 102.

When we compare this Figure with Figure 4.7 on page 94, the only difference is in line 4 where we test whether the `limit` is 0. In this case, instead of trying to recursively evaluate the states reachable from `State`, we evaluate the `State` with a `heuristic` function that tries to guess the approximate value of a given state. Notice that in the calls of the function `evaluate` we have to take care to decrease the parameter `limit`. The function `evaluate` is responsible for administering the cache as previously.

There is one further difference between the functions `maxValue` and `minValue` shown in Figure 4.11 and those versions of these functions that were shown previously: In Figure 4.11 the `NextStates` are stored in a priority queue such that the move that is considered to be the best has the highest priority. This way, the best moves are tried first and as a result alpha-beta-pruning is able to prune larger parts of the search space. In order to guess which move is best we use the cached values of the corresponding states. This is the real reason for using progressive deepening: When we evaluate the states with a depth limit of l, we can use the values of the states that has been stored previously when those states were evaluated with a depth limit of $l-2$. At this point the reader might be surprised: Wouldn't the value computed for a depth limit of $l-1$ be more accurate than the value for a depth limit of $l-2$? The answer is no. This can be both verified experimentally and explained theoretically. To understand why the value for the depth of $l-2$ is better than the value for $l-1$, let us think about the game of chess. Assume first that we have a depth limit of 1, i.e. we look only one half move into the future. This would result in very aggressive play, i.e. the computer would always try to capture a piece if possible. For example, if the only capture possible would be the queen capturing a pawn, the computer would take this pawn, even if it would loose the queen in the following move because a look ahead depth of 1 is just not sufficient. With a depth limit of 2 the computer would play more defensive. However, when the depth is incremented to 3, the computer would play more aggressive again. Although it would then not sacrifice a queen to capture a pawn, it still would prepare to capture a pawn with the queen in its second move. For this reason, it is usually best to have a depth limit that is even, because if the depth limit is odd, the computer would play too aggressive as it would not be able to see the way in which his last move is answered by its opponent. Therefore, the values stored for a depth limit of $l-2$ are actually better than those for a depth limit of $l-1$.

For a game like tic-tac-toe it is difficult to come up with a decent heuristic. A very crude approach would be to define:

```
heuristic := [State, player] |-> 0;
```

This heuristic would simply estimate the value of all states to be 0. As this heuristic is only called after it has been tested that the game has not yet been decided, this approach is not utterly unreasonable. For a more complex game like chess, the heuristic could instead be a weighted count of all pieces. Concretely, the algorithm for estimating the value of a state would work as follows:

1. Initially, the variable `sum` is set to 0:

   ```
   sum := 0;
   ```

2. We would count the number of white rooks $\texttt{Rook}_{\text{white}}$ and black rooks $\texttt{Rook}_{\text{black}}$, subtract these numbers from each other and multiply the difference by 5. The resulting number would be added to `sum`:

   $$\texttt{sum += } (\texttt{Rook}_{\text{white}} - \texttt{Rook}_{\text{black}}) \cdot 5;$$

3. We would count the number of white bishops $\texttt{Bishop}_{\text{white}}$ and black bishops $\texttt{Bishop}_{\text{black}}$, subtract these numbers from each other and multiply the difference by 3. The resulting number would be added to `sum`:

$$\texttt{sum += } (\texttt{Bishop}_{\text{white}} - \texttt{Bishop}_{\text{black}}) \cdot 3\texttt{;}$$

4. In a similar way we would count knights, queens, and pawns. Approximately, the weights of knights are 3, a queen is worth 9 and a pawn is worth 1.

The resulting `sum` can then be used as an approximation of the value of a state. More details about the weights of the pieces can be found in the Wikipedia article "chess piece relative value".

I have tested the program described so far with the game Connect Four. You can play this game online at

`https://connect4.gamesolver.org/en/`

The implementation can be found at:
Artificial-Intelligence/blob/master/Python/3 Games/Connect-Four.ipynb

The heuristic that I have implemented uses triples. A triple is defined as a set of three marks of either `X`s or `O`s in a row that is followed by a blank space. The blank space could also be between the marks. Now if there is a state s that has a triples of `X`s and b triples of `O`s and the game is not finished, then define

$$\texttt{value}(s, \texttt{X}, \texttt{limit}, \alpha, \beta) = \frac{a-b}{10} \quad \text{if } \texttt{limit} = 0.$$

In a similar way, pairs can be defined as a set of two marks of the same player. These are valued with a factor of $\frac{1}{100}$. Using this heuristic, the resulting game engine is already quite strong when looking 10 moves ahead.

```
def maxValue(State, limit, alpha=-1, beta=1):
    if finished(State):
        return utility(State)
    if limit == 0:
        return heuristic(State)
    value      = alpha
    NextStates = next_states(State, gPlayers[0])
    Moves      = []  # empty priority queue
    for ns in NextStates:
        val = value_cache(ns, limit-2)
        if val == None:
            val = -1 # unknown values are assumed to be worse than known values
        # heaps are sorted ascendingly, hence the minus
        heapq.heappush(Moves, (-val, ns))
    while Moves != []:
        _, ns = heapq.heappop(Moves)
        value = max(value, evaluate(ns, limit-1, minValue, value, beta))
        if value >= beta:
            return value
    return value

def minValue(State, limit, alpha=-1, beta=1):
    if finished(State):
        return utility(State)
    if limit == 0:
        return heuristic(State)
    value      = beta
    NextStates = next_states(State, gPlayers[1])
    Moves      = []  # empty priority queue
    for ns in NextStates:
        val = value_cache(ns, limit-2)
        if val == None:
            val = 1
        heapq.heappush(Moves, (val, ns))
    while Moves != []:
        _, ns = heapq.heappop(Moves)
        value = min(value, evaluate(ns, limit-1, maxValue, alpha, value))
        if value <= alpha:
            return value
    return value

    def value_cache(State, limit):
    flag, value = gCache.get((State, limit), ('?', None))
        return value
```

Figure 4.11: Depth-limited α-β-pruning.

Chapter 5

Equational Theorem Proving

Mathematics, particularly the field of mathematical theorem proving, is intrinsically linked to the notion of intelligence. Automated theorem proving represents a significant branch of artificial intelligence dedicated to the application of AI techniques within mathematics. The domain of automated theorem proving is extensive enough to warrant a book of its own. Due to time constraints, this chapter will focus exclusively on equational theorem proving. In *equational theorem proving*, we start with a collection of axioms, which are expressed as equations, and seek to determine which additional equations can be inferred from these axioms. As an illustration, consider a group $\mathcal{G}$, defined as a quadruple:

$$\mathcal{G} = \langle G, e, \circ, i \rangle$$

subject to the following conditions:

(a) G is a set, whose elements are referred to as group elements.

(b) $e \in G$, signifying that e is an element of G.

The element e is the neutral element of $\mathcal{G}$.

(c) $\circ : G \times G \to G$, indicating that $\circ$ is a binary operation on G that maps pairs of group elements to a group element. This operation is termed the multiplication of the group $\mathcal{G}$.

(d) $i : G \to G$, denoting that i is a unary operation on G mapping each group element to another group element. For any element $x \in G$, $i(x)$ is known as the inverse of x.

(e) The set G, along with the operations defined, must satisfy the group axioms:

- $e \circ x = x$ for all $x \in G$,
- $i(x) \circ x = e$ for all $x \in G$,
- $(x \circ y) \circ z = x \circ (y \circ z)$ for all $x, y, z \in G$.

In abstract algebra, it is shown that these axioms imply the equation

$$x \circ i(x) = e,$$

i.e. the left inverse of any group element x is also a right inverse of x. A possible proof runs as follows:

$$\begin{array}{lcll}
x \circ i(x) & = & e \circ \bigl(x \circ i(x)\bigr) & \text{because } e \text{ is left-neutral} \\
 & = & \Bigl(i\bigl(x \circ i(x)\bigr) \circ \bigl(x \circ i(x)\bigr)\Bigr) \circ \bigl(x \circ i(x)\bigr) & \text{because } i\bigl(x \circ i(x)\bigr) \circ \bigl(x \circ i(x)\bigr) = e \\
 & = & i\bigl(x \circ i(x)\bigr) \circ \Bigl(\bigl(x \circ i(x)\bigr) \circ \bigl(x \circ i(x)\bigr)\Bigr) & \text{associativity} \\
 & = & i\bigl(x \circ i(x)\bigr) \circ \biggl(x \circ \Bigl(i(x) \circ \bigl(x \circ i(x)\bigr)\Bigr)\biggr) & \text{associativity} \\
 & = & i\bigl(x \circ i(x)\bigr) \circ \biggl(x \circ \Bigl(\bigl(i(x) \circ x\bigr) \circ i(x)\Bigr)\biggr) & \text{associativity} \\
 & = & i\bigl(x \circ i(x)\bigr) \circ \Bigl(x \circ \bigl(e \circ i(x)\bigr)\Bigr) & \text{because } i(x) \circ x = e \\
 & = & i\bigl(x \circ i(x)\bigr) \circ \bigl(x \circ i(x)\bigr) & \text{because } e \circ i(x) = i(x) \\
 & = & e & \text{because } i(z) \circ z = e \text{ where } z = x \circ i(x).
\end{array}$$

The formulation of proofs for such equations is far from straightforward. However, a systematic method exists for addressing these and related equational challenges. In this chapter, we introduce an algorithm capable of autonomously generating equational proofs similar to those discussed earlier. This algorithm is known as the Knuth-Bendix completion algorithm, a significant discovery by Donald E. Knuth and Peter B. Bendix [KB70].

The structure of this chapter is organized as follows:

1. Initially, we will formally define equational proofs and term rewriting, setting the foundation for subsequent discussions.
2. Subsequently, we explore abstract properties of relations, introducing essential concepts such as confluence and providing proofs for the Church-Rosser theorem and Newman's lemma.
3. The third section delves into term orderings, including the introduction of the Knuth-Bendix ordering, which plays an important role in the algorithm.
4. The final section presents the Knuth-Bendix completion algorithm.

5.1 Equational Proofs

This section defines the notion of an equational proof precisely and discusses how equational proofs can be carried out via term rewriting. In order to do this, we have to define a number of more elementary notions like functions symbols, variables, terms, and substitutions. We begin with the notion of a signature.

Definition 7 (Signature) A signature is a triple of the form

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \textit{arity} \rangle,$$

where we have the following:

1. $\mathcal{V}$ is the set of variables.
2. $\mathcal{F}$ is the set of function symbols.
3. *arity* is a function such that

   $$\textit{arity} : \mathcal{F} \to \mathbb{N}.$$

   If we have $\textit{arity}(f) = n$, then f is said to be an n-ary function symbol.

4. We have $\mathcal{V} \cap \mathcal{F} = \{\}$, i.e. variables are different from function symbols. $\diamond$

Remark: Compared to the definition given in the lecture on logic, the new definition does not include a set $\mathcal{P}$ of predicate symbols.

Example: The signature of group theory Σ_G can be defined as follows:

(a) $\mathcal{V} := \{w, x, y, z\}$,

(b) $\mathcal{F} := \{e, i, \circ\}$,

(c) $\mathit{arity} := \{e \mapsto 0, i \mapsto 1, \circ \mapsto 2\}$,
i.e. e is a constant symbol, i is a unary function symbol, and $\circ$ is a binary function symbol.

(d) $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$. $\diamond$

Having defined the notion of a signature we proceed to define terms.

Definition 8 (Term, $\mathcal{T}_\Sigma$) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$ is a signature, the set of Σ-terms $\mathcal{T}_\Sigma$ is defined inductively:

1. For every variable $x \in \mathcal{V}$ we have $x \in \mathcal{T}_\Sigma$.
2. If $f \in \mathcal{F}$ and $\mathit{arity}(f) = 0$, then $f \in \mathcal{T}_\Sigma$.
3. If $f \in \mathcal{F}$ and $n := \mathit{arity}(f) > 0$ and, furthermore, $t_1, \cdots, t_n \in \mathcal{T}_\Sigma$, then we have
$$f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma.$$
$\diamond$

Example: Given the signature Σ_G defined above, we have the following:

1. $x \in \mathcal{T}_{\Sigma_G}$,
because every variable is a Σ_G-term.
2. $e \in \mathcal{T}_{\Sigma_G}$.
3. $\circ(e, x) \in \mathcal{T}_{\Sigma_G}$.
4. $\circ\big(\circ(x, y), z\big) \in \mathcal{T}_{\Sigma_G}$.

