An Introduction to Artificial Intelligence

— Lecture Notes in Progress —

Prof. Dr. Karl Stroetmann

March 19, 2017

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 repository containing the lecture notes can be cloned using the command

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

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 this is the first time that I give these lectures, these lecture notes are very incomplete and will be changed frequently during the semester.

Contents

Chapter 1

Introduction

1.1 What is Artificial Intelligence?

Before we start to dive into the subject of Artificial Intelligence we have to answer the following question:

What is *Artificial Intelligence*?

Historically, there have been a number of different answers to this question [5]. We will look at these different answers and discuss them.

1. *Artificial Intelligence* is the study of creating machines that think like humans.

   As we have a working prototype of intelligence, namely humans, it is quite natural to try to build machines that work in a way similar to humans, thereby creating artificial intelligence. As a first step in this endeavor we would have to study how humans actually think and thus we would have to study the brain. Unfortunately, as of today, no one really knows how the brain works. Although there are branches of science devoted to studying the human thought processes and the human brain, namely cognitive science and computational neuroscience, this approach has not proven to be fruitful for creating thinking machines, the reason being that the current understanding of the human thought processes is just not sufficient.

2. *Artificial Intelligence* is the science of machines that act like people.

   Since we do not know how humans think, we cannot build machines that think like people. Therefore, the next best thing might be to build machines that act and behave like humans. Actually, the Turing Test is based on this idea: Turing suggested that if we want to know whether we have succeeded in building an intelligent machine, we should place it at the other end of chat line. If we cannot distinguish the computer from a human, then we have succeeded at creating intelligence.

   However, with respect to the kind of Artificial Intelligence that is needed in industry, this approach isn't very useful. To illustrate the point, consider an analogy with aerodynamics: In aerodynamics we try to build planes that fly fast and efficiently, not planes that flap their wings like birds do, as the later approach has failed historically, e.g. Daedalus and Icarus.

3. *Artificial Intelligence* is the science of creating machines that think logically.

   The idea with this approach is to create machines that are based on mathematical logic. If a goal is given to these machines, then these machines use logical reasoning in order to deduce those actions that need to be performed in order to best achieve the given goals. Unfortunately, this approach had only limited success: In playing games the approach was quite successful for dealing with games like checkers or chess. However, the approach was mostly unsuccessful for dealing with many real world problems. There were two main reasons for its failure:

   (a) In order for the logical approach to be successful, the environment has to be *completely* described by mathematical axioms. It has turned out that our knowledge of the real world is often not sufficient to completely describe the environment via axioms.

(b) Even if we have complete knowledge, it often turns out that describing every possible case via logic formulae is just unwieldy. Consider the following formula:

$$\forall x : \big(\texttt{bird}(x) \to \texttt{flies}(x)\big)$$

The problem with this formula is that although it appears to be common sense, there are a number of counter examples:

i. Penguins, Emus, and ostriches don't fly.
However, if we put a penguin into a plane, it turns out the penguin will fly.
ii. Birds that are two young do not fly.
iii. Birds with clapped wings do not fly.
iv. $\cdots$

Trying to model all eventualities with logic has turned out to be too unwieldy to be practical.

(c) In real life situations we often deal with *uncertainty*. Classical logic does not perform well when it has to deal with uncertainties.

4. *Artificial Intelligence* is the science of creating machines that act rationally.

All we really want is to build machines that, given the knowledge we have, try to *optimize the expected results*: In our world, there is lots of uncertainty. We cannot hope to create machines that always make the decisions that turn out to be optimal. What we can hope is to create machines that will make decisions that turn out to be good on average. For example, suppose we try to create a program for asset management: We cannot hope to build a machine that always buys the best company share in the stock market. Rather, our goal should be to build a program that maximizes our expected profits in the long term.

It has turned out that the main tool needed for this approach is not mathematical logic but rather numerical analysis and mathematical statistics. The shift from logic to numerical analysis and statistics has been the most important reason for the success of Artificial Intelligence in the recent years. Another important factor is the *enhanced performance of modern hardware*.

Now that we have clarified the notion of artificial intelligence, we should set its goals. As we can never achieve more than what we aim for, we have every reason to be ambitious here. For example, my personal vision of Artificial Intelligence goes like this: Imagine 70 years from now you (not feeling too well) have a conversation with Siri. Instead of asking Siri for the best graveyard in the vicinity, you think about all the sins you have committed. As Siri has accompanied you for your whole life, she knows about these sins better than you. Hence, the conversation with Siri works out as follows:

You (with trembling voice):	Hey Siri, does God exist?
Siri (with the voice of Darth Vader): After a small pause which almost drains the battery of your phone completely, Siri gets back with a soothing announcement:	Your voice seems troubled, let me think $\cdots$ You don't have to worry any more, I have fixed the problem. He is dead now.

May The Force be with us on achieving our goals!

1.2 Literature

The main sources of these lecture notes are the following:

1. A course on artificial intelligence that was offered on the EDX platform. The course materials are available at

   http://ai.berkeley.edu/home.html.

2. The book

   Introduction to Artificial Intelligence

   written by Stuart Russel and Peter Norvig [5].

3. A course on artificial that is offered on Udacity. The title of the course is

   Intro to Artificial Intelligence

   and the course is given by Peter Norvig, who is director of research at Google and Sebastian Thrun, who is the chairman of Udacity.

The programs presented in these lecture notes have been tested with version 2.6 of SETLX.

Chapter 2

Search

In this chapter we discuss various *search algorithms*. First, we define the notion of a *search problem*. As one of the examples, we will discuss the sliding puzzle. Then 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,
5. A* search,
6. bidirectional A* search,
7. iterative deepening A* search, and
8. A*-IDA* search.

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

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

where

1. Q is the set of states, also known as the *state space*.
2. `nextStates` 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

   $$\texttt{nextStates} : Q \to 2^Q.$$

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

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

   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 `goal` there is a set of goal states G.

A *path* is a list $[s_1, \cdots, s_n]$ such that $\langle s_i, s_{i+1} \rangle \in R$ for all $i \in \{1, \cdots, n-1\}$. The *length* of this path is defined as the length of the list. A path $[s_1, \cdots, s_n]$ is a *solution* to the search problem P iff the following conditions are satisfied:

1. $s_1 =$ `start`, i.e. the first element of the path is the start state.
2. $s_n =$ `goal`, i.e. the last element of the path is the goal state.

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 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 problem: Three missionaries and three infidels have to cross a river that runs from the west to the east. Initially, they are on the northern shore. There is just one small boat and that boat has only room for 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.

```
1   problem := [m, i] |-> m > 0 && m < i;
2
3   noProblemAtAll := [m, i] |-> !problem(m, i) && !problem(3 - m, 3 - i);
4
5   nextStates := procedure(s) {
6       [m, i, b] := s;
7       if (b == 1) {  // The boat is on the northern shore.
8           return { [m - mb, i - ib, 0]
9                  : mb in {0 .. m}, ib in {0 .. i}
10                 | mb + ib in {1, 2} && noProblemAtAll(m - mb, i - ib)
11                 };
12      } else {
13          return { [m + mb, i + ib, 1]
14                 : mb in {0 .. 3 - m}, ib in {0 .. 3 - i}
15                 | mb + ib in {1, 2} && noProblemAtAll(m + mb, i + ib)
16                 };
17      }
18  };
19  start := [3, 3, 1];
20  goal  := [0, 0, 0];
```

Figure 2.1: The missionary and cannibals problem codes as a search problem.

Figure 2.1 shows a formalization of the missionaries and cannibals problem as a search problem. We discuss this formalization line be line.

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

   If m is the number of missionaries on a given shore, while i is the number of infidels on that same shore, then $problem(m, i)$ is `true` iff the missionaries have a problem on that shore.

2. Line 3 defines the auxiliary function `noProblemAtAll`.

   If m is the number of missionaries on the northern shore and i is the number of infidels on that shore, then the expression `noProblemAtAll`(m, i) is true, if there is no problem for the missionaries on either shore.

The implementation of this function uses the fact that if m is the number of missionaries on the northern shore, then $3 - m$ is the number of missionaries on the southern shore. Similarly, if i is the number of infidels on the northern shore, then the number of infidels on the southern shore is $3 - i$.

3. Line 5 to 18 define the function `nextStates`. A state s is represented as a triple of the form

   $$s = [m, i, b] \quad \text{where } m \in \{0, 1, 2, 3\},\ i \in \{0, 1, 2, 3\},\ b \in \{0, 1\}.$$

   Here m is the number of missionaries on the northern shore, i is the number of infidels on the northern shore, and b is the number of boats on the northern shore.

   (a) Line 6 extracts the components m, i, and b from the state s.

   (b) Line 7 checks whether the boat is on the northern 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 southern shore, `m` $-$ `mb` missionaries and `i` $-$ `ib` infidels remain on the northern shore. Of course, after the crossing the boat is no longer on the northern shore. Therefore, the new state has the form

   ```
   [m - mb, i - ib, 0].
   ```

   This explains line 8.

   (d) Since the number `mb` of missionaries leaving the northern shore can not be greater than the number m of all missionaries on the northern shore, we have the condition

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

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

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

   This explains line 9.

   (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

   $$\texttt{mb} + \texttt{im} \in \{1, 2\}.$$

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

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

   These two checks are performed in line 10.

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

   ```
   [m - mb, i - ib, 0].
   ```

   As the number of missionaries on the southern shore is $3 - \texttt{m}$ and the number of infidels on the southern shore is $3 - \texttt{i}$, `mb` is now a member of the set $\{0, \cdots, 3 - \texttt{m}\}$, while `ib` is a member of the set $\{0, \cdots, 3 - \texttt{i}\}$.