Remark: Later on we will often use an infix notation for binary function symbols. In general, if f is a binary function symbol, then the term $f(t_1, t_2)$ is written as $t_1\ f\ t_2$. If this notation would result in an ambiguity because either t_1 or t_2 is also written in infix notation, then we use parenthesis to resolve the ambiguity. For example, we will write

$$(x \circ y) \circ z \quad \text{instead of} \quad \circ\big(\circ(x, y), z\big) \in \mathcal{T}_{\Sigma_G}.$$

Note that we cannot write the term $\circ\big(\circ(x, y), z\big)$ as $x \circ y \circ z$ because that notation is ambiguous, since it can be interpreted as either $(x \circ y) \circ z$ or $x \circ (y \circ z)$. $\diamond$

Definition 9 (Σ-Equation) Assume a signature Σ is given. A Σ-equation is a pair $\langle s, t \rangle$ such that both s and t are Σ-terms. The Σ-equation $\langle s, t \rangle$ is written as

$$s \approx t.$$
$\diamond$

Remark: We use the notation $s \approx t$ instead of the notation $s = t$ in order to distinguish between the notion of a Σ-equation and the notion of equality of terms. So when s and t are Σ-terms and we write $s = t$ we do mean that s and t are literally the same terms, while writing $s \approx t$ means that we are interested in the logical consequences that would follow from the assumption that the interpretation of s and t are the same in certain Σ-algebras. The notion of a Σ-algebra is defined next. ◇

Definition 10 (Σ-Algebra) Assume a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$ is given. A Σ-algebra[1] is a pair of the form $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ where:

1. A is a nonempty set that is called the universe of the Σ-algebra $\mathfrak{A}$.
2. $\mathcal{J}$ is the interpretation of the function symbols. Technically, $\mathcal{J}$ is a function that is defined on the set $\mathcal{F}$ of all function symbols. For every function symbol $f \in \mathcal{F}$ we have that

   $$\mathcal{J}(f) : A^{\mathit{arity}(f)} \to A,$$

   i.e. $\mathcal{J}(f)$ is a function from A^n to A where n is the arity of the function symbol f.

   If $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ is a Σ-algebra, then the function $\mathcal{J}(f)$ is usually written more concisely as $f^{\mathfrak{A}}$.

The set of all Σ-algebras is written as $\mathtt{Alg}(\Sigma)$. ◇

Example: In this example we construct a Σ_G-algebra where Σ_G is the signature of group theory defined earlier. We define $G := \{0, 1\}$ and define the interpretations $\mathcal{J}(f)$ for $f \in \{e, i, \circ\}$ as follows:

1. $\mathcal{J}(e) := 0$.
2. $\mathcal{J}(i) := \{0 \mapsto 0, 1 \mapsto 1\}$.
3. $\mathcal{J}(\circ) := \{\langle 0,0 \rangle \mapsto 0, \langle 0,1 \rangle \mapsto 1, \langle 1,0 \rangle \mapsto 1, \langle 1,1 \rangle \mapsto 0\}$.

Then $\mathfrak{G} = \langle G, \mathcal{J} \rangle$ is a Σ_G-algebra. ◇

Remark: Alternatively, we could have given the interpretation of the multiplication symbol $\circ$ as

$$\mathcal{J}(\circ)(x, y) := (x + y) \,\%\, 2.$$

Definition 11 (Variable Assignment)
If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$ is a signature and $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ is a Σ-algebra, then a variable assignment is a function of the form

$$I : \mathcal{V} \to A$$

that is the variable assignment I maps every variable $v \in \mathcal{V}$ to a value in the set A.

Definition 12 (Evaluation, Valid Equation)
If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$ is a signature, $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ is a Σ-algebra, and I is a variable assignment, then we can evaluate Σ-terms in $\mathfrak{A}$ as follows:

1. $\mathtt{eval}(x, I) := I(x)$ for all $x \in \mathcal{V}$.
2. $\mathtt{eval}(c, I) := c^{\mathfrak{A}}$ for every constant symbol $c \in \mathcal{F}$.

[1]The notion of a Σ-algebra is a notion that is used both in logic and in universal algebra. In universal algebra, a Σ-algebra is also known as an algebraic structure. This notion is not related to and should not be confused with the notion of a σ-algebra, which is a notion used in the field of measure theory. Note that the notion used in measure theory is always written with a lower case σ, while the notion used in logic is written with a capital Σ.

3. $\texttt{eval}\big(f(t_1, \cdots, t_n), I\big) := f^{\mathfrak{A}}\big(\texttt{eval}(t_1, I), \cdots, \texttt{eval}(t_n, I)\big)$.

A Σ-equation $s \approx t$ is valid in the Σ-algebra $\mathfrak{A}$ iff we have

$$\texttt{eval}(s, I) = \texttt{eval}(t, I) \text{ for all variable assignments } I : \mathcal{V} \to A.$$

This is written as

$$\mathfrak{A} \models s \approx t$$

and we say that $\mathfrak{A}$ satisfies the equation $s \approx t$. ◇

Example: Continuing the previous example we have the following:

1. $\mathfrak{G} \models e \circ x \approx x$,
2. $\mathfrak{G} \models i(x) \circ x \approx e$,
3. $\mathfrak{G} \models (x \circ y) \circ z \approx x \circ (y \circ z)$. ◇

Definition 13 (E-Variety) Assume that Σ is a signature and E is a set of Σ-equations. The collection of all Σ-structures that satisfy every equation from E is called the E-variety.

$$\texttt{Variety}(E) := \big\{\mathfrak{A} \in \texttt{Alg}(\Sigma) \bigm| \forall (s \approx t) \in E : \mathfrak{A} \models s \approx t\big\}.$$

To put it differently, the Σ-structure $\mathfrak{A}$ is a member of $\texttt{Variety}(E)$ iff

$$\mathfrak{A} \models s \approx t \quad \text{for every equation } s \approx t \text{ in } E.$$ ◇

Example: Define $E := \big\{e \circ x = x, i(x) \circ x = e, (x \circ y) \circ z = x \circ (y \circ z)\big\}$. This set of equations defines the variety of groups. You can check that the Σ_G-algebra $\mathfrak{G}$ is a member of this variety and hence it is a group. ◇

Given a set of Σ-equations E it is natural to ask which other equations are logical consequences of the equations in E. This notion is defined below.

Definition 14 (Logical Consequence) Assume a signature Σ and a set E of Σ-equations to be given. Then the equation $s \approx t$ is a logical consequence of E iff we have

$$\mathfrak{A} \models s \approx t \quad \text{for every } \mathfrak{A} \in \texttt{Variety}(E).$$

If $s \approx t$ is a logical consequence of the set of equations E, then this is written as

$$E \models s \approx t.$$

Therefore we have $E \models s \approx t$ if and only if every Σ-algebra that satisfies all equations from E also satisfies the equation $s \approx t$. ◇

Example: In the introduction of this chapter we have already seen that if we define

$$E := \big\{e \circ x = x, i(x) \circ x = e, (x \circ y) \circ z = x \circ (y \circ z)\big\},$$

then we have

$$E \models x \circ i(x) \approx e.$$ ◇

The notion $E \models s \approx t$ is a semantic notion. We cannot hope to implement this notion directly because if a set of equations E and a possible logical consequence $s \approx t$ is given, there are, in general, infinitely many Σ-algebras that have to be checked. Fortunately, the notion $E \models s \approx t$ has a syntactical analog $E \vdash s \approx t$ (read: E proves $s \approx t$) that can be implemented and that is at least semi-decidable, i.e. we can create a program that given a set of equations E and an equation $s \approx t$ will return `True` if $E \vdash s \approx t$ holds, and will either return `False` or run forever if $E \vdash s \approx t$ does not hold. Even more fortunately, Gödels completeness theorem implies that the syntactical notion coincides with the semantic notion, i.e. we have

$$E \models s \approx t \quad \text{if and only if} \quad E \vdash s \approx t.$$

5.1.1 A Calculus for Equality

In this subsection we assume a signature Σ and a set of Σ-equations E to be given. Our goal is to define the provability notion $E \vdash s \approx t$, which is read as E proves $s \approx t$. However, in order to do this we first need to define the notion of a substitution.

Definition 15 (Σ-Substitution) Assume that a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathit{arity} \rangle$ is given. A Σ-substitution σ is a map of the form

$$\sigma : \mathcal{V} \to \mathcal{T}_\Sigma$$

such that the set $\mathit{dom}(\sigma) := \big\{ x \in \mathcal{V} \mid \sigma(x) \neq x \big\}$ is finite. If we have $\mathit{dom}(\sigma) = \{x_1, \cdots, x_n\}$ and $t_i = \sigma(x_i)$ for all $i = 1, \cdots, n$, then we use the following notation:

$$\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}.$$

The set of all Σ-Substitutions is denoted as $\texttt{Subst}(\Sigma)$. $\diamond$

A substitution $\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ can be applied to a term t by replacing the variables x_i with the terms t_i. We will use the postfix notation $t\sigma$ to denote the application of the substitution σ to the term t. Formally, the notation $t\sigma$ is defined by induction on t:

1. $x\sigma := \sigma(x) \quad$ for all $x \in \mathcal{V}$.
2. $c\sigma = c \quad$ for every constant $c \in \mathcal{F}$.
3. $f(t_1, \cdots, t_n)\sigma := f\big(t_1\sigma, \cdots, t_n\sigma\big)$.

Definition 16 ($E \vdash s \approx t$)
Given a signature Σ and a set of Σ-equations E, the notion $E \vdash s \approx t$ is defined inductively.

1. $E \vdash s \approx t \quad$ for every Σ-equations $(s \approx t) \in E$. (Axioms)
2. $E \vdash s \approx s \quad$ for every Σ-term s. (Reflexivity)
3. If $E \vdash s \approx t$, then $E \vdash t \approx s$. (Symmetry)
4. If $E \vdash r \approx s$ and $E \vdash s \approx t$, then $E \vdash r \approx t$. (Transitivity)
5. If $\mathit{arity}(f) = n$ and $E \vdash s_i \approx t_i$ for all $i \in \{1, \cdots, n\}$, then $E \vdash f(s_1, \cdots, s_n) \approx f(t_1, \cdots, t_n)$. (Congruence)
6. If $E \vdash s \approx t$ and σ is a Σ-substitution, then $E \vdash s\sigma \approx t\sigma$. (Stability)

We read $E \vdash s \approx t$ as E proves $s \approx t$. $\diamond$

The definition of $E \vdash s \approx t$ given above is due to Gottfried Wilhelm Leibniz.

5.1.2 Term Rewriting

It turns out that, although it is possible to implement the notion $E \vdash s \approx t$ directly, it is much more efficient to refine this notion a little bit. To this end we need to introduce the notion of a position u in a term t, the notion of the subterm of a given term t at a given position, and the notion of replacing a subterm at a given position by another term. We define these notions next.

Definition 17 (Positions of a Term t, $\mathcal{P}\mathrm{os}(t)$)
Given a term t the set of all positions of t is written as $\mathcal{P}\mathrm{os}(t)$ and is defined by induction on t:

1. $\mathcal{P}\mathrm{os}(x) := \big\{[\,]\big\}$ for every variable x.
2. $\mathcal{P}\mathrm{os}(c) := \big\{[\,]\big\}$ for every constant c.
3. $\mathcal{P}\mathrm{os}\big(f(t_1, \cdots, t_n)\big) := \big\{[\,]\big\} \cup \bigcup_{i=1}^{n} \big\{[i] + u \mid u \in \mathcal{P}\mathrm{os}(t_i)\big\}$ for every term $f(t_1, \cdots, t_n)$. ◇

Definition 18 (Subterm at a given Position, t/u)
Given a term t and a position $u \in \mathcal{P}\mathrm{os}(t)$, the subterm of t at position u is written as t/u. This expression is defined by induction on u.

1. $t/[\,] := t$,
2. $f(t_1, \cdots, t_n)/\big([i] + u\big) := t_i/u$. ◇

Definition 19 (Subterm Replacement, $t[u \mapsto s]$)
Given a term t, a position $u \in \mathcal{P}\mathrm{os}(t)$, and a term s, the expression $t[u \mapsto s]$ denotes the term that results from t when the subterm t/u is replaced by the term s. This expression is defined by induction on u.

1. $t\big[[\,] \mapsto s\big] := s$,
2. $f(t_1, \cdots, t_n)\big[[i] + u \mapsto s\big] := f(t_1, \cdots, t_{i-1}, t_i[u \mapsto s], t_{i+1}, \cdots, t_n)$. ◇

Example: Define $t := (x \circ y) \circ z$. Then we have

$$\mathcal{P}\mathrm{os}\big((x \circ y) \circ z\big) = \big\{[\,], [1], [1, 1], [1, 2], [2]\big\}.$$

Furthermore, we have the following:

1. $\big((x \circ y) \circ z\big)/[\,] = (x \circ y) \circ z$,
2. $\big((x \circ y) \circ z\big)/[1] = x \circ y$,
3. $\big((x \circ y) \circ z\big)/[1, 1] = x$,
4. $\big((x \circ y) \circ z\big)/[1, 2] = y$,
5. $\big((x \circ y) \circ z\big)/[2] = z$.

We also have

$$\big((x \circ y) \circ z\big)\big[[1] \mapsto y \circ x\big] = (y \circ x) \circ z.$$ ◇

Definition 20 ($\leftrightarrow_E$) Given a set of Σ-equations E and two Σ-terms s and t we define that

$$s \leftrightarrow_E t$$

holds if and only if the following conditions are satisfied:

(a) There exists an equation $l \approx r$ such that either $(l \approx r) \in E$ or $(r \approx l) \in E$.

(b) There is a position $u \in \mathcal{P}\mathrm{os}(s)$ and a substitution σ such that $s/u = l\sigma$.

(c) $t = s[u \mapsto r\sigma]$. $\diamond$

To put this is words: We have $s \leftrightarrow_E t$ iff there is an equation $l \approx r$ such that either the equation $l \approx r$ or the equation $r \approx l$ is an element of the set of equations E and, furthermore, s contains the subterm $l\sigma$ and t results from s by replacing the subterm $l\sigma$ with the subterm $r\sigma$.

Example: If we have $E = \bigl\{i(x) \circ x \approx e\bigr\}$ then

$$\bigl(i(a) \circ a\bigr) \circ b \;\leftrightarrow_E\; e \circ b$$

because the right hand side $e \circ b$ results from the left hand side $\bigl(i(a) \circ a\bigr) \circ b$ by replacing the subterm $i(a) \circ a$ that occurs at position [1] by the term e. This is possible because the equation $i(x) \circ x \approx e$ tells us that $i(a) \circ a$ is equal to e. $\diamond$

Next, we define the relation $\leftrightarrow_E^*$ as the reflexive and transitive closure of the relation $\leftrightarrow_E$.

Definition 21 ($\leftrightarrow_E^*$) For Σ-terms s and t the notion $s \leftrightarrow_E^* t$ is defined inductively as follows:

1. We have $s \leftrightarrow_E^* s$ for every Σ-term s.
2. If $s \leftrightarrow_E t$, then $s \leftrightarrow_E^* t$.
3. If u is a Σ-term such that both $s \leftrightarrow_E u$ and $u \leftrightarrow_E^* t$ holds, then we have $s \leftrightarrow_E^* t$. $\diamond$

Given this definition it is now possible to show the following theorem.

Theorem 22 If E is a set of equations and $s \approx t$ is an equation, then we have

$$E \vdash s \approx t \quad \text{if and only if} \quad s \leftrightarrow_E^* t.$$

For each of the two directions that has to be proven, the proof can be done by a straightforward, but tedious induction.

5.1.3 Proofs via Term Rewriting

The implementation of the relation $\leftrightarrow_E$ remains inefficient due to the dual utility of each equation within E. Specifically, for an equation $l \approx r$ contained in E, it can be applied in two directions: we can replace a subterm matching $l\sigma$ with $r\sigma$ utilizing the equation from left to right, or conversely, substitute a subterm resembling $r\sigma$ with $l\sigma$ by applying the equation from right to left. The seminal insight of Donald E. Knuth and Peter B. Bendix [KB70] was to recognize that if the equations could be ordered such that the left-hand side is consistently more complex than the right-hand side, then it would be feasible to apply these equations in a singular direction by incorporating certain supplementary equations into E throughout this procedure. This approach necessitates the subsequent definition.

Definition 23 (Rewrite Order)
A binary relation $\prec \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ is a rewrite order if and only if we have the following:

1. $\prec$ is a strict partial order on $\mathcal{T}_\Sigma$, i.e. we have
   (a) $\neg(s \prec s)$ for all $s \in \mathcal{T}_\Sigma$, i.e. $\prec$ is irreflexive.
   (b) $r \prec s \wedge s \prec t \;\Rightarrow\; r \prec t$ for all $r, s, t \in \mathcal{T}_\Sigma$, i.e. $\prec$ is transitive.

2. $\prec$ is stable under substitutions, i.e we have

$$r \prec l \;\Rightarrow\; r\sigma \prec l\sigma \quad \text{for every substitution } \sigma.$$

3. $\prec$ is a congruence, i.e. we have

$$r \prec l \;\Rightarrow\; s[u \mapsto r] \prec s[u \mapsto l] \text{ for every } \Sigma\text{-term } s \text{ and every } u \in \mathcal{P}\mathrm{os}(s).$$

4. The relation $\prec$ is well-founded, i.e. there is no infinite sequence of the form $(s_n)_{n\in\mathbb{N}}$ such that we have

$$s_{n+1} \prec s_n \quad \text{for all } n \in \mathbb{N}.$$

If E is a set of equations, then a binary relation $\prec\, \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ is a rewrite order w.r.t. E if, in addition to being a rewrite order, it satisfies

$$r \prec l \quad \text{for every equation } (l \approx r) \in E.$$

This means that all equations in E are ordered such that the right hand side is smaller than the left hand side w.r.t. $\prec$. ◇

Later we will introduce the Knuth-Bendix order as an example of a rewrite order.

In the following let us assume that a binary relation $\prec$ on terms is given and let us, furthermore, assume that this relation is a rewrite order with respect to a some set of equations R. We will call these equations rewrite rules. We proceed to define the relation $\to_R$ on the set of Σ-terms $\mathcal{T}_\Sigma$.

Definition 24 (Rewrite Relation $\to_R$)
Given a set of rewrite rules R and two Σ-terms s and t we define that

$$s \to_R t \quad (\text{read: } s \text{ rewrites to } t)$$

if and only if there exists a rewrite rule $(l \approx r) \in R$ such that the following conditions are satisfied:

(a) There is a position $u \in \mathcal{P}\mathrm{os}(s)$ and a substitution σ such that $s/u = l\sigma$.

(b) $t = s[u \mapsto r\sigma]$. ◇

To put it differently, we have $s \to_R t$ if there is a rewrite rule $l \approx r$ in R and the left hand side l of this rewrite rule matches a subterm s/u of s via a substitution σ. If replacing this subterm by $r\sigma$ results in t, then s rewrites to t.

Similar to the definition of $\leftrightarrow_E^*$ we next define $\to_R^*$ as the reflexive and transitive closure of $\to_R$.

Definition 25 ($\to_R^*$) For Σ-terms s and t the notion $s \to_R^* t$ is defined inductively as follows:

1. We have $s \to_R^* s$ for every Σ-term s.
2. If $s \to_R t$, then $s \to_R^* t$.
3. If u is a Σ-term such that both $s \to_R u$ and $u \to_R^* t$ holds, then we have $s \to_R^* t$. ◇

Definition 26 (Normal Form)
A Σ-term s is in normal form w.r.t. a rewrite relation $\to_R$ iff there is no Σ-term t such that $s \to_R t$, i.e. the term s cannot be simplified anymore by rewriting. ◇

The basic idea of a rewrite proof of an equation $s \approx t$ is now the following:

1. We rewrite s using the rewrite rules from R into a term $\widehat{s}$ that is in normal form:

$$s \rightarrow_R s_1 \rightarrow_R s_2 \rightarrow_R \cdots \rightarrow_R s_m = \widehat{s}$$

2. Similarly, we rewrite t using the rewrite rules from R into a term $\widehat{t}$ that is in normal form:

$$t \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \cdots \rightarrow_R t_n = \widehat{t}\,.$$

3. If the relation $s \rightarrow t$ is confluent (this notion is defined in the next section), then we have

$$s \leftrightarrow_E t \quad \Leftrightarrow \quad \widehat{s} = \widehat{t}.$$

By this method, the question of whether $s \leftrightarrow_E^* t$ holds can be simplified to the computation of normal forms, which is often achievable with high efficiency. The subsequent sections of this chapter are structured as follows:

(a) The forthcoming section explores the concept of confluence and establishes a theorem instrumental in demonstrating the confluence of a relation.

(b) Subsequently, we investigate rewrite orderings. Specifically, we examine the Knuth-Bendix ordering, the designated rewrite ordering that facilitates the construction of several confluent term rewriting systems.

(c) Thereafter, we introduce the Knuth-Bendix completion algorithm, a methodology capable of augmenting a set of equations to ensure the rewrite relation $\rightarrow_R$ is made confluent.

(d) Finally, an implementation of the Knuth-Bendix completion algorithm is given.

(e) Lastly, we apply this algorithm to the axioms of group theory and a number of theories that are related to group theory.

5.2 Confluence

In this section we assume that a binary relation $\rightarrow$ is given on a set M, that is we have $\rightarrow\ \subseteq\ M \times M$. Instead of writing $(a, b) \in\ \rightarrow$ we use infix notation and write $a \rightarrow b$. Furthermore, we assume that $\rightarrow$ is well-founded, i.e. there is no infinite sequence $(x_n)_{n\in\mathbb{N}}$ such that

$$s_n \rightarrow s_{n+1} \quad \text{holds for all } n \in \mathbb{N}.$$

We denote the equivalence relation generated by $\rightarrow$ as $\leftrightarrow^*$ and the reflexive and transitive closure of $\rightarrow$ is written as $\rightarrow^*$.

Definition 27 (Confluence) The relation $\rightarrow\ \subseteq\ M \times M$ is confluent iff the following holds:

$$\forall a, b, c \in M : \Big(a \rightarrow^* b \,\wedge\, a \rightarrow^* c \quad \Rightarrow \quad \exists d \in M : \big(b \rightarrow^* d \,\wedge\, c \rightarrow^* d\big)\Big) \qquad \diamond$$

Figure 5.1 shows a diagram picturing the notion of confluence. Here, the relation $\rightarrow^*$ is denoted by a snakelike arrow.

The next theorem shows that confluence is all we need to reduce the relation $\leftrightarrow^*$ to the relation $\rightarrow^*$.

Theorem 28 (Church-Rosser) If the relation $\rightarrow\ \subseteq\ M \times M$ is confluent, then we have

$$\forall a, b \in M : \Big(a \leftrightarrow^* b \,\Leftrightarrow\, \exists c \in M : \big(a \rightarrow^* c \,\wedge b \rightarrow^* c\big)\Big).$$

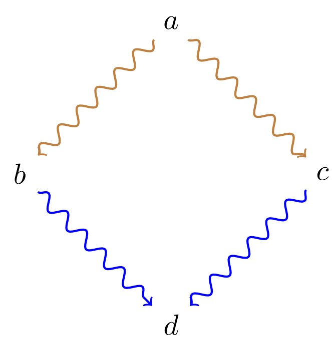


Figure 5.1: Confluence

Proof: If $a \leftrightarrow^* b$ then there is a finite sequence $(s_k)_{k \in \{0, \cdots, n\}}$ such that

$$a = s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{n-1} \leftrightarrow s_n = b.$$

We prove by induction on n that there is an element $c \in M$ such that both $a \to^* c$ and $b \to^* c$ holds.

Base Case: $n = 0$.

Then we have $a = b$ and we can define $c := a$.

Induction Step: $n \mapsto n + 1$

We have $a = s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_n \leftrightarrow s_{n+1} = b$. By induction hypotheses we know that there exists a $d \in M$ such that

$$a \to^* d \quad \text{and} \quad s_n \to^* d$$

hold. Furthermore, we either have

$$s_n \to b \quad \text{or} \quad b \to s_n.$$

We discuss these cases one by one.

1. Case: $s_n \to b$.
   Since we also have $s_n \to^* d$, the confluence of the relation $\to$ shows that there is an element $c \in M$ such

   $$b \to^* c \quad \text{and} \quad d \to^* c$$

   holds. From $a \to^* d$ and $d \to^* c$ we have that $a \to^* c$. Since we already know that $b \to^* c$, the proof is complete in this case.
2. Case: $b \to s_n$.
   Since we have $b \to s_n$ and $s_n \to^* d$, we can conclude $b \to^* d$. Since we also have $a \to^* d$, the proof is complete if we define $c := d$. □

In general, it is hard to prove that a relation $\to$ is confluent. Things get easier if the relation $\to$ is well-founded, since then there is a weaker notion than confluence that is already sufficient to guarantee confluence.

Definition 29 (Local Confluence)
The relation $\to \subseteq M \times M$ is locally confluent iff the following holds:

$$\forall a, b, c \in M : \Big(a \to b \;\wedge\; a \to c \quad \Rightarrow \quad \exists d \in M : \big(b \to^* d \;\wedge\; c \to^* d\big)\Big) \qquad \diamond$$

Figure 5.2 shows a diagram picturing the notion of local confluence. Here, the relation $\to^*$ is denoted by a snakelike arrow.

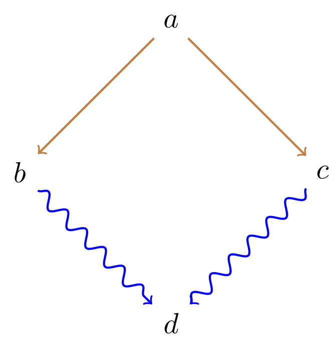


Figure 5.2: Local Confluence

Lemma 30 Assume that the binary relation $\prec\; \subseteq\; M \times M$ is a well-founded partial ordering and that $A \subseteq M$ is a non-empty set. Then A must have a minimal element, i.e. there is an element $m \in A$ such that there is no element $n \in A$ that satisfies $n \prec m$.