5. Finally the start state and the goal state are defined in line 19 and line 20.

The code in Figure 2.1 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] : m in {0..3}, i in {0..3}, b in {0,1} | noProblemAtAll(m, i) };
```

Figure 2.2 shows a graphical representation of the transition relation of the missionaries and cannibals puzzle. In that figure, for every state both the northern 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. ◇

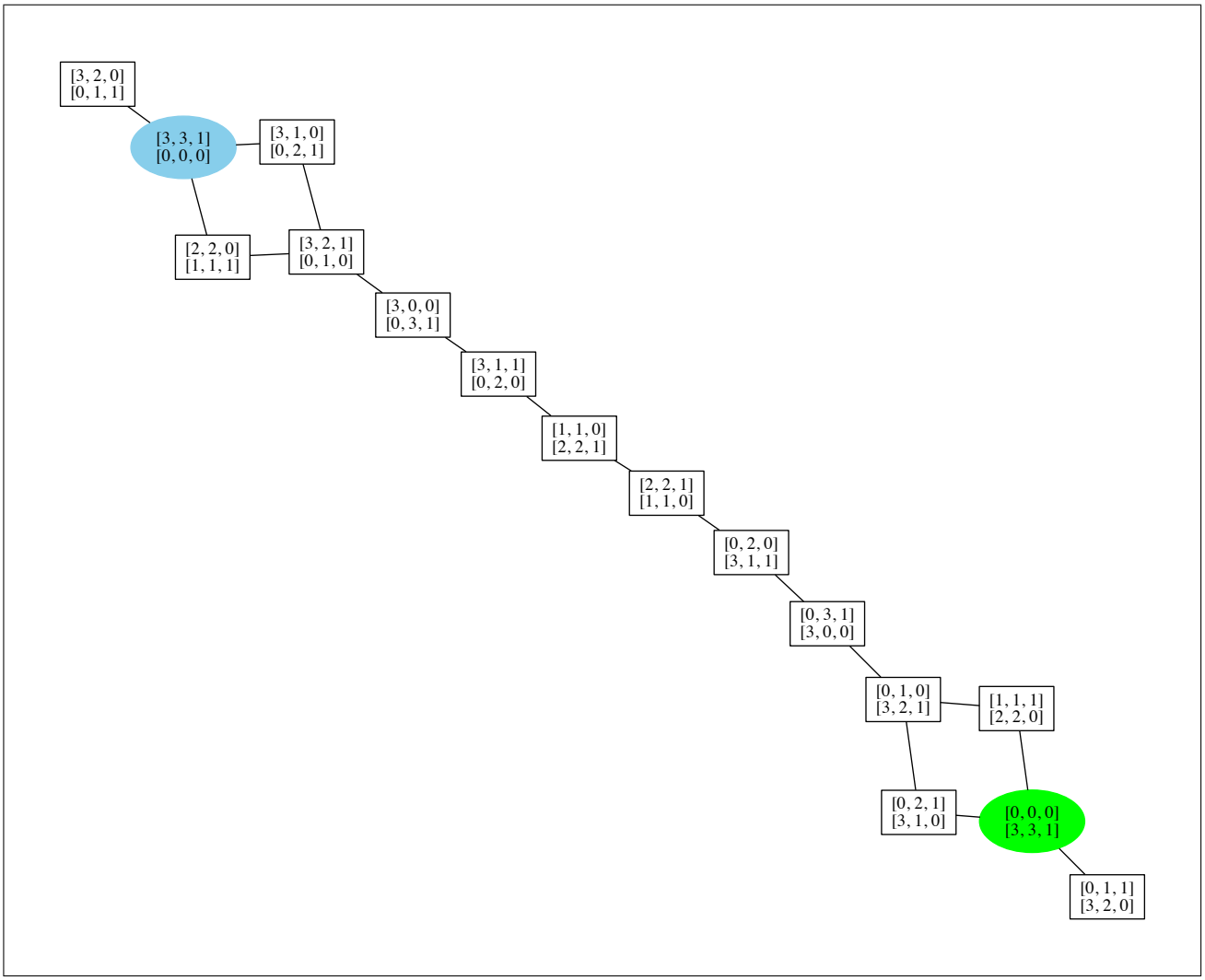

Figure 2.2: A graphical representation of the missionaries and cannibals problem.

2.1 The Sliding Puzzle

The 3×3 sliding puzzle uses a square board of length 3. This board is subdivided into $3 \times 3 = 9$ squares of length 1. Of these 9 squares, 8 are occupied with square tiles that are numbered from 1 to 8. One square remains empty. Figure 2.3 on page 2.3 shows two possible states of this sliding puzzle. The 4×4 sliding puzzle is similar to the 3×3 sliding puzzle but it is played on a square board of length 4 instead. The 4×4 sliding puzzle is also known as the *15 puzzle*, while the 3×3 puzzle is called the *8 puzzle*.

In order to solve the 3×3 sliding puzzle shown in Figure 2.3 we have to transform the state shown on the left of Figure 2.3 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.

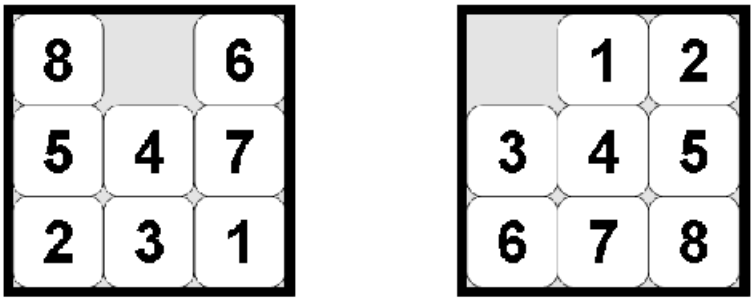

Figure 2.3: The 3×3 sliding puzzle.

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

http://mypuzzle.org/sliding.

The sliding puzzle is much more complex than the missionaries and cannibals problem 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 still be stored in 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 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 2.4 shows how the 3×3 sliding puzzle can be formulated as a search problem. We discuss this program line by line.

1. `findTile` is an auxiliary procedure that takes a `number` and a `state` and returns the row and column where the tile labeled with `number` can be found.

   Here, a state is represented as a list of lists. For example, the states shown in Figure 2.3 are represented as shown in line 26 and line 30. The empty tile is coded as 0.

2. `moveDir` 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 $\langle$`row` + `dx`, `col` + `dy`$\rangle$ is moved into the position $\langle$`row`, `col`$\rangle$, while the tile at position $\langle$`row` + `dx`, `col` + `dy`$\rangle$ becomes empty.

3. Given a `state`, the procedure `newStates` 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 [`row`, `col`] and the direction of the movement is given as [`dx`, `dy`], then in order to ensure that the empty tile can be moved to the position [`row` + `dx`, `col` + `dy`], we have to ensure that both

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

```
findTile := procedure(number, state) {
    n := #state;
    L := [1 .. n];
    for (row in L, col in L | state[row][col] == number) {
        return [row, col];
    }
};
moveDir := procedure(state, row, col, dx, dy) {
    state[row     ][col     ] := state[row + dx][col + dy];
    state[row + dx][col + dy] := 0;
    return state;
};
nextStates := procedure(state) {
    n          := #state;
    [row, col] := findTile(0, state);
    newStates  := [];
    directions := [ [1, 0], [-1, 0], [0, 1], [0, -1] ];
    L          := [1 .. n];
    for ([dx, dy] in directions) {
        if (row + dx in L && col + dy in L) {
            newStates += [ moveDir(state, row, col, dx, dy) ];
        }
    }
    return newStates;
};
start := [ [8, 0, 6],
           [5, 4, 7],
           [2, 3, 1]
         ];
goal := [ [0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]
        ];
```

Figure 2.4: The 3×3 sliding puzzle.

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 problems $\langle Q, \texttt{nextStates}, \texttt{start}, \texttt{goal} \rangle$, 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 step from a state in `Frontier`. Then, the states that had been previously present in `Frontier` are removed. These old states are then saved into a set `Visited`.

In order to avoid loops, an implementation of breadth first search keeps track of those states that have been visited. These states a collected in a 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 have a dictionary `Parent`. For every state s that is in `Frontier`, `Parent`$[s]$ is the state that caused s to be added to the set `Frontier`, i.e. we have

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

```
1  search := procedure(start, goal, nextStates) {
2      Frontier := { start };
3      Visited  := {}; // set of nodes that have been expanded
4      Parent   := {};
5      while (Frontier != {}) {
6          NewFrontier := {};
7          for (s in Frontier, ns in nextStates(s) | !(ns in Visited)) {
8              NewFrontier += { ns };
9              Parent[ns]  := s;
10             if (ns == goal) {
11                 return pathTo(goal, Parent);
12             }
13         }
14         Visited  += Frontier;
15         Frontier := NewFrontier;
16     }
17 };
```

Figure 2.5: Breadth first search.

Figure 2.5 on page 12 shows an implementation of breadth first search in SetlX. We discuss this implementation line by line:

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`.

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

3. `Parent` is a dictionary keeping track of the state leading to a given state.

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`. The procedure `pathTo`() is shown in Figure 2.6 on page 13.

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 `Frontier` is updated to `NewFrontier`.

The procedure call `pathTo`(`state`, `Parent`) constructs a path reaching from `start` to `state` in reverse by looking up the parent states.

If we try breadth first search to solve the missionaries and cannibals problem, we immediately get the solution shown in Figure 2.7. 15 nodes had to be expanded to find this solution. To keep this in perspective, we note that Figure 2.2 shows that the entire state space contains 16 states. Therefore, with the exception of one state, we have inspected all the states. This is a typical behaviour for breadth first search.

```
pathTo := procedure(state, Parent) {
    Path := [];
    while (state != om) {
        Path  += [state];
        state := Parent[state];
    }
    return reverse(Path);
};
```

Figure 2.6: The procedure `pathTo`().

```
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 2.7: A solution of the missionaries and cannibals problem.

Next, let us try to solve the 3×3 sliding puzzle. It takes less about 9 seconds to solve this problem on my computer[1], while 181439 states are touched. Again, we see that breadth first search touches nearly all the states reachable from the start state.

2.2.1 A Queue Based Implementation of Breadth First Search

In the literature, for example in Figure 3.11 of Russell & Norvig [5], breadth first search is often implemented using a queue data structure. Figure 2.8 on page 14 shows an implementation of breadth first search that uses a queue to store the set `Frontier`. However, when we run this version, it turns out that the solution of the 3×3 sliding puzzle needs about 58 seconds, which is a lot slower than our set based implementation that has been presented in Figure 2.5.

[1] I happen to own an iMac from 2011. This iMac is equipped with 16 Gigabytes of main memory and a quad core 2.7 GHz "Intel Core i5" processor. I suspect this to be the I5-2500S (Sandy Bridge) processor.

```
search := procedure(start, goal, nextStates) {
    Queue   := [ start ];
    Visited := {};
    Parent  := {};
    while (Queue != []) {
        state := Queue[1];
        Queue := Queue[2..];
        if (state == goal) {
            return pathTo(state, Parent);
        }
        Visited   += { state };
        newStates := nextStates(state);
        for (ns in newStates | !(ns in Visited) && Parent[ns] == om) {
            Parent[ns] := state;
            Queue      += [ ns ];
        }
    }
};
```

Figure 2.8: A queue based implementation of breadth first search.

The solution of the 3×3 sliding puzzle that is found by breadth first search is shown in Figure 2.9 and Figure 2.10.

We conclude our discussion of breadth first search by noting the two most important properties of breadth first search.

1. Breadth first search is *complete*: If there is a solution to the given search problem, then breadth first search is going to find it.

2. The solution found by breadth first search is *optimal*, i.e. it is the shortest possible solution.

Proof: Both of these claims can be shown simultaneously. Consider the implementation of breadth first search shown in Figure 2.5. An easy induction on the number of iterations of the `while` loop shows 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`. This claim is obviously true before the first iteration of the while loop as in this case, `Frontier` only contains the state `start`. 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 these states. There is a shorter path to these states if these states are already a member of the set `Visited`. Hence, the claim is true after $n + 1$ iterations also.

Now, if there is a path form `start` to `goal`, there must also be a shortest path. Assume this path has a length of k. Then, `goal` is reached in the iteration number k 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 not possible. For example, it is impossible to solve the more interesting cases of the 4×4 sliding puzzle.

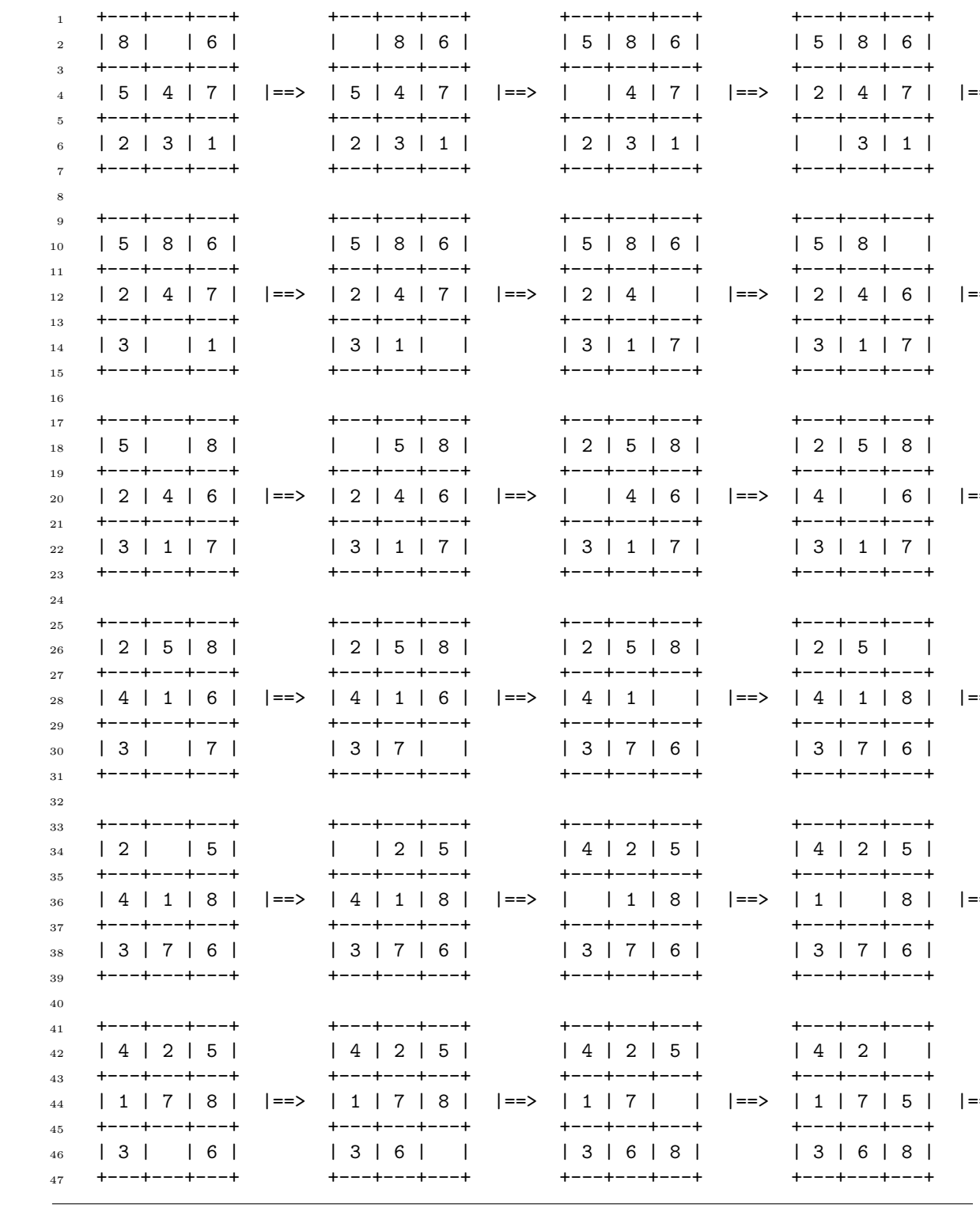

Figure 2.9: The first 24 steps in the solution of the 3×3 sliding puzzle.

```
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 4 |   | 2 |       |   | 4 | 2 |       | 1 | 4 | 2 |       | 1 | 4 | 2 |
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 1 | 7 | 5 | |==>  | 1 | 7 | 5 | |==>  |   | 7 | 5 | |==>  | 3 | 7 | 5 | |==>
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 3 | 6 | 8 |       | 3 | 6 | 8 |       | 3 | 6 | 8 |       |   | 6 | 8 |
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+

+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 1 | 4 | 2 |       | 1 | 4 | 2 |       | 1 |   | 2 |       |   | 1 | 2 |
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 3 | 7 | 5 | |==>  | 3 |   | 5 | |==>  | 3 | 4 | 5 | |==>  | 3 | 4 | 5 |
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
| 6 |   | 8 |       | 6 | 7 | 8 |       | 6 | 7 | 8 |       | 6 | 7 | 8 |
+---+---+---+       +---+---+---+       +---+---+---+       +---+---+---+
```

Figure 2.10: The last 7 steps in the solution of the 3×3 sliding puzzle.

2.3 Depth First Search

To overcome the memory limitations of breadth first search, the depth first search algorithm has been developed. The basic idea is to replace the queue of Figure 2.8 by a stack. The resulting algorithm is shown in Figure 2.11 on page 16. 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.

```
search := procedure(start, goal, nextStates) {
    Stack  := [ start ];
    Parent := {};
    while (Stack != []) {
        state := Stack[-1];
        Stack := Stack[..-2];
        if (state == goal) {
            return pathTo(state, Parent);
        }
        newStates := nextStates(state);
        for (ns in newStates | ns != start && Parent[ns] == om) {
            Parent[ns] := state;
            Stack      += [ns];
        }
    }
};
```

Figure 2.11: The depth first search algorithm.

Actually, it is not necessary to understand the details of the implementation shown in Figure 2.11 on page 16. The reason is that the recursive implementation of depth first search that is presented in the following subsection is superior to the implementation shown in Figure 2.11 on page 16. When we test the implementation shown above with the 3×3 sliding puzzle, it takes about 96 seconds to find a solution. The solution that is found has a length of $41,553$ steps. As the shortest path from `start` to `goal` has 31 steps, the solution found by depth first search is very far from optimal. All this is rather disappointing news. The only good news is that there is no longer a need to keep the set `Visited` around. However, we still have to maintain the set `Parent`. If we

were more ambitious, we could eliminate the use of this dictionary also, but the resulting implementation would be rather unwieldy. Fortunately, we will be able to get rid of the set `Parent` with next to no effort when we develop a recursive implementation of depth first search in the following subsection.

2.3.1 Getting Rid of the `Parent` Dictionary

It can be argued that the implementation of depth first search discussed previously is not really depth first search because it uses the dictionary `Parent`. As states are only added to `Parent` and never removed, at the end of the search this dictionary will contain all states that have been visited. This defeats the most important advantage of depth first search which is the fact that it should only store the current path that is investigated. Therefore, it has been suggested (for example compare Russel and Norvig [5]) that instead of storing single states, the stack should store the full paths leading to these states. This leads to the implementation shown in Figure 2.12 on page 17.

```
1  search := procedure(start, goal, nextStates) {
2      Stack  := [ [start] ];
3      while (Stack != []) {
4          Path  := Stack[-1];
5          Stack := Stack[..-2];
6          state := Path[-1];
7          if (state == goal) {
8              return Path;
9          }
10         newStates := nextStates(state);
11         for (ns in newStates | !(ns in Path)) {
12             Stack += [ Path + [ns] ];
13         }
14     }
15 };
```

Figure 2.12: An path-based implementation of depth first search.

Unfortunately, it turns out that the paths get very long and hence need a lot of memory to be stored and this fact defeats the main idea of this implementation. As a result, the procedure `search` that is given in Figure 2.12 on page 17 is not able to solve the instance of the 3×3 sliding puzzle that was shown in Figure 2.3 on page 10.

Exercise 1: Assume the set of states Q is defined as

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

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

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

Next, the function `nextStates` is defined as

$$\texttt{nextStates}\big(\langle a, b\rangle\big) := \big\{\langle a+1, b\rangle, \langle a, b+1\rangle\big\}.$$

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

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

Assume that states can be stored using 8 bytes. Furthermore, assume that we use the algorithm given in Figure 2.12 on page 17 to solve the search problem $\mathcal{P}$. How many bytes do we need to store all the states on the stack in the moment that the `goal` is reached? How big is this number if $n = 10,000$?

(a) **Note** that the question only asks for the memory needed to store the states. The memory needed to store the stack itself and the various lists on the stack comes on top of the memory needed to store the states. However, to answer this exercise correctly, you should ignore this type of memory.

(b) In order to understand how the stack evolves, we need to know that in SetlX sets are ordered ascendigly. Furthermore, pairs are ordered lexicographically in SetlX, i.e. we have

$$\langle x_1, y_1\rangle < \langle x_2, y_2\rangle \quad \Longleftrightarrow \quad x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2).$$

Hence, when we have a state $\langle a, b\rangle$, the set `nextStates`$\big(\langle a, b\rangle\big)$ is ordered as follows:

$$\big\{\langle a, b+1\rangle\big\}, \langle a+1, b\rangle\big\}. \qquad \diamond$$

2.3.2 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 more easy 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 2.13 on page 18.

```
1  search := procedure(start, goal, nextStates) {
2      return dfs(start, goal, nextStates, [start]);
3  };
4  dfs := procedure(state, goal, nextStates, Path) {
5      if (state == goal) {
6          return Path;
7      }
8      newStates := nextStates(state);
9      for (ns in newStates | !(ns in Path)) {
10         result     := dfs(ns, goal, nextStates, Path + [ns]);
11         if (result != om) {
12             return result;
13         }
14     }
15 };
```

Figure 2.13: A recursive implementation of depth first search.

The only purpose of the procedure `search` is to call the procedure `dfs`, which needs one additional argument. This argument is called `Path`. The idea is that `Path` is a path leading from the state `start` to the current `state` that is the first argument of the procedure `dfs`. Of course, on the first 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, `newStates` 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 procedure `dfs` recursively. This way, we retain the invariant of `dfs` that the list `Path` is a path connecting `start` with `state`.

3. 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 list `Path`. In the non-recursive version of depth

first search, we had used the set `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.

4. 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 the undefined value `om` is returned, the `for` loop needs to carry on and test the other successors of `state`.

5. Note that the recursive invocation of `dfs` returns `om` if the end of the `for` loop is reached and no solution has been returned so far. The reason is that there is no `return` statement at the end of the procedure `dfs`. Hence, if the last line of the procedure `dfs` is reached, `om` is returned by default.

For the 3×3 puzzle, it takes about 2 seconds to compute the solution. In this case, the length of the solution is still 3653 steps, which is unsatisfying. The good news is that this program does not need much memory. The only variable that uses considerable memory is the variable `Path`. If we can somehow keep the list `Path` short, then the recursive version of depth first search uses only a tiny fraction of the memory needed by breadth first search.