Proof: The proof is indirect. Assume that the set A has no minimal element. We will define an infinite sequence $(a_n)_{n\in\mathbb{N}}$ by induction on n. This sequence will satisfy the following conditions:

(a) $a_n \in A$ for all $n \in \mathbb{N}$,

(b) $a_{n+1} \prec a_n$ for all $n \in \mathbb{N}$.

However, the second property would contradict the fact that $\prec$ is well-founded. Therefore, if we are able to construct the sequence $(a_n)_{n\in\mathbb{N}}$ with the properties stated above, then our assumption that A has no minimal element must be wrong.

B.C.: $n = 0$.

Since A is not empty, there is some element $x \in A$. We pick any such x and define $a_0 := x$.

I.S.: $n \mapsto n + 1$

Since we have assumed that A has no minimal element, a_n is not minimal. Therefore the set

$$B := \{y \in A \mid y \prec a_n\}$$

is not empty. We pick an arbitrary element y from this set and define

$$a_{n+1} := y.$$

Then we have both $a_{n+1} \in A$ and $a_{n+1} \prec a_n$ and hence we have constructed the sequence $(a_n)_{n\in\mathbb{N}}$ with the properties stated above.

Since the existence of this infinite decreasing sequence contradicts the well-foundednes of $\prec$, our proof is complete. $\Box$

Theorem 31 (**Transfinite Induction**)
Assume the the binary relation $\prec \subseteq M \times M$ is a well-founded partial orderings and $F(x)$ is some formula. If we have that

$$\forall a \in M : \Big(\forall b \in M : \big(b \prec a \;\Rightarrow\; F(b)\big) \;\Rightarrow\; F(a)\Big), \tag{TI}$$

then we can conclude that $\forall c \in M : F(c)$ holds.

Proof: Before we start with the proof, let us put the formula given above in words. The formula

$$\forall b \in M : \big(b \prec a \;\Rightarrow\; F(b)\big)$$

expresses that $F(b)$ holds for all b smaller than a. The principle of transfinite induction tells us that, if we are able to conclude $F(a)$ for an arbitrary a from the fact that $F(b)$ holds for all b that are strictly less than a, then we can conclude that $F(c)$ holds for all $c \in M$.

The proof that $F(c)$ holds for all $c \in M$ is indirect. Assume that there is an $a \in M$ such that $F(a)$ does not hold. By the last Lemma we may furthermore assume that this a is minimal w.r.t. the ordering $\prec$. But this implies that

$$\big\{b \in M \mid (b \prec a \wedge \neg F(b)\big\} = \emptyset,$$

Therefore, $F(b)$ holds for all b less than a. By the assumption (TI) we can conclude that $F(a)$ holds, contradicting the assumption that $F(a)$ does not hold. □

Theorem 32 (**Newman's Lemma**)
If the relation $\rightarrow \subseteq M \times M$ is well-founded and locally confluent, then it is already confluent.

Proof: Given any $a \in M$, we define the following formula:

$$F(a) \;:=\; \forall b, c \in M : \Big(a \rightarrow^* b \;\wedge\; a \rightarrow^* c \quad\Rightarrow\quad \exists d \in M : \big(b \rightarrow^* d \;\wedge\; c \rightarrow^* d\big)\Big)$$

We prove that $F(a)$ holds for all $a \in M$ by transfinite induction. The relation $b \prec a$ that is needed in the proof of transfinite induction is defined as $a \rightarrow^+ b$, that is we have

$$b \prec a \quad \text{iff } a \rightarrow^+ b.$$

Here, $\rightarrow^+$ is the transitive closure of $\rightarrow$, i.e. we have

$$a \rightarrow^+ b$$

iff there is a finite sequence $c_0,\, c_1,\, \cdots,\, c_n,\, c_{n+1}$ such that we have

$$a = c_0, \quad c_i \rightarrow c_{i+1} \text{ for all } i = 1, \cdots, n, \quad \text{and} \quad c_{n+1} = b.$$

Therefore, in order to prove $F(a)$ we may assume that $F(b)$ already holds for all b such that $a \rightarrow^+ b$. So let us assume that we have

$$a \rightarrow^* b \quad \text{and} \quad a \rightarrow^* c.$$

We have to find an element $d \in M$ such that both $b \rightarrow^* d$ and $c \rightarrow^* d$ holds. Now since $a \rightarrow^* b$, either $a = b$ or there is an element b_1 such that

$$a \rightarrow b_1 \rightarrow^* b$$

holds. If $a = b$ we can define $d := c$ and because of $a \rightarrow^* c$ we would then have both

$$b \rightarrow^* d \quad \text{and} \quad c \rightarrow^* d$$

and therefore, in the case $a = b$, we are done. Similarly, since $a \rightarrow^* c$ we either have $a = c$ or there is an element c_1 such that

$$a \rightarrow c_1 \rightarrow^* c$$

holds. If $a = c$ we can define $d := b$ and because of $a \rightarrow^* b$ we would then have both

$$b \rightarrow^* d \quad \text{and} \quad c \rightarrow^* d$$

and are done again. Now the case that is left is the following:

$$a \rightarrow b_1 \rightarrow^* b \quad \text{and} \quad a \rightarrow c_1 \rightarrow^* c.$$

Since $\rightarrow$ is locally confluent and we have both $a \rightarrow b_1$ and $a \rightarrow c_1$ there exists an element d_1 such that we have

$$b_1 \rightarrow^* d_1 \quad \text{and} \quad c_1 \rightarrow^* d_1.$$

Now as b_1 is a successor of a and we have both

$$b_1 \rightarrow^* b \quad \text{and} \quad b_1 \rightarrow^* d_1,$$

our induction hypotheses tells us that there is an element d_2 such that we have both

$$b \rightarrow^* d_2 \quad \text{and} \quad d_1 \rightarrow^* d_2.$$

Now we have $c_1 \rightarrow^* d_1$ and $d_1 \rightarrow^* d_2$, which implies

$$c_1 \rightarrow^* d_2$$

As we also have $c_1 \rightarrow^* c$ we have both

$$c_1 \rightarrow^* d_2 \quad \text{and} \quad c_1 \rightarrow^* c.$$

Since c_1 is a successor of a, the induction hypotheses tells us that there is an element d such that we have both

$$d_2 \rightarrow^* d \quad \text{and} \quad c \rightarrow^* d.$$

As we have $b \rightarrow^* d_2$ and $d_2 \rightarrow^* d$ we can conclude $b \rightarrow^* d$. Hence we have

$$b \rightarrow^* d \quad \text{and} \quad c \rightarrow^* d$$

and the proof is complete. Figure 5.3 on page 117 shows how the different elements are related and conveys the idea of the proof in a concise way. □

Exercise 13: Assume that M is a non-empty set and that $\prec \subseteq M \times M$ is a well-founded partial order on M. Define the relation $\preceq$ as follows:

$$x \preceq y \quad \text{iff} \quad x = y \vee x \prec y.$$

Furthermore, assume that $f : M \rightarrow M$ has the following property:

$$f(x) \preceq x \quad \text{for all } x \in M$$

Prove that f has a fixed point, i.e. there is an element $z \in M$ such that $f(z) = z$. ◇

5.3 The Knuth-Bendix Order

In this section we define the Knuth-Bendix order $\prec$ on the set $\mathcal{T}_\Sigma$ of Σ-terms.

Definition 33 (Knuth-Bendix Order)
Assume $\Sigma = \langle \mathcal{V}, \mathcal{F}, \texttt{arity} \rangle$ is a signature. A Knuth-Bendix order for Σ is a pair $\langle w, < \rangle$ such that

1. $w : \mathcal{F} \rightarrow \mathbb{N}$,

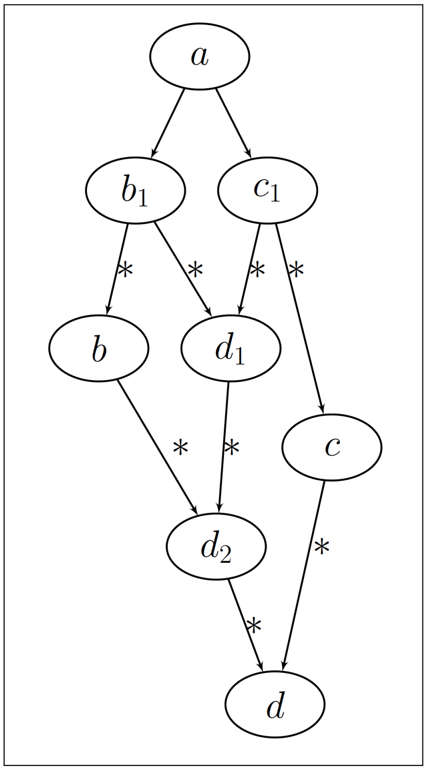


Figure 5.3: The Proof of Newman's Lemma.

i.e. w is a function assigning a natural number to every function symbol. This function w is called the weight function. Furthermore, we must have

(a) There must be at most one function symbol g such that $w(g) = 0$.

(b) If $w(g) = 0$, then g has to be unary, i.e. $\texttt{arity}(g) = 1$.

2. $<$ is a strict total order $<$ on the set of function symbols, i.e. the following conditions need to be satisfied:

   (a) The relation $<$ is irreflexive, that is we have $\neg(f < f)$ for all function symbols f.

   (b) The relation $<$ is transitive, that is we have

   $$f < g \wedge g < h \Rightarrow f < h \quad \text{for all function symbols } f,\ g, \text{ and } h.$$

   (c) The relation $<$ is total, that is we have

   $$f < g \;\vee\; f = g \;\vee\; g < f \quad \text{for all function symbols } f \text{ and } g.$$

3. The order $<$ on the function symbols has to be admissible with respect to the weight function w, i.e. the following condition needs to be satisfied:

$$w(f) = 0 \rightarrow \forall g : \big(g \neq f \rightarrow g < f\big).$$

To put this in words: If the function symbol f has a weight of 0, then all other function symbols g have to be smaller than f w.r.t. the strict order $<$. Note that this implies that there can be at most one function symbol with f such that $w(f) = 0$. This function symbol f is then the maximum w.r.t. the order $<$. ◇

Example: For the signature $\Sigma_g = \big\langle \{w, x, y, z\}, \{e, i, \circ\}, \{e \mapsto 0, i \mapsto 1, \circ \mapsto 2\} \big\rangle$, we can define

(a) $w := \{w(e) \mapsto 1,\ w(i) \mapsto 0,\ w(\circ) \mapsto 1\}$

(b) The relation $<$ is defined as

$$e < \circ < i.$$

Then $\langle w, < \rangle$ is a Knuth-Bendix order. $\diamond$

Definition 34 (Weight of a Term) If Σ is a signature and $\langle w, < \rangle$ is a Knuth-Bendix w.r.t. Σ we define the weight of a term $t \in \mathcal{T}_\Sigma$ by induction on t.

(a) $w(x) := 1$ for all variables x,

(b) $w\big(f(t_1, \cdots, t_n)\big) := w(f) + \sum\limits_{i=1}^{n} w(t_i)$. $\diamond$

Definition 35 (Count) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \texttt{arity} \rangle$ is a signature, then we define the function

$$\texttt{count} : \mathcal{T}_\Sigma \times \mathcal{V} \to \mathbb{N}$$

that takes a term t and a variable x and returns the number of times that x occurs in t. We define $\texttt{count}(t, x)$ by induction on t.