2.4 Iterative Deepening

The fact that the recursive version of depth first search took just 2 seconds to find a solution is very impressive. The questions 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 2.14 on page 19 implements this strategy. We proceed to discuss the details of this program.

```
1  search := procedure(start, goal, nextStates) {
2      limit := 1;
3      while (true) {
4          Path := depthLimitedSearch(start, goal, nextStates, limit);
5          if (Path != om) { return Path; }
6          limit += 1;
7      }
8  };
9  depthLimitedSearch := procedure(start, goal, nextStates, limit) {
10     Stack := [ [start] ];
11     while (Stack != []) {
12         Path  := Stack[-1];
13         Stack := Stack[..-2];
14         state := Path[-1];
15         if (state == goal)  { return Path; }
16         if (#Path >= limit) { continue;    }
17         for (ns in nextStates(state) | !(ns in Path)) {
18             Stack += [ Path + [ns] ];
19         }
20     }
21 };
```

Figure 2.14: Iterative deepening implemented in SetlX.

1. The procedure `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 procedure `depthLimitedSearch` implements depth first search but takes care to compute only those paths that have a length of at most `limit`. The implementation shown in Figure 2.14 is stack based. In this implementation, the stack contains paths leading from `start` to the state at the end of a given path. Hence it is similar to the implementation of depth first search shown in Figure 2.11 on page 16.
3. The stack is initialized to contain the path [`start`].
4. In the `while`-loop, the first thing that happens is that the `Path` on top of the stack is removed from the stack. The state at the end of this `Path` is called `state`. If this `state` happens to be the `goal`, a solution to the search problem has been found and this solution is returned.
5. Otherwise, we check the length of `Path`. If this length is greater than or equal to the `limit`, the `Path` can be discarded as we have already checked that it does not end in the `goal`.
6. Otherwise, the neighbours of `state` are computed. For every neighbour `ns` of `state` that has not yet been encountered in `Path`, we extend `Path` to a new list that ends in `ns`.
7. This process is iterated until the `Stack` is exhausted.

The nice thing about the program presented in this section is the fact that it does not use much memory. The reason is that the stack can never have a size that is longer than `limit` and therefore the overall memory that is needed can be bounded by $\mathcal{O}(\texttt{limit}^2)$. However, when we run this program to solve the 3×3 sliding puzzle, the algorithm takes about 42 minutes. There are two reasons for this:

1. First, it is quite wasteful to run the search for a depth limit of 1, 2, 3, $\cdots$ all the way up to 31. Essentially, all the computations done with a limit less than 31 are essentially wasted.
2. Given a state s that is reachable from the `start`, there often is 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.

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

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

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 `nextStates` is defined as

$$\texttt{nextStates}\big(\langle a, b\rangle\big) := \big\{\langle a+1, b\rangle, \langle a, b+1\rangle\big\}.$$

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

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

Given a natural number n, compute the number of different solutions of this search problem and prove your claim. ◇

Exercise 3: If there is no solution, the implementation of iterative deepening that is shown in Figure 2.14 does not terminate. The reason is that the function `depthLimitedSearch` does not distinguish between the following two reasons for failure:

1. It can fail to find a solution because the depth limit is reached.
2. It can also fail it has tried all paths without hitting the depth limit but the `Stack` is exhausted.

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

2.4.1 A Recursive Implementation of Iterative Deepening

If we implement iterative deepening recursively, then we know that the call stack is bounded by the length of the shortest solution. Figure 2.15 on page 21 shows a recursive implementation of iterative deepening. This implementation has several nice features:

```
search := procedure(start, goal, nextStates) {
    limit := 1;
    while (true) {
        result := dfsLimited(start, goal, nextStates, [start], limit);
        if (result != om) {
            return result;
        }
        limit += 1;
    }
};
dfsLimited := procedure(state, goal, nextStates, Path, limit) {
    if (state == goal) {
        return Path;
    }
    if (limit == 0) {
        return;  // limit execceded
    }
    for (ns in nextStates(state) | !(ns in Path)) {
        result := dfsLimited(ns, goal, nextStates, Path + [ns], limit - 1);
        if (result != om) {
            return result;
        }
    }
};
```

Figure 2.15: A recursive implementation of iterative deepening.

1. The path that is computed no longer requires the dictionary `Parent` as it is built incrementally in the argument `Path` of the procedure `dfsLimited`.
2. Similarly, there is no longer a need to keep the dictionary `Distance`.

Unfortunately, the running time of the recursive implementation of iterative deepening is still quite big: On my computer, the recursive implementation takes about 36 minutes.

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 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

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

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. On simple idea is to start searching both from the node `start` and the node `goal` simultaneously. The justification is that we can hope that the path starting form `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

$$2 \cdot \frac{b^{d/2} - 1}{b - 1}$$

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 40. Then we need to explore

$$2^{40} - 1 = 1,099,511,627,775$$

in breadth first search, while we only have to explore

$$2^{40/2} - 1 = 1,048,575$$

with bidirectional depth first search. While it is certainly feasible to keep a million states in memory, keeping a trillion states in memory is impossible on most devices.

Figure 2.16 on page 23 shows the implementation of bidirectional breadth first search. Essentially, we have to copy the breadth first program shown in Figure 2.5. 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. `VisitedA` is the set of states that have been visited starting from `start`, while `VisitedB` is the set of states that have been visited starting from `goal`.

3. 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`.

4. 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.

5. Initially, the `while` loop adds the frontier sets to the visited sets as all the neighbours of the frontier sets will now be explored.

6. Then the `while` loop computes those states that can be reached from `FrontierA` and have not been visited from `start`. If a state `ns` is a neighbour of a state `s` from the set `FrontierA` and the state `ns` has already been encountered during the search that started from `goal`, then a path leading from `start` to `goal` has been found and this path is returned. The function `combinePaths` that computes this path by combining the path that leads from `start` to `ns` and then from `ns` to `goal` to is shown in Figure 2.17 on page 23.

7. Next, the same computation is done with the role of the states `start` and `goal` exchanged.

On my computer, bidirectional breadth first search solves the 3×3 sliding puzzle in less than a second! However, bidirectional breadth first search is still not able to solve the 4×4 sliding puzzle since the portion of the search space that needs to be computed is just too big to fit into memory.

```
search := procedure(start, goal, nextStates) {
    FrontierA := { start };
    VisitedA  := {}; // set of nodes expanded starting from start
    ParentA   := {};
    FrontierB := { goal };
    VisitedB  := {}; // set of nodes expanded starting from goal
    ParentB   := {};
    while (FrontierA != {} && FrontierB != {}) {
        VisitedA += FrontierA;
        VisitedB += FrontierB;
        NewFrontier := {};
        for (s in FrontierA, ns in nextStates(s) | !(ns in VisitedA)) {
            NewFrontier += { ns };
            ParentA[ns] := s;
            if (ns in VisitedB) {
                return combinePaths(ns, ParentA, ParentB);
            }
        }
        FrontierA   := NewFrontier;
        NewFrontier := {};
        for (s in FrontierB, ns in nextStates(s) | !(ns in VisitedB)) {
            NewFrontier += { ns };
            ParentB[ns] := s;
            if (ns in VisitedA) {
                return combinePaths(ns, ParentA, ParentB);
            }
        }
        FrontierB := NewFrontier;
    }
};
```

Figure 2.16: Bidirectional breadth first search.

```
combinePaths := procedure(node, ParentA, ParentB) {
    Path1 := pathTo(node, ParentA);
    Path2 := pathTo(node, ParentB);
    return Path1[..-2] + reverse(Path2);
};
```

Figure 2.17: Combining two paths.

2.6 Best First Search

Up to now, all the search algorithms we have discussed were 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. If a human tries to solve a problem, she usually will develop a feeling 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 *heuristic*.

Definition 2 (Heuristic) Given a search problem

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

a *heuristic* is a function

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

that computes an approximation of the distance of a given state s to the goal state `goal`. The heuristic is *admissible* if it always underestimates the true distance, i.e. if the function

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

computes the true distance of 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*: An admissible heuristic must never overestimate the distance to the goal, but it is free to underestimate this distance.

Finally, the heuristic h is called *consistent* iff we have

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

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

$$s_2 \in \texttt{nextStates}(s_1).$$

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 cheaper path from s_1 leading to the `goal` than the one that visits s_2 first. Hence we have the inequality

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

Theorem 3 Every consistent heuristic is also admissible.

Proof: Assume that the heuristic h is consistent. Assume further that $s \in Q$ is some state such that there is a 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 p is n and we have to show that $h(s) \leq n$. In order to prove this claim, 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:

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

This concludes the proof. □

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 this heuristic is too trivial to be of any use.

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 2.3 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 state can differ by at most one.

   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

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

   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 is $\langle 3, 3\rangle$, 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 position $\texttt{pos}(t, s)$ for all tiles $t \in \{1, \cdots, 8\}$ in a given state s as follows:

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

   i.e. given a state s, the expression $\texttt{pos}(t, s)$ computes the Cartesian coordinates of the tile t with respect to 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 measure the number of moves that would be needed if we wanted to put every tile of s into its final positions and if we were allowed to slide tiles over each other. Figure 2.18 on page 26 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.

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

   (b) Next, the `for` loop 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 `findTile`.

   (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 distance is admissible. The reason is that if $s_2 \in \texttt{nextStates}(s_1)$, 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

```
manhattan := procedure(stateA, stateB) {
    n := #stateA;
    L := [1 .. n];
    result := 0;
    for (rowA in L, colA in L | stateA[rowA][colA] != 0) {
        [rowB, colB] := findTile(stateA[rowA][colA], stateB);
        result += abs(rowA - rowB) + abs(colA - colB);
    }
    return result;
};
```

Figure 2.18: The Manhattan distance between two states.

position differs by either one row or one column. Therefore,

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

and hence $h(s_1) \leq 1 + h(s_2)$. □

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.

```
bestFirstSearch := procedure(start, goal, nextStates, heuristic) {
    PrioQueue := { [0, [start]] };
    while (PrioQueue != {}) {
        [_, Path] := fromB(PrioQueue);
        state     := Path[-1];
        if (state == goal) { return Path; }
        newStates := nextStates(state);
        for (ns in newStates | !(ns in Path)) {
            PrioQueue += { [heuristic(ns, goal), Path + [ns]] };
        }
    }
};
```

Figure 2.19: The best first search algorithm.

The procedure `bestFirstSearch` shown in Figure 2.19 on page 26 takes four parameters. The first three of these parameters are the same as in the previous search algorithm. The last parameter `heuristic` is a function that takes to states and then estimates the distance between these states. Later, we will use the Manhattan distance to serve as this `heuristic`. The details of the implementation are as follows:

1. The variable `PrioQueue` serves as a priority queue. We take advantage of the fact that SetlX stores sets as ordered binary trees that store their elements in increasing order. Hence, the smallest element of a set is the first element.

   Furthermore, SetlX orders pairs lexicographically. Hence, we store the paths in `PrioQueue` as pairs of the form

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

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

2. As long as `PrioQueue` is not empty, we take the `Path` from the beginning of this priority queue and remove it from the queue. The state at the end of `Path` is named `state`.

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

4. Otherwise, the states reachable from `state` are inserted into the priority queue. When these states are inserted, we have to compute their estimated distance to `goal` since this distance is used as the priority in `PrioQueue`.

Best first search solves the instance of the 3×3 puzzle shown in Figure 2.3 on page 10 in less than half a second. However, the solution that is found takes 75 steps. While this is not as ridiculous as the solution found by depth first search, it is still far from an optimal solution. Furthermore, best first search is still not strong enough to solve the 4×4 puzzle shown in Figure 2.22 on page 31.

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 The A* Search Algorithm

We have seen that best first search can be very fast. However, the solution returned by best first search is not optimal. In contrast, the A* algorithm described next guarantees that a shortest path is found provided the heuristic used is consistent. The basic idea is that the A* search algorithm works similar to the queue based version of breadth first search, but instead of using a simple queue, a priority queue is used instead. The priority $f(s)$ of every state s is given as

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

where $g(s)$ computes the length of the path leading from `start` to s and $h(s)$ is the heuristical estimate of the distance from s to `goal`. The details of the A* algorithm are given in Figure 2.20 on page 28 and discussed below.

The function `aStarSearch` takes 4 parameters:

1. `start` is a state. This state represents the start state of the search problem.

2. `goal` is the goal state.

3. `nextStates` is a function that takes a state as a parameter. For a state s,

   $$\texttt{nextStates}(s)$$

   computes the set of all those states that can be reached from s in a single step.

4. `heuristic` is a function that takes two parameters. For two states s_1 and s_2, the expression

   $$\texttt{heuristic}(s_1, s_2)$$

   computes an estimate of the distance between s_1 and s_2.

The function `aStarSearch` maintains 5 variables that are crucial for the understanding of the algorithm.

1. `Parent` is a dictionary associating a parent state with those states that have already been encountered during the search, i.e. we have

   $$\texttt{Parent}[s_2] = s_1 \;\Rightarrow\; s_2 \in \texttt{nextStates}(s_1).$$

   Once the goal has been found, this dictionary is used to compute the path from `start` to `goal`.

```
aStarSearch := procedure(start, goal, nextStates, heuristic) {
    Parent   := {};                    // back pointers, represented as dictionary
    Distance := { [start, 0] };
    estGoal  := heuristic(start, goal);
    Estimate := { [start, estGoal] };  // estimated distances
    Frontier := { [estGoal, start] };  // priority queue
    while (Frontier != {}) {
        [stateEstimate, state] := fromB(Frontier);
        if (state == goal) {
            return pathTo(state, Parent);
        }
        stateDist := Distance[state];
        for (neighbour in nextStates(state)) {
            oldEstimate := Estimate[neighbour];
            newEstimate := stateDist + 1 + heuristic(neighbour, goal);
            if (oldEstimate == om || newEstimate < oldEstimate) {
                Parent[neighbour]   := state;
                Distance[neighbour] := stateDist + 1;
                Estimate[neighbour] := newEstimate;
                Frontier            += { [newEstimate, neighbour] };
                if (oldEstimate != om) {
                    Frontier -= { [oldEstimate, neighbour] };
                }
            }
        }
    }
};
```

Figure 2.20: The A* search algorithm.

2. `Distance` is a dictionary. For every state s that is encountered during the search, this dictionary records the length of the shortest path from `start` to s.

3. `Estimate` is a dictionary. For every state s encountered in the search, `Estimate`$[s]$ is an estimate of the length that a path from `start` to `goal` would have if it would pass through the state s. This estimate is calculated using the equation

   $$\texttt{Estimate}[s] = \texttt{Distance}[s] + \texttt{heuristic}(s, \texttt{goal}).$$

   Instead of recalculating this sum every time we need it, we store it in the dictionary `Estimate`. This increases the efficiency of the algorithm.

4. `Frontier` is a priority queue. The elements of `Frontier` are pairs of the form

   $$[d, s] \quad \text{such that} \quad d = \texttt{Estimate}[s],$$

   i.e. if $[d, s] \in$ `Frontier`, then the state s has been encountered in the search and it is estimated that a path leading from `start` to `goal` and passing through s would have a length of d.

Now that we have established the key variables, the A* algorithm runs in a `while` loop that does only terminate if either a solution is found or the priority queue `Frontier` is exhausted.

1. First, the `state` with the smallest estimated distance for a path running from `start` to `goal` and passing through `state` is chosen from the priority queue `Frontier`. Note that the call to `fromB` does not only return the pair

[stateEstimate, state]

from Frontier that has the lowest value of stateEstimate, but also removes this pair from the priority queue.

2. Now if this state is the goal a solution has been found. Hence, in this case the solution is returned and the function aStarSearch terminates.

3. Otherwise, we check the length of the path leading from start to state. This length is stored in stateDist. Effectively, this is the distance between start and state.

4. Next, we have a loop that iterates over all neighbours of state.

   (a) For every neighbour we check the estimated length of a solution passing through neighbour and store this length in oldEstimate. Note that oldEstimate is undefined, i.e. it has the value om, if we haven't yet encountered the node neighbour in our search.

   (b) If a solution would go from start to state and from there proceed to neighbour, the estimated length of this solution would be

   $$\texttt{stateDist} + 1 + \texttt{heuristic}(\texttt{neighbour}, \texttt{goal}).$$

   Therefore this value is stored in newEstimate.

   (c) Next, we need to check whether this new solution that first passes through state and then proceeds to neighbour is better than the previous solution that passes through neighbour. This check is done by comparing newEstimate and oldEstimate. Note that we have to take care of the fact that there might be no valid oldEstimate.

   In case the new solution seems to be better than the old solution, we have to update the Parent dictionary, the Distance dictionary, and the Estimate dictionary. Furthermore, we have to update the priority queue Frontier. Here, we have to take care to remove the previous entry for the state neighbour if it exists, which is the case if oldEstimate is not om.

It can be shown that the A* search algorithm is complete and that the computed solution is optimal. The A* algorithm has been discovered by Hart, Nilsson, and Raphael and was first published in 1968 [2]. However, there was subtle bug in the first publication which was corrected in 1972 [3].

When we run A* on the 3×3 sliding puzzle, it takes about 17 seconds to solve the instance shown in Figure 2.3 on page 10. If we just look at the time, this seems to be disappointing. However, the good news is that now only $10,061$ states are touched in the search for a solution. This is more than a tenfold reduction when compared with breadth first search. The fact that the running time is, nevertheless, quite high results from the complexity of computing the Manhattan distance.

2.8 Bidirectional A* Search

So far, the best search algorithm we have encountered is bidirectional breadth first search. However, in terms of memory consumption, the A* algorithm also looks very promising. Hence, it might be a good idea to combine these two algorithms. Figure 2.21 on page 30 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.20 on page 28 as the algorithm for bidirectional search shown in Figure 2.16 on page 23 relates to breadth first search shown in Figure 2.5 on page 12. Hence, we will not discuss the details any further.

When we run bidirectional A* search for the 3×3 sliding puzzle shown in Figure 2.3 on page 10, the program takes 2 second but only uses $2,963$ states. Therefore, I have tried to solve the 4×4 sliding puzzle shown in Figure 2.22 on page 31 using bidirectional A* search. A solution of 44 steps was found in less than 58 seconds. Only $20,624$ states had to be processed to compute this solution! None of the other algorithms presented so far was able to compute the solution.

```
aStarSearch := procedure(start, goal, nextStates, heuristic) {
    ParentA    := {};                    ParentB    := {};
    DistanceA  := { [start, 0] };        DistanceB  := { [goal,  0] };
    estimate   := heuristic(start, goal);
    EstimateA  := { [start, estimate] }; EstimateB  := { [goal,  estimate] };
    FrontierA  := { [estimate, start] }; FrontierB  := { [estimate, goal ] };
    while (FrontierA != {} && FrontierB != {}) {
        [guessA, stateA] := first(FrontierA);
        stateADist       := DistanceA[stateA];
        [guessB, stateB] := first(FrontierB);
        stateBDist       := DistanceB[stateB];
        if (guessA <= guessB) {
            FrontierA -= { [guessA, stateA] };
            for (neighbour in nextStates(stateA)) {
                oldEstimate := EstimateA[neighbour];
                newEstimate := stateADist + 1 + heuristic(neighbour, goal);
                if (oldEstimate == om || newEstimate < oldEstimate) {
                    ParentA[neighbour]   := stateA;
                    DistanceA[neighbour] := stateADist + 1;
                    EstimateA[neighbour] := newEstimate;
                    FrontierA            += { [newEstimate, neighbour] };
                    if (oldEstimate != om) { FrontierA -= { [oldEstimate, neighbour] }; }
                }
                if (DistanceB[neighbour] != om) {
                    return combinePaths(neighbour, ParentA, ParentB);
                }
            }
        } else {
            FrontierB -= { [guessB, stateB] };
            for (neighbour in nextStates(stateB)) {
                oldEstimate := EstimateB[neighbour];
                newEstimate := stateBDist + 1 + heuristic(start, neighbour);
                if (oldEstimate == om || newEstimate < oldEstimate) {
                    ParentB[neighbour]   := stateB;
                    DistanceB[neighbour] := stateBDist + 1;
                    EstimateB[neighbour] := newEstimate;
                    FrontierB            += { [newEstimate, neighbour] };
                    if (oldEstimate != om) { FrontierB -= { [oldEstimate, neighbour] }; }
                }
                if (DistanceA[neighbour] != om) {
                    return combinePaths(neighbour, ParentA, ParentB);
                }
            }
        }
    }
};
```

Figure 2.21: Bidirectional A* search.

```
start := [ [  1, 2,  0,  4 ],
           [ 14, 7, 12, 10 ],
           [  3, 5,  6, 13 ],
           [ 15, 9,  8, 11 ]
         ];
goal  := [ [  1,  2,  3,  4 ],
           [  5,  6,  7,  8 ],
           [  9, 10, 11, 12 ],
           [ 13, 14, 15,  0 ]
         ];
```

Figure 2.22: 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*. It has been invented by Richard Korf [4]. Figure 2.23 on page 32 shows an implementation of IDA* in SetlX. We proceed to discuss this program.

1. As in the A* search algorithm, the function `idaStarSearch` 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) `nextStates` 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
   $$\texttt{heuristic}(s_1, s_2)$$
   computes an estimate of the distance between s_1 and s_2. It is assumed that this estimate is optimistic.
2. The function `idaStarSearch` 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 loop. In this loop, we call the function `search` to compute a path from `start` to `goal` that has a length of at most `limit`. This function `search` uses A* search and is described in detail below. Now there are two cases:
   (a) `search` does find a path. In this case, this path is returned in the variable `Path` and this variable is a list. This list is returned as the solution to the search problem.
   (b) `search` is not able to find a path within the given `limit`. In this case, search will not return a path but instead it will return a number. This number 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 `search`.
   Note that the fact that `search` is able to compute this new `limit` is a significant enhancement of 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 `search`. This function takes 7 parameters, which we describe next.

```
idaStarSearch := procedure(start, goal, nextStates, heuristic) {
    limit := heuristic(start, goal);
    while (true) {
        Path := search(start, goal, nextStates, 0, limit, [start], heuristic);
        if (isList(Path)) {
            return Path;
        }
        limit := Path;
    }
};
search := procedure(state, goal, nextStates, distance, limit, Path, heuristic) {
    total  := distance + heuristic(state, goal);
    if (total > limit) {
        return total;
    }
    if (state == goal) {
        return Path;
    }
    smallest := mathConst("Infinity");
    for (ns in nextStates(state) | !(ns in Path) ) {
        result := search(ns, goal, nextStates, distance + 1, limit,
                         Path + [ ns ], heuristic);
        if (isList(result)) {
            return result;
        }
        smallest := min([result, smallest]);
    }
    return smallest;
};
```

Figure 2.23: Iterative deepening A^* search.