1. $\texttt{count}(x, x) := 1$ for every variable $x \in \mathcal{V}$.
2. $\texttt{count}(y, x) := 0$ if $x \neq y$ for all variables $x, y \in \mathcal{V}$.
3. $\texttt{count}\big(f(t_1, \cdots, t_n), x\big) := \sum\limits_{i=1}^{n} \texttt{count}(t_i, x)$. $\diamond$

Definition 36 ($\prec_{kb}$) Now we are ready to define the Knuth-Bendix ordering on terms. Assume Σ is a signature and $\langle w, < \rangle$ is a Knuth-Bendix order. Given two terms s and t we define $s \prec_{kb} t$ iff one of the following two conditions hold:

1. $w(s) < w(t)$ and $\texttt{count}(s, x) \leq \texttt{count}(t, x)$ for all variables x occurring in s.
2. $w(s) = w(t)$, $\texttt{count}(s, x) \leq \texttt{count}(t, x)$ for all variables x occuring in s, and one of the following subconditions holds:
   (a) $t = f^n(s)$ where $n \geq 1$ and f is the maximum w.r.t. the order $<$ on function symbols, i.e. we have $g < f$ for all function symbols $g \neq f$.
   (b) $s = f(s_1, \cdots, s_m)$, $t = g(t_1, \cdots, t_n)$, and $f < g$.
   (c) $s = f(s_1, \cdots, s_m)$, $t = f(t_1, \cdots, t_m)$, and $[s_1, \cdots, s_m] \prec_{\text{lex}} [t_1, \cdots, t_m]$.
   Here, $\prec_{\text{lex}}$ denotes the lexicographic extension of the ordering $\prec_{kb}$ to lists of terms. It is defined as follows:
   $$[x] + R_1 \prec_{\text{lex}} [y] + R_2 \overset{\text{def}}{\iff} x \prec_{kb} y \vee \big(x = y \wedge R_1 \prec_{\text{lex}} R_2\big)$$

In order to remove clutter we will write $s \prec t$ instead of $s \prec_{kb} t$. $\diamond$

Example: For the signature of group theory $\Sigma_G = \big\langle\{w,x,y,z\},\{e,i,\circ\},\{e \mapsto 0, i \mapsto 1, \circ \mapsto 2\}\big\rangle$ we define the weight function w as follows:

$$w(e) := 1, \quad w(\circ) := 1, \quad \text{and} \quad w(i) := 0.$$

Furthermore, we define a strict total order on the function symbols by setting

$$e < \circ \quad \text{and} \quad \circ < i.$$

Then the order $<$ is admissible with respect to the weight function because the only function symbol that has a weight of 0 is the largest function symbol with respect to the order $<$. Furthermore, we have the following:

(a) $x \prec e \circ x$,

because $w(x) = 1$, $w(e \circ x) = 3$, and $1 < 3$.

(b) $e \prec i(x) \circ x$,

because $w(e) = 1$, $w\big(i(x) \circ x\big) = 3$, and $1 < 3$

(c) $x \circ (y \circ z) \prec (x \circ y) \circ z$

because $w\big((x \circ y) \circ z\big) = 5 = w\big(x \circ (y \circ z)\big)$ and $x \prec x \circ y$, since $w(x) = 1$ and $w(x \circ y) = 3$.

(d) $i(y) \circ i(x) \prec i(x \circ y)$

because we have the following:

(a) $w\big(i(y) \circ i(x)\big) = 3 = w\big(i(x \circ y)\big)$,
(b) $\texttt{count}\big(i(y) \circ i(x), x\big) = 1 = \texttt{count}\big(i(x \circ y), x\big)$,
(c) $\texttt{count}\big(i(y) \circ i(x), y\big) = 1 = \texttt{count}\big(i(x \circ y), y\big)$, and
(d) $\circ < i$. ◇

Theorem 37 If $\langle w, <\rangle$ is a Knuth-Bendix order, then the Knuth-Bendix ordering $\prec_{kb}$ is a rewrite order.

Proving that the Knuth-Bendix order is a strict partial order on the set $\mathcal{T}_\Sigma$ that is stable and a congruence can be done via induction on the structure of the terms. This part of the proof is tedious, but straightforward. The hard part of the proof is to show that the Knuth-Bendix order is well-founded. A proof is given in the book by Franz Baader and Tobias Nipkow [BN98].

5.4 Unification

This section introduces the notion of a most general unifier of two terms. To begin, we define the composition of two Σ-substitutions.

Definition 38 (Composition of Substitutions) Assume that

$$\sigma = \{x_1 \mapsto s_1, \cdots, x_m \mapsto s_m\} \quad \text{and} \quad \tau = \{y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\}$$

are two substitutions such that $\mathit{dom}(\sigma) \cap \mathit{dom}(\tau) = \{\}$. We define the composition $\sigma\tau$ of σ and τ as

$$\sigma\tau := \{x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\}$$ ◇

Example: If we define

$$\sigma := \{x_1 \mapsto c,\ x_2 \mapsto f(x_3)\} \quad \text{and} \quad \tau := \{x_3 \mapsto h(c,c),\ x_4 \mapsto d\},$$

then we have

$$\sigma\tau = \{x_1 \mapsto c,\ x_2 \mapsto f(h(c,c)),\ x_3 \mapsto h(c,c),\ x_4 \mapsto d\}. \qquad \diamond$$

Proposition 39 If t is a term and σ and τ are substitutions such that $dom(\sigma) \cap dom(\tau) = \{\}$ holds, then we have

$$(t\sigma)\tau = t(\sigma\tau). \qquad \diamond$$

This proposition may be proven by induction on t.

Definition 40 (Syntactical Equation) A syntactical equation is a pair $\langle s, t\rangle$ of terms. It is written as $s \doteq t$. A system of syntactical equations is a set of syntactical equations. $\diamond$

Definition 41 (Unifier) A substitution σ solves a syntactical equation $s \doteq t$ iff we have $s\sigma = t\sigma$. If E is a system of syntactical equations and σ is a substitution that solves every syntactical equations in E, then σ is a unifier of E. $\diamond$

If $E = \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}$ is a system of syntactical equations and σ is a substitution, then we define

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

Example: Let us consider the syntactical equation

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

and define the substitution

$$\sigma := \{x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\}.$$

Then σ solves the given syntactical equation because we have

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4)) \quad \text{und}$$
$$p(x_2, x_3)\sigma \quad = p(x_2, f(x_4)). \qquad \diamond$$

Next we develop an algorithm for solving a system of syntactical equations. The algorithm we present was published by Martelli and Montanari [MM82]. To begin, we first consider the cases where a syntactical equation $s \doteq t$ is unsolvable. There are two cases: A syntactical equation of the form

$$f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$$

is certainly unsolvable if f and g are different function symbols. The reason is that for any substitution σ we have that

$$f(s_1, \cdots, s_m)\sigma = f(s_1\sigma, \cdots, s_m\sigma) \quad \text{und} \quad g(t_1, \cdots, t_n)\sigma = g(t_1\sigma, \cdots, t_n\sigma).$$

If $f \neq g$, then the terms $f(s_1, \cdots, s_m)\sigma$ and $g(t_1, \cdots, t_n)\sigma$ start with different function symbols and hence they can't be identical.

The other case where a syntactical equation is unsolvable, is a syntactical equation of the following form:

$$x \doteq f(t_1, \cdots, t_n) \quad \text{where } x \in \texttt{var}\big(f(t_1, \cdots, t_n)\big).$$

This syntactical equation is unsolvable because the term $f(t_1, \cdots, t_n)\sigma$ will always contain at least one more occurrence of the function symbol f than the term $x\sigma$.

Now we are able to present an algorithm for solving a system of syntactical equations, provided the system is solvable. The algorithm will also discover if a system of syntactical equations is unsolvable. The algorithm works on pairs of the form $\langle F, \tau\rangle$ where F is a system of syntactical equations and τ is a substitution. The algorithm starts with the pair $\langle E, \{\}\rangle$. Here E is the system of syntactical equations that is to be solved and $\{\}$ represents the empty substitution. The system works by simplifying the pairs $\langle F, \tau\rangle$ using certain reduction rules that are presented below. These reduction rules are applied until we either discover that the system of syntactical equations is unsolvable or else we reduce the pairs until we finally arrive at a pair of the form $\langle \{\}, \mu\rangle$. In this case μ is a unifier of the system of syntactical equations E. The reduction rules are as follows:

1. If $y \in \mathcal{V}$ is a variable that does not occur in the term t, then we can perform the following reduction:

$$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma\Big\rangle \quad \rightsquigarrow \quad \Big\langle E\{y \mapsto t\}, \sigma\{y \mapsto t\}\Big\rangle \qquad \text{if } y \in \mathcal{V} \text{ and } y \notin \texttt{var}(t)$$

   This reduction rule can be understood as follows: If the system of syntactical equations that is to be solved contains a syntactical equation of the form $y \doteq t$, where the variable y does not occur in the term t, then the syntactical equation $y \doteq t$ can be removed if we apply the substitution $\{y \mapsto t\}$ to both components of the pair

$$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma\Big\rangle.$$

2. If the variable y occurs in the term t, i.e. if $y \in \mathit{Var}(t)$ and, furthermore, $t \neq y$, then the system of syntactical equations $E \cup \big\{y \doteq t\big\}$ has no solution. We write this as

$$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma\Big\rangle \rightsquigarrow \Omega \quad \text{if } y \in \texttt{var}(t) \text{ and } y \neq t.$$

3. If $y \in \mathcal{V}$ and $t \notin \mathcal{V}$, then we have:

$$\Big\langle E \cup \big\{t \doteq y\big\}, \sigma\Big\rangle \quad \rightsquigarrow \quad \Big\langle E \cup \big\{y \doteq t\big\}, \sigma\Big\rangle \qquad \text{if } y \in \mathcal{V} \text{ and } t \notin \mathcal{V}.$$

   After we apply this rule, we can apply either the first or the second reduction rule thereafter.

4. Trivial syntactical equations can be deleted:

$$\Big\langle E \cup \big\{x \doteq x\big\}, \sigma\Big\rangle \quad \rightsquigarrow \quad \Big\langle E, \sigma\Big\rangle \qquad \text{if } x \in \mathcal{V}.$$

5. If f is an n-ary function symbol we have

$$\Big\langle E \cup \big\{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\big\}, \sigma\Big\rangle \rightsquigarrow \Big\langle E \cup \big\{s_1 \doteq t_1, \cdots, s_n \doteq t_n\big\}, \sigma\Big\rangle.$$

   This rule is the reason that we have to work with a system of syntactical equations, because even if we start with a single syntactical equation the rule given above can increase the number of syntactical equations.

   A special case of this rule is the following:

$$\Big\langle E \cup \big\{c \doteq c\big\}, \sigma\Big\rangle \rightsquigarrow \Big\langle E, \sigma\Big\rangle.$$

   Here c is a nullary function symbol.

6. The system of syntactical equations $E \cup \big\{ f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n) \big\}$ has no solution if the function symbols f and g are different. Hence we have
$$\Big\langle E \cup \big\{ f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n) \big\}, \sigma \Big\rangle \rightsquigarrow \Omega \qquad \text{provided } f \neq g.$$

If a system of syntactical equations E is given and we start with the pair $\langle E, \{\} \rangle$, then we can apply the rules given above until one of the following two cases happens:

1. We use the second or the sixth of the reduction rules given above. In this case the system of syntactical equations E is unsolvable.
2. The pair $\langle E, \{\} \rangle$ is reduced into a pair of the form $\langle \{\}, \mu \rangle$. Then μ is a unifier of E. In this case we write $\mu = \mathtt{mgu}(E)$. If $E = \{ s \doteq t \}$, we write $\mu = \mathtt{mgu}(s, t)$. The abbreviation `mgu` is short for "most general unifier".

Example: We show how to solve the syntactical equation
$$p(x_1, f(x_4)) \doteq p(x_2, x_3).$$
We have the following reductions:
$$\begin{array}{ll} & \big\langle \big\{ p(x_1, f(x_4)) \doteq p(x_2, x_3) \big\}, \{\} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ x_1 \doteq x_2, f(x_4) \doteq x_3 \big\}, \{\} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ f(x_4) \doteq x_3 \big\}, \{ x_1 \mapsto x_2 \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ x_3 \doteq f(x_4) \big\}, \{ x_1 \mapsto x_2 \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ \big\}, \{ x_1 \mapsto x_2,\ x_3 \mapsto f(x_4) \} \big\rangle \end{array}$$
Hence the method is successful and we have that the substitution
$$\{ x_1 \mapsto x_2,\ x_3 \mapsto f(x_4) \}$$
is a solution of the syntactical equation given above. ◇

Example: Next we try to solve the following system of syntactical equations:
$$E = \big\{ p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4) \big\}$$
We have the following reductions:
$$\begin{array}{ll} & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4) \big\}, \{\} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ d \doteq x_4 \big\}, \{\} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ x_4 \doteq d \big\}, \{\} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c) \big\}, \{ x_4 \mapsto d \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ p(h(x_1, c)) \doteq p(h(d, c)) \big\}, \{ x_4 \mapsto d,\ x_2 \mapsto h(d, c) \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ h(x_1, c) \doteq h(d, c) \big\}, \{ x_4 \mapsto d,\ x_2 \mapsto h(d, c) \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ x_1 \doteq d,\ c \doteq c \big\}, \{ x_4 \mapsto d,\ x_2 \mapsto h(d, c) \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ x_1 \doteq d, \big\}, \{ x_4 \mapsto d,\ x_2 \mapsto h(d, c) \} \big\rangle \\ \rightsquigarrow & \big\langle \big\{ \big\}, \{ x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d \} \big\rangle \end{array}$$
Hence the substitution $\{ x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d \}$ is a solution of the system of syntactical equations given above. ◇

5.5 The Knuth-Bendix Algorithm

Assume we have been given a set R of rewrite rules such that

$$r \prec l \quad \text{holds for all } l \approx r \text{ in } R$$

such that the relation $\prec$ is a rewrite order. Given two terms s and t, the Church-Rosser Theorem tells us, that we can decide the question whether $s \leftrightarrow_R^* t$ holds by rewriting s and t into normal forms, provided the relation $\to_R$ is confluent. By Newman's Lemma we know that local confluence is sufficient. Donald E. Knuth and Peter B. Bendix [KB70] have discovered a way to decide whether the term rewriting relation $\to_R$ is locally confluent. To understand their idea, we introduce the notion of a critical pair.

Definition 42 (Critical Pair)
Assume we have been given the equations $l_1 \approx r_1$ and $l_2 \approx r_2$. These equations generate a critical pair if and only if the following conditions hold:

(a) There exists a position $u \in \mathcal{P}\text{os}(l_1)$ such that l_1/u is not a variable.

(b) The subterm l_1/u of l_1 is unifiable with l_2. For the following, assume that μ is a most general unifier of l_1/u and l_2, i.e. we have

$$\mu = \mathtt{mgu}(l_1/u, l_2).$$

(c) The term s results from rewriting the term $l_1\mu$ by rewriting the subterm $l_1\mu/u$ to the new subterm $r_2\mu$ using the rewrite rule $l_2 \approx r_2$:

$$s = l_1\mu[u \mapsto r_2\mu].$$

(d) The term t results from rewriting the term $l_1\mu$ into the term $r_1\mu$ using the rule $l_1 \approx r_1$, i.e. we have

$$t = r_1\mu.$$

Then the pair $\langle s, t\rangle$, which is

$$\langle l_1\mu[u \mapsto r_2\mu],\ r_1\mu\rangle$$

is a critical pair of $l_1 \approx r_1$ and $l_2 \approx r_2$. $\diamond$

Example: The following example assumes the signature Σ_G from group theory as given. We start with the two equations $(x \circ y) \circ z \approx x \circ (y \circ z)$ and $i(w) \circ w \approx e$. Then $u = [1]$ is a position in the term $(x \circ y) \circ z$ and we have

$$\big((x \circ y) \circ z\big)/[1] = x \circ y, \text{ which is not a variable.}$$

The term $x \circ y$ can be unified with the term $i(w) \circ w$ and we have

$$\mu := \mathtt{mgu}\big(x \circ y, i(w) \circ w\big) = \{x \mapsto i(w),\ y \mapsto w\}.$$

Therefore we have

$$\big((x \circ y) \circ z\big)\mu = \big(i(w) \circ w\big) \circ z$$

and the right hand side of this equations can be rewritten by the equation $i(w) \circ w \approx e$ into the term $e \circ z$, i.e. we have

$$\big(i(w) \circ w\big) \circ z \to_{\{i(w)\circ w \approx e\}} e \circ z.$$

Furthermore, the same term $\big(i(w) \circ w\big) \circ z$ can be rewritten by the equation $(x \circ y) \circ z \approx x \circ (y \circ z)$

into the term $i(w) \circ (w \circ z)$:

$$\big(i(w) \circ w\big) \circ z \rightarrow_{\{(x \circ y) \circ z \approx x \circ (y \circ z)\}} i(w) \circ (w \circ z).$$

Therefore, the pair

$$\big\langle e \circ z,\ i(w) \circ (w \circ z)\big\rangle$$

is a critical pair of the two equations $(x \circ y) \circ z \approx x \circ (y \circ z)$ and $i(w) \circ w \approx e$. $\diamond$

Remark: If $\langle s, t\rangle$ is a critical pair from two equations in a set R, then the equation $s \approx t$ is a logical consequence of the equations from R, i.e. we have

$$R \models s \approx t. \qquad \diamond$$

Definition 43 (Confluent Critical Pair)
A critical pair $\langle s_1, s_2\rangle$ is confluent w.r.t. a rewrite relation R iff there is a term t such that we have both

$$s_1 \rightarrow_R^* t \quad \text{and} \quad s_2 \rightarrow_R^* t.$$

Theorem 44 (Knuth-Bendix) If R is a set of rewrite equations such that all critical pairs between equations from R are confluent, then the rewrite relation $\rightarrow_R^*$ is confluent and hence the question, whether $R \models s \approx t$ can be decided by rewriting both s and t into normal forms $\widehat{s}$ and $\widehat{t}$:

$$s \rightarrow_R^* \widehat{s} \quad \text{and} \quad t \rightarrow_R^* \widehat{t}$$

Then we have

$$R \models s \approx t \quad \text{if and only if} \quad \widehat{s} = \widehat{t}.$$

To make the above theorem work, if we start with a set E of equations, we first have to order them into a set of rewrite rules R. In general, this will not be sufficient because there will be critical pairs that are not confluent. However, if we can orient these newly derived critical pairs into new rewrite rules, we might be able to extend the set R to a new set of rewrite $\widehat{R}$ such that all critical pairs from equations from $\widehat{R}$ are confluent.

Knuth-Bendix Algorithm: Given a set of equations E the Knuth-Bendix algorithm proceeds as follows:

1. We define a suitable Knuth-Bendix order $\langle w, <\rangle$ for the function symbols occurring in E such that every equation $(s \approx t) \in E$ can be ordered as either $s \prec t$ or $t \prec s$. If this is not possible, the algorithm fails.
2. Otherwise, call R the set of oriented rewrite rules that result from orienting the equations in E into rewrite rules.
3. Compute all critical pairs that can be build from equations in R.
   (a) If all critical pairs are confluent, then the rewrite relation $\rightarrow_R^*$ is confluent and the algorithm is successful.
   (b) If we have found a critical pair $\langle s, t\rangle$ that is not confluent, we use the rewrite rules to simplify s and t into terms $\widehat{s}$ and $\widehat{t}$ that are in normal form with respect to $\rightarrow_r$.
   (c) Then we try to orient the equation $\widehat{s} \approx \widehat{t}$ into a rewrite rule $l \approx r$ such that $r \prec l$.
   (d) If this is impossible, the algorithm fails.
   (e) Otherwise, we add the equation $l \approx r$ to R:
   $$R := R \cup \{l \approx r\}.$$

(f) The newly added equation could generate additional critical pairs. Hence we must go back to the beginning of step 3. $\diamond$

The algorithm shown above can have three different outcomes:

1. It can fail because it has generated an equation that can not be oriented into a rewrite rule.
2. It can stop with a set of rewrite rules R such that $\rightarrow_R$ is confluent.
3. It can run forever because an infinite set of critical pairs is generated.

My GitHub repository contains the Jupyter notebook

1-Knuth-Bendix-Algorithm-KBO.ipynb

which contains an implementation of the Knuth-Bendix algorithm. It also contains a number of equational theories E where the Knuth-Bendix algorithm is successful.

Example: We test the Knuth-Bendix algorithm with the axioms of group theory. We extend the signature of group theory to contain all lowercase letters

$$a, b, c, d, e, f, g, h, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z$$

with the exception of i as variables. Then we have to denote the neutral element as 1, since e is now a variable. The multiplication is still denoted as $\circ$, and $i(x)$ denotes the inverse of x. Then the axioms are:

(a) $1 \circ x \approx x$,

(b) $i(x) \circ x \approx 1$, and

(c) $(x \circ y) \circ z \approx x \circ (y \circ z)$.

Using the Knuth-Bendix ordering described previously, we can turn these equations into rewrite rules as follows:

- $1 \circ x \rightarrow x$,
- $i(x) \circ x \rightarrow 1$, and
- $(x \circ y) \circ z \rightarrow x \circ (y \circ z)$.

By taking the rewrite rule $(x \circ y) \circ z \rightarrow x \circ (y \circ z)$ and superposing the rewrite rule $i(a) \circ a \rightarrow 1$ at position [1], utilizing the most general unifier $[x \mapsto i(a), y \mapsto a]$, we deduce the following relationships:

$$(i(a) \circ a) \circ z \rightarrow i(a) \circ (a \circ z) \quad \text{and}$$
$$(i(a) \circ a) \circ z \rightarrow 1 \circ z.$$

As $1 \circ z$ can be rewritten to z and $z \prec i(a) \circ (a \circ z)$ in the Knuth-Bendix ordering, we have found the new rewrite rule

$$i(a) \circ (a \circ z) \rightarrow z.$$

By taking the rewrite rule $i(a) \circ (a \circ z) \rightarrow z$ and superposing the rewrite rule $i(x) \circ x \rightarrow 1$ at position [2] using the most general unifier

$$\texttt{mgu}\big(i(x) \circ x, a \circ z\big) = \big[a \mapsto i(x), z \mapsto x\big]$$

we conclude

$$i(i(x)) \circ \bigl(i(x) \circ x\bigr) \rightarrow x \quad \text{and} \quad i\bigl(i(x)\bigr) \circ \bigl(i(x) \circ x\bigr) \rightarrow i\bigl(i(x)\bigr) \circ 1.$$

Hence we have found the new rewrite rule

$$i(i(x)) \circ 1 \rightarrow x.$$

In the next step, we build a critical pair between a rule $l \rightarrow r$ and the same rule $l \rightarrow r$. However, note that we hav to rename the variables in the second instand of the rule to prevent accidental name clashes. We take the rewrite rule $i(x) \circ (x \circ y) \rightarrow y$ and superposing the (renamed) rewrite rule $i(a) \circ (a \circ b) \rightarrow b$ at position [2] using the most general unifier

$$\texttt{mgu}\bigl(x \circ y, i(a) \circ (a \circ b)\bigr) = \bigl[x \mapsto i(a), y \mapsto a \circ b\bigr].$$

This gives

$$i\bigl(i(a)\bigr) \circ \bigl(i(a) \circ (a \circ b)\bigr) \rightarrow a \circ b \quad \text{and} \quad i\bigl(i(a)\bigr) \circ \bigl(i(a) \circ (a \circ b)\bigr) \rightarrow i\bigl(i(a)\bigr) \circ b.$$

Since $a \circ b \prec i\bigl(i(a)\bigr) \circ b$ we have found the new rewrite rule

$$i\bigl(i(a)\bigr) \circ b \rightarrow a \circ b.$$

By taking the rewrite rule $i\bigl(i(a)\bigr) \circ b \rightarrow a \circ b$ and superposing the rewrite rule $i(i(x)) \circ 1 \rightarrow x$ at position [] using the most general unifier

$$\texttt{mgu}\bigl(i(i(a)) \circ b, i(i(x)) \circ 1\bigr) = \bigl[x \mapsto a, b \mapsto 1\bigr]$$

we conclude

$$i\bigl(i(a)\bigr) \circ 1 \rightarrow a \circ 1 \quad \text{and} \quad i\bigl(i(a)\bigr) \circ 1 \rightarrow a$$

Hence we have found the new rewrite rule

$$a \circ 1 \rightarrow a.$$

At this point, the rewrite rule $i\bigl(i(x)\bigr) \circ 1 \rightarrow x$ is simplified into the rule

$$i\bigl(i(x)\bigr) \rightarrow x.$$

By taking the rewrite rule $i(x) \circ (x \circ y) \rightarrow y$ and superposing the rewrite rule $i\bigl(i(a)\bigr) \rightarrow a$ at position [1] using the most general unifier

$$\texttt{mgu}\bigl(i(x), i(i(a))\bigr) = \bigl[x \mapsto i(a)\bigr],$$

we conclude

$$i\bigl(i(a)\bigr) \circ \bigl(i(a) \circ y\bigr) \rightarrow y \quad \text{and} \quad i\bigl(i(a)\bigr) \circ \bigl(i(a) \circ y\bigr) \rightarrow a \circ \bigl(i(a) \circ y\bigr).$$

Hence we have found the new rule

$$a \circ \bigl(i(a) \circ y\bigr) \rightarrow y.$$

By taking the rewrite rule $i(x) \circ x \rightarrow 1$ and superposing the rewrite rule $i\bigl(i(a)\bigr) \rightarrow a$ at position [1] using the most general unifier

$$\texttt{mgu}\bigl(i(x), i(i(a))\bigr) = \bigl[x \mapsto i(a)\bigr]$$

we conclude

$$i\bigl(i(a)\bigr) \circ i(a) \rightarrow 1 \quad \text{and} \quad i\bigl(i(a)\bigr) \circ i(a) \rightarrow a \circ i(a).$$

Hence we have found the new rule

$$a \circ i(a) \rightarrow 1.$$

By taking the rewrite rule $a \circ i(a) \rightarrow 1$ and superposing the rewrite rule $1 \circ x \rightarrow x$ at position [] using the most general unifier

$$\texttt{mgu}\big(a \circ i(a), 1 \circ x\big) = \big[a \mapsto 1, x \mapsto i(1)\big],$$

we conclude

$$1 \circ i(1) \rightarrow 1 \quad \text{and} \quad 1 \circ i(1) \rightarrow i(1).$$