1. `state` is a state. Initially, `state` is the `start` state. However, on recursive invocations of `search`, `state` is some state such that we have already found a path from `start` to `state`.

2. `goal` is another state. The purpose of the recursive invocations of `search` is to find a path from `state` to `goal`.

3. `nextStates` 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. `distance` is the distance between `start` and `state`. It is also the length of the list `Path` described below.

5. `limit` is the maximal length of the path from `start` to `goal`.

6. `Path` is a path from `start` to `state`.

7. `heuristic`(s_1, s_2) computes an *estimate* of the distance between 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 or equal than the true distance between s_1 and s_2.

We proceed to describe the implementation of the function `search`.

1. As `distance` is the length of `Path` and the heuristic is assumed to be optimistic, i.e. it always underestimates the true distance, 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

`distance` + `heuristic`(`state`, `goal`).

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

2. 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`.

3. If we are lucky and have found the `goal`, the `Path` is returned.

4. Otherwise, we iterate over all nodes 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 is

   `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 `ns`. If this distance is smaller than the distance returned from previous nodes which is stored in the variable `smallest`, this variable is updated accordingly. This way, if the `for` loop is not able to return a path leading to `goal`, the variable `smallest` contains the minimum 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 that path that is shortest. However, this is not necessary: Every time the function `search` is invoked it is already guaranteed that there is no path that is shorter than the parameter `limit`. Therefore, if `search` is able to find a path that has a length of at most `limit`, this path is already know 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 2.3 on page 10, this algorithm takes about 2.6 seconds to solve the puzzle. For the 4×4 sliding puzzle, the algorithm takes about 518 seconds. 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.

2.10 The A*-IDA* Search Algorithm

So far, from all of the algorithms we have tried, the bidirectional A* search has performed best. However, bidirectional A* search is only feasible if sufficient memory is available. While IDA* requires more time, its memory consumption is much lower than the memory consumption of bidirectional A*. Hence, it is natural to try to combine the A* algorithm and the IDA* algorithm. Concretely, the idea is to run an A* search from the `start` node until memory is more or less exhausted. Then, we start IDA* from the `goal` node and search until we find any of the nodes discovered by the A* search that had been started from the `start` node.

An implementation of the A*-IDA* algorithm is shown in Figure 2.24 on page 34 and Figure 2.25 on page 35. We begin with a discussion of the procedure `aStarIdaStarSearch`.

1. The procedure takes 5 arguments.

   (a) `start` and `goal` are nodes. The procedure tries to find a path connecting `start` and `goal`.

   (b) `nextStates` is a function that takes a state s as input and computes the set of states that are reachable from s in one step.

   (c) `heuristic` computes an *estimate* of the distance between 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 or equal than the true distance between s_1 and s_2.

   (d) `size` is the maximal number of states that the A* search is allowed to explore before the algorithm switches over to IDA* search.

```
aStarIdaStarSearch := procedure(start, goal, nextStates, heuristic, size) {
    Parent   := {};
    Distance := { [start, 0] };
    est      := heuristic(start, goal);
    Estimate := { [start, est] };
    Frontier := { [est, start] };
    while (#Distance < size && Frontier != {}) {
        [guess, state] := first(Frontier);
        if (state == goal) {
            return pathTo(state, Parent);
        }
        stateDist := Distance[state];
        Frontier  -= { [guess, state] };
        for (neighbour in nextStates(state)) {
            oldEstimate := Estimate[neighbour];
            newEstimate := stateDist + 1 + heuristic(neighbour, goal);
            if (oldEstimate == om || newEstimate < oldEstimate) {
                Parent[neighbour]   := state;
                Distance[neighbour] := stateDist + 1;
                Estimate[neighbour] := newEstimate;
                Frontier            += { [newEstimate, neighbour] };
                if (oldEstimate != om) {
                    Frontier -= { [oldEstimate, neighbour] };
                }
            }
        }
    }
    [s, P] := deepeningSearch(goal, start, nextStates, heuristic, Distance);
    return pathTo(s, Parent) + P;
};
```

Figure 2.24: The A*-IDA* search algorithm, part I.

2. The basic idea behind the A*-IDA* algorithm is to first use A* search to find a path from `start` to `goal`. If this is successfully done without visiting more than `size` nodes, the algorithm terminates and returns the path that has been found. Otherwise, the algorithm switches over to an IDA* search that starts from `goal` and tries to connect goal to any of the nodes that have been encountered during the A* search. To this end, the procedure `aStarIdaStarSearch` maintains the following variables.

   (a) `Parent` is a dictionary associating a parent state with those states that have already been encountered during the search, i.e. we have

   $$\texttt{Parent}[s_2] = s_1 \;\Rightarrow\; s_2 \in \texttt{nextStates}(s_1).$$

   Once the goal has been found, this dictionary is used to compute the path from `start` to `goal`.

   (b) `Distance` is a dictionary that remembers for every state s that is encountered during the A* search the length of the shortest path from `start` to s.

   (c) `Estimate` is a dictionary. For every state s encountered in the A* search, $\texttt{Estimate}[s]$ is an estimate of the length that a path from `start` to `goal` would have if it would pass through the state s. This estimate is calculated using the equation

   $$\texttt{Estimate}[s] = \texttt{Distance}[s] + \texttt{heuristic}(s, \texttt{goal}).$$

   Instead of recalculating this sum every time we need it, we store it in the dictionary `Estimate`.

(d) `Frontier` is a priority queue. The elements of `Frontier` are pairs of the form

$$[d, s] \quad \text{such that} \quad d = \texttt{Estimate}[s],$$

i.e. if $[d, s] \in$ `Frontier`, then the state s has been encountered in the A* search and it is estimated that a path leading from `start` to `goal` and passing through s would have a length of d.

3. The A* search runs exactly as discussed previously. The only difference is that the `while` loop is terminated once the dictionary `Distance` has more than `size` entries. If we are lucky, the A* search is already able to find the `goal` and the algorithm terminates.

4. Otherwise, the procedure `deepeningSearch` is called. This procedure starts an iterative deepening A* search from the node `goal`. This search terminates as soon as a state is found that has already been encountered during the A* search. The set of these nodes is given to the procedure `deepeningSearch` via the parameter Distance. The procedure `deepeningSearch` returns a pair. The first component of this pair is the state `s`. This is the state in `Distance` that has been reached by the IDA* search. The second component is the path `P` that leads from the node `s` to the node `goal` but that does not include the node `s`. In order to compute a path from `start` to `goal`, we still have to compute a path from `start` to s. This path is then combined with the path `P` and the resulting path is returned.

```
1  deepeningSearch := procedure(g, s, nextStates, heuristic, Distance) {
2      limit := 0;
3      while (true) {
4          Path := search(g, s, nextStates, 0, limit, heuristic, [g], Distance);
5          if (isList(Path)) {
6              return Path;
7          }
8          limit := Path;
9      }
10 };
11 search := procedure(g, s, nextStates, d, l, heuristic, Path, Dist) {
12     total := d + heuristic(g, s);
13     if (total > l) {
14         return total;
15     }
16     if (Dist[g] != om) {
17         return [g, Path[2..]];
18     }
19     smallest := mathConst("Infinity");
20     for (ns in nextStates(g) | !(ns in Path)) {
21         result := search(ns, s, nextStates, d+1, l, heuristic, [ns]+Path, Dist);
22         if (isList(result)) {
23             return result;
24         }
25         smallest := min([smallest, result]);
26     }
27     return smallest;
28 };
```

Figure 2.25: The A*-IDA* search algorithm, part II.

Iterative deepening A*-IDA* is a complete search algorithm. On the instance of the 3×3 sliding puzzle shown on Figure 2.3 on page 10, this algorithm takes about 1.4 seconds to solve the puzzle. For the 4×4 sliding puzzle, if the algorithm is allowed to visit at most 3 000 states, the algorithm takes less than 9 seconds.

Exercise 4: Assume that you have 3 water buckets: The first bucket can hold 12 liters of water, the second bucket can hold 8 liters, while the last bucket can hold 3 liters. There are three types of action:

1. A bucket can be completely filled.
2. A bucket can be completely emptied.
3. The contend of one bucket can be poured into another bucket. Then, there are two cases.
   (a) If the second bucket has enough free space for all the water in the first bucket, then the first bucket is emptied and all the water from the first bucket is poured into the second bucket.
   (b) However, if there is not enough space in the second bucket, then the second bucket is filled completely, while the water that does not fit into the second bucket remains in the first bucket.

Your goal is to measure out exactly one liter. Write a program that computes a plan to achieve this goal. ◇

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 but that are nearer to the goal that 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 :=  +---+---+---+---+
          |14 |15 | 8 |12 |              | 1 | 2 | 3 | 4 |
          +---+---+---+---+              +---+---+---+---+
          |10 |11 | 9 |13 |              | 5 | 6 | 7 | 8 |
          +---+---+---+---+              +---+---+---+---+
          | 2 | 6 | 5 | 1 |              | 9 |10 |11 |12 |
          +---+---+---+---+              +---+---+---+---+
          | 3 | 7 | 4 |   |              |13 |14 |15 |   |
          +---+---+---+---+              +---+---+---+---+
```

This is one of the hardest instances of the fifteen puzzle. In order to solve the puzzle, we could try to first move the tile numbered with a 1 into the upper left corner. The resulting state would have the following form:

```
+---+---+---+---+
| 1 | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
```

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 upper left corner contains a 1. Once we have reached a state specified by the pattern given above, we could the proceed to reach a state that is described by the following pattern:

```
+---+---+---+---+
| 1 | 2 | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
```

This way, slowly but surely we will reach the goal. I have prepared a framework for taoistic search. The file

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/SetlX/sliding-puzzle-frame.stlx

contains a framework for the sliding puzzle where some functions are left unimplemented. The file

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/SetlX/a-star-lao-tzu.stlx

is also required. It contains an adapted form of A* search. More Details will be given in the lecture. Your task is to implement the missing functions in the file `sliding-puzzle-frame.stlx`. ◇

Chapter 3

Constraint Satisfaction

In this chapter we discuss algorithms for solving *constraint satisfaction problems*. This chapter is structured as follows:

1. The first section defines the notion of a constraint satisfaction problem. In order to illustrate this concept, two examples of constraint satisfaction problems are presented. After that, we discuss applications of constraint satisfaction problems.
2. The simplest algorithm to solve a constraint satisfaction problem is via *brute force search*. The idea behind brute force search is to test all possible variable assignments.
3. In most cases, the search space is too big to be enumerated completely. *Backtracking search* improves on brute force search by mixing the generation of assignments with the testing of the constraints. In most cases, this approach can drastically improve the performance of the search algorithm.
4. Backtracking search can be refined by using *constraint propagation* and by using the *most restricted variable* heuristic.
5. Furthermore, checking the *consistency* of the values assigned to different variables can reduce the size of the search space considerably.
6. Finally, *local search* is a completely different approach to solve constraint satisfaction problems.

3.1 Formal Definition of Constraint Satisfaction Problems

Formally, we define a constraint satisfaction problem as a triple

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

where

1. `Vars` is a set of strings which serve as *variables*,
2. `Values` is a set of *values* that can be assigned to the variables in `Vars`.
3. `Constraints` is a set of formulæ from *first order logic*. Each of these formulæ is called a *constraint* of $\mathcal{P}$.

   In order to be able to interpret these formulæ, we need a *first order structure* $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$. Here, $\mathcal{U}$ is the *universe* of $\mathcal{S}$ and we will assume that this universe is identical to the set `Values`. The second component $\mathcal{J}$ is the *interpretation* of the function symbols and predicate symbols that are used in the constraints. In what follows we assume that this interpretation is understood from the context of the constraint satisfaction problem $\mathcal{P}$.

In the following, the abbreviation Csp is short for *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

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

A variable assignment A is a *solution* of the Csp $\mathcal{P}$ if, given the assignment A, all constraints of $\mathcal{P}$ are satisfied. Finally, a *partial variable assignment* B for $\mathcal{P}$ is a function

$$B : \texttt{Vars} \to \texttt{Values} \cup \{\Omega\}.$$

Hence, a partial variable assignment does not assign values to all variables. Instead, it assigns values only to a subset of the set Vars. The *domain* $\texttt{dom}(B)$ of a partial variable assignment B is the set of those variables that are assigned a value different from Ω, i.e. we define

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

We proceed to illustrate the definitions given so far with two examples.

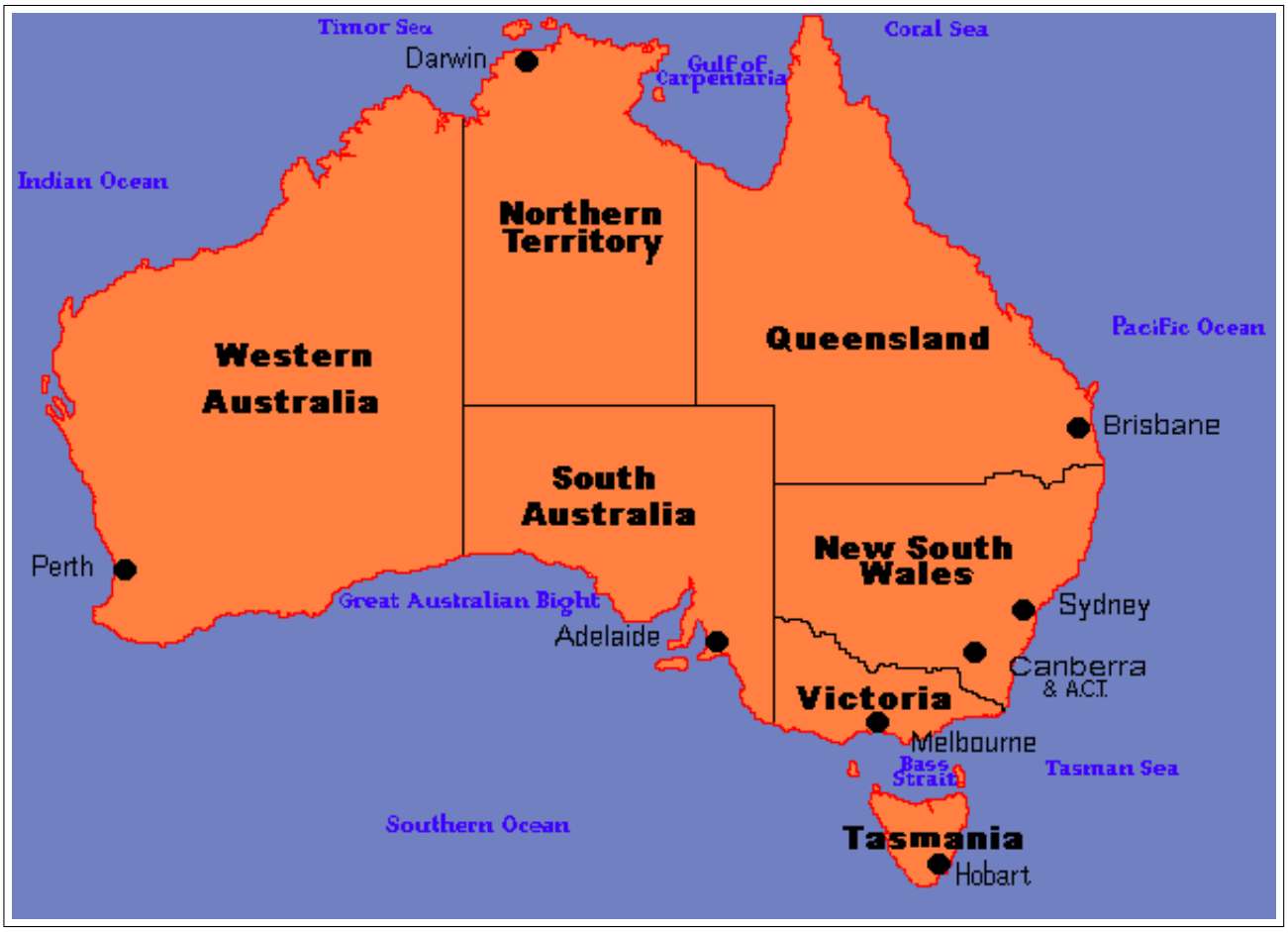

Figure 3.1: A map of Australia.

3.1.1 Example: Map Coloring

In map colouring a map showing different state 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 39 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
7. Tasmania, abbreviated as T.

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

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

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

1. $A(\text{WA}) = \texttt{blue}$,
2. $A(\text{NT}) = \texttt{red}$,
3. $A(\text{SA}) = \texttt{green}$,
4. $A(\text{Q}) = \texttt{blue}$,
5. $A(\text{NSW}) = \texttt{red}$,
6. $A(\text{V}) = \texttt{blue}$,
7. $A(\text{T}) = \texttt{red}$,

then you can check that the assignment A is indeed a solution to the constraint satisfaction problem $\mathcal{P}$.

3.1.2 Example: The Eight Queens Puzzle

The eight queens problem asks to put 8 queens onto 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 can 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 columns run from one to eight, we define the set $\texttt{Values}$ as

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

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

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

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

   Here the condition $i < j$ ensures that, for example, we have the constraint $\texttt{V}_2 \neq \texttt{V}_1$ but not the constraint $\texttt{V}_1 \neq \texttt{V}_2$, as the latter would be redundant if the former is already given.

2. We have constraints that express that no two queens positioned in different rows share the same rising diagonal. To capture these constraints, we define

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

3. We have constraints that express that no two queens positioned in different rows share the same falling diagonal. To capture these constraints, we define

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

Then, the set of constraints is defined as

$$\texttt{Constraints} := \texttt{SameRow} \cup \texttt{SameRising} \cup \texttt{SameFalling}$$

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

$$A(1) := 4,\ A(2) := 8,\ A(3) := 1,\ A(4) := 2,\ A(5) := 6,\ A(6) := 2,\ A(7) := 7,\ A(8) := 5,$$

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 41.

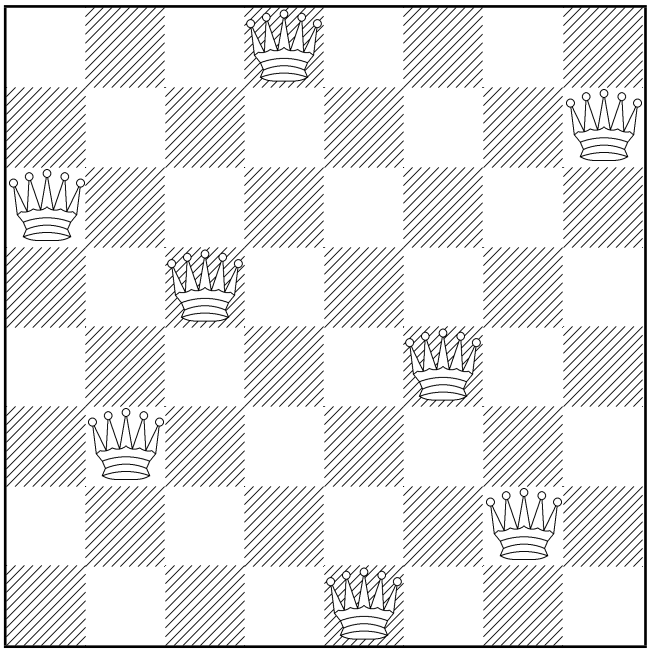

Figure 3.2: A solution of the eight queens problem.

Later, when we implement procedures to solve CSPs, we will represent variable assignments and partial variable assignments as binary relations. For example, A would then be represented as the relation

$$A = \big\{ \langle \text{V}_1, 4 \rangle, \langle \text{V}_2, 8 \rangle, \langle \text{V}_3, 1 \rangle, \langle \text{V}_4, 2 \rangle, \langle \text{V}_5, 6 \rangle, \langle \text{V}_6, 2 \rangle, \langle \text{V}_7, 7 \rangle, \langle \text{V}_8, 5 \rangle \big\}.$$

If we define

$$B := \big\{\langle \mathrm{V}_1, 4\rangle, \langle \mathrm{V}_2, 8\rangle, \langle \mathrm{V}_3, 1\rangle\big\},$$

then B is a partial assignment and $\texttt{dom}(B) = \{\mathrm{V}_1, \mathrm{V}_2, \mathrm{V}_3\}$. This partial assignment is shown in Figure 3.3 on page 42.

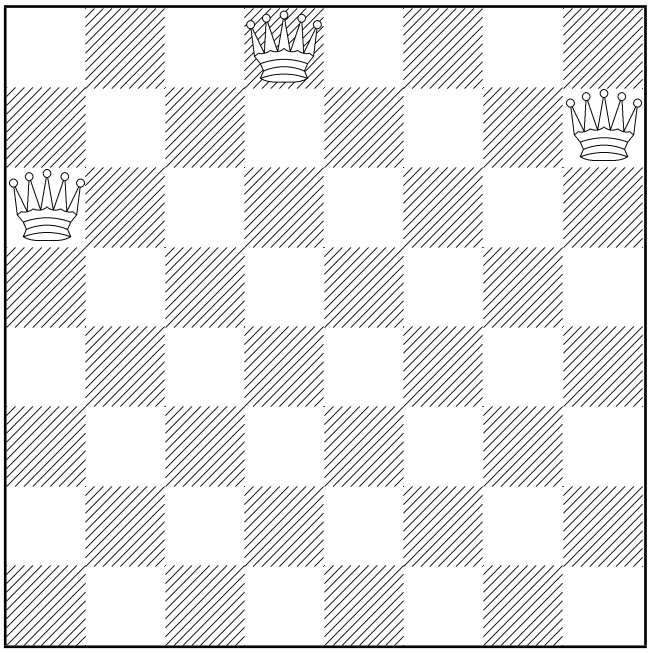

Figure 3.3: The partial assignment $\big\{\langle \mathrm{V}_1, 4\rangle, \langle \mathrm{V}_2, 8\rangle, \langle \mathrm{V}_3, 1\rangle\big\}$.

Figure 3.4 on page 42 shows a SetlX program that can be used to create the eight queens puzzle as a Csp. The code shown in this figure is more general than the eight queens puzzle: Given a natural number n, the function call `queensCSP`(n) creates a constraint satisfaction problem $\mathcal{P}$ that generalizes the eight queens problem to the problem of putting n queens on a board of size n times n.