Hence we have shown

$$i(1) \rightarrow 1.$$

By taking the rewrite rule $a \circ i(a) \rightarrow 1$ and superposing the rewrite rule $(x \circ y) \circ z \rightarrow x \circ (y \circ z)$ at position [] using the most general unifier

$$\texttt{mgu}\big(a \circ i(a), (x \circ y) \circ z\big) = \big[a \mapsto x \circ y, z \rightarrow i(x \circ y\big],$$

we conclude

$$(x \circ y) \circ i\big(x \circ y) \rightarrow 1 \quad \text{and} \quad (x \circ y) \circ i\big(x \circ y) \rightarrow x \circ \big(y \circ i(x \circ y)\big).$$

Hence we have found the new rule

$$x \circ \big(y \circ i(x \circ y)\big) \rightarrow 1.$$

By taking the rewrite rule $a \circ \big(i(a) \circ b\big) \rightarrow b$ and superposing the rewrite rule $x \circ \big(y \circ i(x \circ y)\big) \rightarrow 1$ at position [2] using the most general unifier

$$\texttt{mgu}\Big(i(a) \circ b, x \circ \big(y \circ i(x \circ y)\big)\Big) = \Big[x \mapsto i(a), b \mapsto y \circ i\big(i(a) \circ y\big)\Big],$$

we conclude

$$a \circ \Big(i(a) \circ \Big(y \circ i\big(i(a) \circ y\big)\Big)\Big) \rightarrow y \circ i\big(i(a) \circ y\big) \quad \text{and} \quad a \circ \Big(i(a) \circ \Big(y \circ i\big(i(a) \circ y\big)\Big)\Big) \rightarrow a \circ 1.$$

As we already know that $a \circ 1 \rightarrow a$ we have found the new rule

$$y \circ i\big(i(a) \circ y\big) \rightarrow a.$$

By taking the rewrite rule $y \circ i\big(i(a) \circ y\big) \rightarrow a$ and superposing the rewrite rule $i\big(i(x)\big) \rightarrow x$ at position [2, 1, 1] using the most general unifier

$$\texttt{mgu}\big(i(a), i(i(x)),\big) = \big[a \mapsto i(x)\big],$$

we conclude

$$y \circ i\Big(i\big(i(x)\big) \circ y\Big) \rightarrow i(x) \quad \text{and} \quad y \circ i\Big(i\big(i(x)\big) \circ y\Big) \rightarrow y \circ i(x \circ y).$$

Hence we have found the new rule

$$y \circ i(x \circ y) \rightarrow i(x).$$

By taking the rewrite rule $i(a) \circ (a \circ b) \rightarrow b$ and superposing the rewrite rule $y \circ i(x \circ y) \rightarrow i(x)$ at the position [2] using the most general unifier

$$\texttt{mgu}\big(a \circ b, y \circ i(x \circ y)\big) = \big[y \mapsto a, b \mapsto i(x \circ a)\big],$$

we conclude

$$i(a) \circ \Big(a \circ i\big(x \circ a\big)\Big) \rightarrow i(x \circ a) \quad \text{and} \quad i(a) \circ \Big(a \circ i\big(x \circ a\big)\Big) \rightarrow i(a) \circ i(x).$$

Since $i(a) \circ i(y) \prec i(x \circ a)$ Hence we have found the new rule

$$i(x \circ a) \rightarrow i(a) \circ i(x).$$

This last rule makes the rules

$$y \circ i(x \circ y) \rightarrow i(x), \quad y \circ i\big(i(a) \circ y\big) \rightarrow a, \quad \text{and} \quad x \circ \big(y \circ i(x \circ y)\big) \rightarrow 1$$

redundant as all of these rules can be simplified to an identity using the rule $i(x \circ a) \rightarrow i(a) \circ i(x)$. Therefore, we have found the following set of rewrite rules.

1. $1 \circ x \rightarrow x$,
2. $i(x) \circ x \rightarrow 1$,
3. $(x \circ y) \circ z \rightarrow x \circ (y \circ z)$,
4. $i(a) \circ (a \circ z) \rightarrow z$.
5. $a \circ 1 \rightarrow a$,
6. $i(1) \rightarrow 1$,
7. $i\bigl(i(x)\bigr) \rightarrow x$,
8. $a \circ i(a) \rightarrow 1$,
9. $a \circ \bigl(i(a) \circ y\bigr) \rightarrow y$,
10. $i(x \circ a) \rightarrow i(a) \circ i(x)$.

It can be shown that all critical pairs resulting from these rules can be simplified to identities. Hence this set is a confluent set of rewrite rules for group theory. Therefore, the validity of any equation $s \approx t$ in group theory can be checked by rewriting s and t into normal forms using the rewrite rules given above. Then the equation $s \approx t$ is valid in group theory if and only if the normal forms of s and t are identical. $\diamond$

Exercise 14: A quasi-group is a structure

$$\mathcal{G} = \langle G, \circ, /, \backslash \rangle$$

such that

1. G is a non-empty set,
2. $\circ : G \times G \rightarrow G$,
3. $/ : G \times G \rightarrow G$,
4. $\backslash : G \times G \rightarrow G$.
5. Furthermore, the following axioms have to be satisfied:
   (a) $x \circ (x \backslash y) = y$,
   (b) $(x / y) \circ y = x$,
   (c) $x \backslash (x \circ y) = y$,
   (d) $(x \circ y) / y = x$.

Compute the set of all non-trivial critical pairs from these equations.

Hint: The two non-trivial critical pairs arise from trying to simplify the left hand side of equation (d) with equation (a) and from simplifying the left hand side of equation (c) with (b). $\diamond$

5.6 Literature

The book Term Rewriting and All That by Franz Baader and Tobias Nipkow [BN98] gives a much more detailed account of equational theorem proving via term rewriting.

Chapter 6

Automatic Differentiation

All modern libraries for Neural networks, i.e. TensorFlow and PyTorch make heavy use of automatic differentiation. Automatic differentiation is a technique for computing the gradient of a function that does neither rely on numeric approximation nor does it force us to compute symbolic derivatives manually. In fact, the technique is one of the major methodological breakthroughs in machine learning in particular and science and engineering in general in recent years. Although the idea was first published in 1964 by R. E. Wengert [Wen64], it has only been widely understood and accepted in recent years, cf. Baydin et. al. [BPRS18].

There are two modes of automatic differentiation: Forward mode and reverse mode. Forward mode is quite inefficient when the number n of input variables is big. The reason is that forward mode needs to traverse the computational graph $n+1$ times (once to compute the values and n times to compute the partial derivatives), while reverse mode needs to traverse the computational graph just twice. Hence for big values of n only reverse mode automatic differentiation is a viable option. We proceed to define the crucial notion of a computational graph.

Definition 45 (Computational Graph) A computational graph is a list of computational nodes. There are four types of computational nodes:

1. A variable node is tuple of length 1 of the form

   $$\langle x, \rangle$$

   where x is a variable from the set of variables $\{x_1, \cdots, x_k\}$. This node represents the given input variable.

2. A constant node is a pair of the form

   $$\langle n, r\rangle$$

   where n is the name of the node and r is a floating point number. This node is interpreted as the assignment

   $$n := r.$$

   The name n is a string that can be understood as the name of an auxiliary variable.

3. A unary node is a tuple of the form

   $$\langle n, f, a\rangle$$

   where, again, n is the name of the node that is used as an auxiliary variable. f is an unary function symbol from a set of unary function. In our examples f will be a mwember of the set

$$\{\text{sqrt}, \exp, \ln, \sin, \cos, \arctan\}$$

and a is the name of another node occurring in the list. This node is interpreted as the assignment

$$n := f(a).$$

4. A binary node is a tuple of the form

   $$\langle n, o, a_1, a_2 \rangle$$

   where n is the name of the node, o is a binary operator from the set

   $$\{+, -, *, /\}$$

   and a_1 and a_2 are the names of computational nodes.

   This node is interpreted as the assignment

   $$n := a_1 \ o \ a_2.$$

   As in the previous case, the name n is a string that can be understood as the name of an auxiliary variable. ◇

```
CG = [ ('x1', ),
       ('x2', ),
       ('v1', '+',   'x1', 'x2'),
       ('v2', '-',   'x1', 'x2'),
       ('v3', 'sin', 'v1'),
       ('v4', 'cos', 'v2'),
       ('v5', '*',   'v3', 'v4'),
       ('v6', '*',   'v1', 'v2'),
       ('y',  '+',   'v5', 'v6')
     ]
```

Figure 6.1: A Computational Graph for $\sin(x_1 + x_2) \cdot \cos(x_1 - x_2) + (x_1 + x_2) \cdot (x_1 - x_2)$.

Example: Figure 6.1 shows a computational graph for the expression

$$\sin(x_1 + x_2) \cdot \cos(x_1 - x_2) + (x_1 + x_2) \cdot (x_1 - x_2).$$

This computational graph uses the input variables `x1` and `x2`. Figure 6.2 shows a rendering of this computational graph. ◇

Definition 46 (admissible) A computational graph G is admissible if for every node of the form

$$\langle n, o, a_1, a_2 \rangle$$

that occurs in the list G there are nodes labelled with a_1 and a_2 that occur in the list G before this node and for every node of the form

$$\langle n, f, a \rangle$$

that occurs in the list G there is a node labelled with a that occurs in the list G before this node. If a computational graph is admissible, the nodes can be evaluated in the same order as they are listed in G. ◇

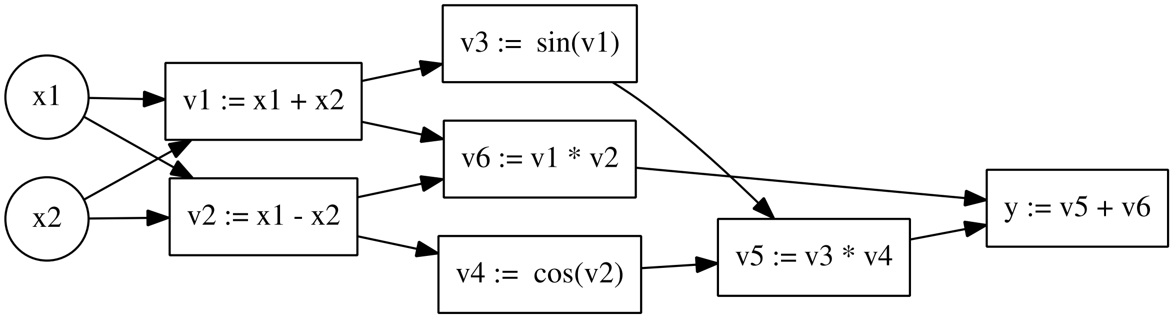


Figure 6.2: Rendering of the computation graph shown in Figure 6.1.

```
def eval_graph(CG, Values):
    for node in CG:
        match node:
            case (v, ):
                pass
            case (v, r):
                Values[v] = r
            case (v, '+', a1, a2):
                Values[v] = Values[a1] + Values[a2]
            case (v, '-', a1, a2):
                Values[v] = Values[a1] - Values[a2]
            case (v, '*', a1, a2):
                Values[v] = Values[a1] * Values[a2]
            case (v, '/', a1, a2):
                Values[v] = Values[a1] / Values[a2]
            case (v, 'sqrt', a):
                Values[v] = math.sqrt(Values[a])
            case (v, 'exp', a):
                Values[v] = math.exp(Values[a])
            case (v, 'log', a):
                Values[v] = math.log(Values[a])
            case (v, 'sin', a):
                Values[v] = math.sin(Values[a])
            case (v, 'cos', a):
                Values[v] = math.cos(Values[a])
            case (v, 'atan', a):
                Values[v] = math.atan(Values[a])
    return Values['y']
```

Figure 6.3: A function that evaluates a computational graph.

In order to evaluate an admissible computational graph that contains n variables, we will assume that the first n nodes are labelled with the variables $\mathtt{x}_1, \cdots, \mathtt{x}_n$ and that the last node in a computational node is labelled with the name `y`. Furthermore, we need a dictionary `Values` that assigns a value to each of the variables $\mathtt{x}_1, \cdots, \mathtt{x}_n$. Then the function `eval_graph` that is shown in Figure 6.3

can be used to evaluate the nodes of the computational graph `CG` one by one. The idea is that initially the dictionary `Values` maps all variables to floating point values. Then the nodes of the computational graph are evaluated one by one. For example, if a node of the form

$$\langle v, \texttt{'+'}, a_1, a_2 \rangle$$

has to be evaluated, then we can assume that the nodes that are labelled with a_1 and a_2 have already been evaluated and that their values are stored in the dictionary `Values`. These values are added and the resulting value is stored under the key v in the dictionary `Values`.

In the following we will assume that all computational graphs are admissible. The crucial definition in the theory of reverse mode automatic differentiation is the notion of an adjoint, which will be given later after we have defined to notion of a parent of a node.

```
def parents(CG):
    Parents = {}
    for node in CG:
        match node:
            case (p, _, a):
                add_to_dictionary(Parents, a, p)
            case (p, _, a1, a2):
                add_to_dictionary(Parents, a1, p)
                add_to_dictionary(Parents, a2, p)
    return Parents

def add_to_dictionary(D, key, value):
    if key in D:
        D[key] |= { value }
    else:
        D[key]  = { value }

def node_dictionary(CG):
    D = {}
    for node in CG:
        name    = node[0]
        D[name] = node
    return D
```

Figure 6.4: Auxiliary functions.

Definition 47 (Parent) If G is a computational graph and $\langle v, o, a_1, a_2 \rangle$ is a node in G, then v is a parent of the nodes that are labelled with a_1 and a_2. Furthermore, if $\langle v, f, a \rangle$ is a node in G, then v is a parent of the node that is labelled with a. $\diamond$

Figure 6.4 shows the implementation of the function `parents` that can be used to compute the parents of a node. It also contains the auxiliary function `node_dictionary` that takes a computational graph `CG` as its argument and returns a dictionary associating every node with its name.

Definition 48 (Adjoint) Assume G is a computational graph such that the last node is labelled with then name $\mathtt{y}$. If v is any node in G, then the adjoint of v, which is written as $\bar{v}$, is defined as the partial derivative of the output variable $\mathtt{y}$ w.r.t. v, i.e.

$$\bar{v} := \frac{\partial \mathtt{y}}{\partial v}. \qquad \diamond$$

The next theorem is an immediate consequence of the multivariable chain rule.

Theorem 49 Assume v is a node of a computational graph G and that $p_1, \cdots, p_k$ are all the parents of this node in G. Then the adjoint $\bar{v}$ of the node v is given as

$$\bar{v} = \frac{\partial y}{\partial v} = \sum_{i=1}^{k} \frac{\partial y}{\partial p_i} \cdot \frac{\partial p_i}{\partial v} = \sum_{i=1}^{k} \bar{p}_i \cdot \frac{\partial p_i}{\partial v}.$$

Example: To keep things simple, assume that the variables $\mathtt{x}_1$ and $\mathtt{x}_2$ that are shown in the computational graph in Figure 6.2 are both initialized with the value $\pi/4$. Before the adjoints can be computed, we have to compute the values associated with the nodes. These are as follows:

1. $\mathtt{v}_1 = \pi/2$,
2. $\mathtt{v}_2 = 0$,
3. $\mathtt{v}_3 = 1$,
4. $\mathtt{v}_4 = 1$,
5. $\mathtt{v}_5 = 1$,
6. $\mathtt{v}_6 = 0$,
7. $\mathtt{y} = 1$.

Next, we compute the adjoints.

1. $\bar{\mathtt{y}} = \dfrac{\partial \mathtt{y}}{\partial \mathtt{y}} = 1$,
2. $\bar{\mathtt{v}}_6 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_6} = 1$,
3. $\bar{\mathtt{v}}_5 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_5} = 1$.
4. $\bar{\mathtt{v}}_4 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_5} \cdot \dfrac{\partial \mathtt{v}_5}{\partial \mathtt{v}_4} = \bar{\mathtt{v}}_5 \cdot \mathtt{v}_3 = 1 \cdot 1 = 1$.
5. $\bar{\mathtt{v}}_3 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_5} \cdot \dfrac{\partial \mathtt{v}_5}{\partial \mathtt{v}_3} = \bar{\mathtt{v}}_5 \cdot \mathtt{v}_4 = 1 \cdot 1 = 1$.
6. $\bar{\mathtt{v}}_2 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_6} \cdot \dfrac{\partial \mathtt{v}_6}{\partial \mathtt{v}_2} + \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_4} \cdot \dfrac{\partial \mathtt{v}_4}{\partial \mathtt{v}_2} = \bar{\mathtt{v}}_6 \cdot \mathtt{v}_1 - \bar{\mathtt{v}}_4 \cdot \sin(\mathtt{v}_2) = 1 \cdot \pi/2 - 1 \cdot \sin(0) = \pi/2 - 1 \cdot 0 = \pi/2$.
7. $\bar{\mathtt{v}}_1 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_6} \cdot \dfrac{\partial \mathtt{v}_6}{\partial \mathtt{v}_1} + \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_3} \cdot \dfrac{\partial \mathtt{v}_3}{\partial \mathtt{v}_1} = \bar{\mathtt{v}}_6 \cdot \mathtt{v}_2 + \bar{\mathtt{v}}_3 \cdot \cos(\mathtt{v}_1) = 1 \cdot 0 + 1 \cdot \cos(\pi/2) = 0 + 0 = 0$.
8. $\bar{\mathtt{x}}_1 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_1} \cdot \dfrac{\partial \mathtt{v}_1}{\partial \mathtt{x}_1} + \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_2} \cdot \dfrac{\partial \mathtt{v}_2}{\partial \mathtt{x}_1} = \bar{\mathtt{v}}_1 \cdot 1 + \bar{\mathtt{v}}_2 \cdot 1 = 0 \cdot 1 + \pi/2 \cdot 1 = \pi/2$.

9. $\bar{\mathtt{x}}_2 = \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_1} \cdot \dfrac{\partial \mathtt{v}_1}{\partial \mathtt{x}_2} + \dfrac{\partial \mathtt{y}}{\partial \mathtt{v}_2} \cdot \dfrac{\partial \mathtt{v}_2}{\partial \mathtt{x}_2} = \bar{\mathtt{v}}_1 \cdot 1 + \bar{\mathtt{v}}_2 \cdot (-1) = 0 \cdot 1 + \pi/2 \cdot (-1) = -\pi/2.$

Hence we have shown the following:

$$\frac{\partial \mathtt{y}}{\partial \mathtt{x}_1}\Big(\frac{\pi}{4}, \frac{\pi}{4}\Big) = \frac{\pi}{2} \quad \text{and} \quad \frac{\partial \mathtt{y}}{\partial \mathtt{x}_2}\Big(\frac{\pi}{4}, \frac{\pi}{4}\Big) = -\frac{\pi}{2}.$$

Note that we have found the exact partial derivatives for a specific point, namely for the arguments $\mathtt{x}_1 = \pi/4$ and $\mathtt{x}_2 = \pi/4$. Automatic differentiation is not symbolic differentiation and hence is not able to derive general formulas but rather computes values for specific arguments. However, these values are not numerical approximations but are, instead, exact. ⋄

Of course, we do not want to perform computations like the following ourselves. The function `partial_derivative` shown in Figure 6.5 takes a computational `Node` and computes the partial derivative of this node with respect to the given argument `arg`. The last argument `Values` is a dictionary containing the values that are associated with the different nodes.

```python
def partial_derivative(Node, arg, Values):
    match Node:
        case n, '+', a1, a2:
            if arg == a1 == a2:
                return 2
            if arg == a1 or arg == a2:
                return 1
        case n, '-', a1, a2:
            if arg == a1 == a2:
                return 0
            if arg == a1:
                return 1
            if arg == a2:
                return -1
        case n, '*', a1, a2:
            if arg == a1 == a2:
                return 2 * Values[a1]
            if arg == a1:
                return Values[a2]
            if arg == a2:
                return Values[a1]
        case n, '/', a1, a2:
            if arg == a1 == a2:
                return 0
            if arg == a1:
                return 1 / Values[a2]
            if arg == a2:
                return -Values[a1] / Values[a2] ** 2
        case n, 'sqrt', a:
            return 0.5 / math.sqrt(Values[a])
        case n, 'exp', a:
            return math.exp(Values[a])
        case n, 'log', a:
            return math.log(Values[a])
        case n, 'sin', a:
            return math.cos(Values[a])
        case n, 'cos', a:
            return -math.sin(Values[a])
        case n, 'atan', a:
            return 1 / (1 + Values[a]**2)
```

Figure 6.5: Computing the partial derivative of a node.

The function `adjoints` shown in Figure 6.6 computes the adjoints of a given computational graph. It needs a dictionary `Values` that maps the variables $x_1, \cdots, x_n$ to their values. It returns a dictionary that associates all node names with their adjoints.

```
def adjoints(CG, Values):
    eval_graph(CG, Values)
    NodeDict = node_dictionary(CG)
    Parents  = parents(CG)
    n        = len(CG)
    Adjoints = {}
    Adjoints['y'] = 1
    for k in range(2, n+1):
        Node   = CG[-k]
        name   = Node[0]
        result = 0
        for parent_name in Parents[name]:
            parent_node = NodeDict[parent_name]
            pd          = partial_derivative(parent_node, name, Values)
            result += Adjoints[parent_name] * pd
        Adjoints[name] = result
    return Adjoints
```

Figure 6.6: Computing the adjoints of a computational graph.

6.1 The Library `autograd`

The library autograd implements the theory shown in the previous section. We will introduce this library via a couple of simple examples.

```
1 import autograd       as ag
2 import autograd.numpy as np
3 
4 def f(x):
5     return x * np.exp(x)
6 
7 fs = ag.grad(f)
8 
9 print(fs(1.0))
```

Figure 6.7: A simple example demonstrating ‘autograd‘.

Figure 6.7 on page 136 shows a simple example of the usage of this library.

1. In line 1 we import the library `autograd` and introduce the abbreviation `ag`.
2. Next, we have to import the autograd version of the library `numpy`. This version offers most, but not all features of `numpy`. We have to use this version because `autograd` works by creating a computational graph behind the scene. If we would use the standard version of `numpy`, which

is implemented outside of *Python* in the programming language `C`, `autograd` would not be able be able to create this computational graph.

3. Next we define the function $f := x \mapsto x \cdot \exp(x)$. According to the product rule, the derivative of f is given as
$$\frac{\mathrm{d}f}{\mathrm{d}x} = \exp(x) + x \cdot \exp(x).$$
4. Line 7 shows how we can implement the derivative of `f` without any knowledge of mathematical analysis. This can be done by calling the function `grad` from the library `autograd` and supplying the function `f` as its argument.
5. The function `fs` that is generated by `autograd` can than be called just like any other Python function.

The previous example isn't too surprising. After all, we can do similar things using the library `SymPy`, which is a Python library for doing symbolic mathematics. The real magic of `autograd` starts to happen when we take the derivative of a *Python* function that uses control structures like `while`-loops or `if`-statements. We proceed to give an example.

```
def mySqrt(x):
    root = x
    eps  = 2.0e-15
    while abs(x - root * root) > eps:
        root = 0.5 * (root + x / root)
    return root

mySqrtGrad = ag.grad(mySqrt)
```

Figure 6.8: The Babylonian method to compute the square root.

The program shown in Figure 6.8 on page 137 shows an implementation of the Babylonian method for computing square roots. The function `mySqrt` defines the sequence $(r_n)_{n\in\mathbb{N}}$ as

(a) $r_0 = \frac{1}{2} \cdot x$,

(b) $r_{n+1} = \frac{1}{2} \cdot \left(r_n + \frac{x}{r_n}\right) \quad$ for all $n \in \mathbb{N}$.

It can be show that this sequence converges quadratically to the square root of x, i.e. we have:
$$\lim_{n\to\infty} r_n = \sqrt{x}$$
We can compute the derivative of the function `mySqrt` by just calling `ag.grad(mySqrt)`. However, when doing this we discover a limitation of `autograd`: The derivative of the square root function is known to be
$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{1}{2 \cdot \sqrt{x}}.$$
When we evaluate `mySqrtGrad` this function returns the same values as the expression given above, with one exception. If $x = 1$, then `mySqrtGrad` returns 1, although the derivative is $\frac{1}{2}$. To understand what is going on let us investigate what happens when `mySqrt`(1) is computed.

1. `x` is set to 1 in line 1.
2. `root` is set to 1 in line 2.
3. Therefore, in line 4 the expression `x - root * root` yields 0 and the `while`-loop is not executed.
4. Finally, `root`, which is equal to `x` is returned.

Effectively, for the argument 1, the computational graph produced by `mySqrt` is the same as the computational graph of the identity function $\mathtt{id}(x) = x$. Hence, the derivative computed by `mySqrtGrad` for $x = 1$ is equal to the derivative of the identity function, which is 1. There is an easy fix to solve this problem: We just have to make sure that the `while`-loop is executed at least once. Figure 6.9 on page 138 shows the resulting implementation.

```
def mySqrt(x):
    root = x
    eps  = 2.0e-15
    while True:
        root = 0.5 * (root + x / root)
        if abs(x - root * root) < eps:
            return root
```

Figure 6.9: A version of `sqrt` that is correctly differentiated by `autograd`.

Bibliography

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BPRS18] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2):100–107, 1968.

[HNR72] Peter Hart, Nils Nilsson, and Bertram Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1972.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Kor85] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[MC82] Tony A. Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 4rd edition, 2020.

[Sam59] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):535–554, 1959.

[Wen64] R.E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

List of Figures

Index