```
1  queensCSP := procedure(n) {
2      Variables   := { "V$i$" : i in {1..n} };
3      Values      := { 1 .. n };
4      Constraints := {};
5      for (i in [2..n], j in [1..i-1]) {
6          Constraints += { "V$i$ != V$j$" };
7          Constraints += { "$i$ + V$i$ != $j$ + V$j$" };
8          Constraints += { "$i$ - V$i$ != $j$ - V$j$" };
9      }
10     return [Variables, Values, Constraints];
11 };
```

Figure 3.4: SetlX code to create the CSP representing the eight-queens puzzle.

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 then capable of computing a solution.

3.1.3 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 time table 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 time table.

3.2 Brute Force Search

The most straightforward algorithm to solve a Csp is to test all possible combinations of assigning values to variables. If there are n different values that can be assigned to k variables, this amounts to checking n^k different assignments. For example, for the eight queens problem there are 8 variables and 8 possible values leading to

$$8^8 = 16,777,216$$

different assignments that need to be tested. Given the clock speed of modern computers, checking a million assignments per second is plausible. Hence, this approach would be able to solve the eight queens problem in about 15 minutes. The approach of testing all possible combinations is known as *brute force search*. An implementation is shown in Figure 3.5 on page 43.

```
1  solve := procedure(csp) {
2      [Variables, Values, Constraints] := csp;
3      return brute_force_search({}, Variables, Values, Constraints);
4  };
5  brute_force_search := procedure(Assignment, Variables, Values, Constraints) {
6      if (Variables == {}) {
7          if (check_all_constraints(Assignment, Constraints)) {
8              return Assignment;
9          }
10         return;
11     }
12     var := from(Variables);
13     for (value in Values) {
14         NewAss := Assignment + { [var, value] };
15         result := brute_force_search(NewAss, Variables, Values, Constraints);
16         if (result != om) {
17             return result;
18         }
19     }
20 };
```

Figure 3.5: Solving a Csp via brute force search.

The procedure `solve` gets a constraint satisfaction problem `cps` as its input. This `csp` is given a triple. The sole purpose of `search` is to extract the components of this triple and then calls the procedure `brute_force_search` with the corresponding arguments.

The function `brute_force_search` takes four arguments.

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

2. `Variables` is the set of variables of the Csp that is to be solved. This set contains only those variables that have not yet been assigned a value.

3. `Values` is the set of values of this Csp.

4. `Constraints` is the corresponding set of constraints.

The implementation of `brute_force_search` works as follows:

1. If all variables have been assigned a value, the set `Variables` will be empty. Then we test whether all constraints are satisfied. This is done using the auxiliary procedure `check_all_constraints` that is shown in Figure 3.6 on page 44. If the current `Assignment` does indeed satisfy all constraints, it is a solution and is returned.

   If, instead, some constraint is not satisfied, then the procedure returns the undefined value Ω.

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

   `Assignment + { [var, value] }`.

   Then, the algorithm recursively tries to find a solution for this new partial assignment. If this recursive call succeeds, the solution is returned. Otherwise, the next `value` is tried.

```
21  check_all_constraints := procedure(Assignment, Constraints) {
22      for (f in Constraints) {
23          Vars := collectVars(f);
24          if (!eval_constraint(Assignment, f, Vars)) {
25              return false;
26          }
27      }
28      return true;
29  };
30  eval_constraint := procedure(Assignment, Formula, Vars) {
31      for (v in Vars) {
32          execute("$v$ := $Assignment[v]$;");
33      }
34      return eval(Formula);
35  };
```

Figure 3.6: Auxiliary procedures for brute force search.

The function `check_all_constraints` takes a complete variable `Assignment` as its first input. The second input is the set of `Constraints`. For all constraints `f`, it computes the set of variables `Vars` occurring in `f`. Then the constraint `f` is evaluated. If the evaluation of any of the constraints returns `false`, the function returns `false`. Otherwise, `true` is returned.

The procedure `eval_constraint` takes a partial `Assignment`, a `Formula` that is supposed to be a constraint, and the set of variables `Vars` that occur in `Formula`. Its purpose is to evaluate `Formula` using the `Assignment`. In order for this to be possible we have to assume that

$$\texttt{var}(\texttt{Formula}) \subseteq \texttt{dom}(\texttt{Assignment}),$$

i.e. all variables occurring in `Formula` have a value assigned in `Assignment`.

The `Formula` is evaluated by first assigning the value prescribed in `Assignment` to all variables `v` occurring in `Vars`. Once these assignments have been executed in the `for`-loop, the `Formula` can be evaluated using the procedure `eval`, which is one of the predefined procedures in SetlX.

When I tested this brute force search with the eight queens problem, it took about 510 seconds to compute a solution. In contrast, the seven queens problem only took 45 seconds. As we have

$$\frac{8^8}{7^7} \approx 20.3 \qquad \frac{510}{45} \approx 11.3$$

this shows that the computation time does indeed 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 space of possible assignments.

3.3 Backtracking Search

One simple approach to solve a Csp is via *backtracking*. Figure 3.7 on page 45 shows a simple Csp solver that employs backtracking. We discuss this program next.

```
1  solve := procedure(csp) {
2      [Vars, Vals, Constrs] := csp;
3      csp := [Vars, Vals, { [f, collectVars(f)] : f in Constrs }];
4      check {
5          return bt_search({}, csp);
6      }
7  };
8  bt_search := procedure(Assignment, csp) {
9      [Variables, Values, Constraints] := csp;
10     if (#Assignment == #Variables) {
11         return Assignment;
12     }
13     var := select_unassigned_variable(Assignment, Variables);
14     for (value in Values) {
15         check {
16             if (is_consistent(var, value, Assignment, Constraints)) {
17                 return bt_search(Assignment + { [var, value] }, csp);
18             }
19         }
20     }
21     backtrack;
22 };
```

Figure 3.7: A backtracking Csp solver.

The procedure `solve` takes a constraint satisfaction problem `csp` as input and tries to find a solution.

1. First, the `csp` 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`. These variables are then stored together with the constraint `f` and the correspondingly modified data structure is stored in `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 B, we only need to check the constraint `f` iff all of its variables are members of the domain of B.

3. Next, we call the procedure `bt_search` to compute a solution of `csp`. This procedure is enclosed in a `check`-block. Conceptually, this `check`-block is the same as if we had enclosed the call to `bt_search` in a `try-catch`-block as shown below:

   ```
   try {
   ```

```
        return bt_search({}, csp);
    } catch(e) {}
```

The point is that the procedure `bt_search` either returns a solution or, if it is not able to find a solution, it throws a special kind of exception, a so called *backtrack exception*. The `check`-block ensures that this exception is silently discarded. It is just a syntactical convenience that is more concise than using `try` and `catch`.

Next, we discuss the procedure `bt_search`. This procedure gets a partial assignment `Assignment` as input together with an augmented `csp`. This partial assignment is *consistent* with `csp`: 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 assignment that satisfies all constraints of the given `csp`.

1. First, the augmented `csp` is split into its components.
2. Next, if `Assignment` is already a complete variable assignment, it is a solution of `csp` and this solution is returned. Since `Assignment` is represented as a binary relation, `Assignment` is complete if its size is the same as the size of `Variables`.
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. This is done using the auxiliary procedure `select_unassigned_variable` that is shown in Figure 3.8 on page 46 and will be discussed later.
4. Next, it is tried to assign a `value` to the selected variable `var`. It is checked whether this assignment would be consistent using the procedure `is_consistent`. If the partial assignment turns out to be consistent, the partial `assignment` is extended to the new partial assignment

   `Assignment + { [var, value] }`.

   Then, the procedure `bt_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 `bt_search` will instead raise an exception. This exception is muted by the `check`-block that surrounds the call to `bt_search`. In that case, the `for`-loop generates a new possible `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 the `backtrack`-statement is executed. Effectively, this statement raises an exception that is caught by one of `check`-blocks.

```
1  select_unassigned_variable := procedure(Assignment, Variables) {
2      return rnd({ v : v in Variables | Assignment[v] == om });
3  };
4  is_consistent := procedure(var, value, Assignment, Constraints) {
5     NewAssignment := Assignment + { [var, value] };
6     for ([Formula, Vars] in Constraints | var in Vars) {
7           if (Vars <= domain(NewAssignment)) {
8              if (!eval_constraint(NewAssignment, Formula, Vars)) {
9                  return false;
10             }
11         }
12     }
13     return true;
14 };
```

Figure 3.8: Auxiliary procedures for the Csp solver shown in Figure 3.7

We still need to discuss the implementation of the auxiliary procedures shown in Figure 3.8 on page 46.

1. The procedure `select_unassigned_variable` takes a partial `Assignment` and the set of all `Variables`. It randomly selects a variable `v` such that

   $$\texttt{v} \notin \texttt{dom(Assignment)}$$

   which is the case if $\texttt{Assignment}[v] = \Omega$.

   It is certainly possible to be more clever here. We will later show that it is beneficial to select a variable that is a *most constrained variable*, i.e. a variable such that the number of values that can be assigned to this variable while still having a consistent assignment is minimal.

2. The procedure `is_consistent` takes a variable `var`, a `value`, a partial `Assignment` and a set of `Constraints`. 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 $\texttt{vars(f)} \subseteq \texttt{dom(Assignment)}$ the formula `f` evaluates to `true` using `Assignment`. The purpose of `is_consistent` is to check, whether the extended assignment

   $$\texttt{NewAssignment} := \texttt{Assignment} \cup \{\langle \texttt{var}, \texttt{value} \rangle\}$$

   that assigns `value` to the variable `var` is still partially consistent with `Constraints`. To this end, the `for`-loop iterates over all `Formula` in `Constraints`. However, we only have to check those `Formula` that contain the variable `var` and, furthermore, have all their variables occurring in $\texttt{dom(NewAssignment)}$. The reasoning is as follows:

   (a) If `var` does not occur in `Formula`, then adding `var` to `Assignment` cannot change the result of evaluating `Formula` and as `Assignment` is assumed to be partially consistent with respect to `Formula`, `NewAssignment` is also partially consistent with respect to `Formula`.

   (b) If $\texttt{dom(NewAssignment)} \not\subseteq \texttt{Vars}$, then `Formula` can not be evaluated anyway.

   Now if any of the `Formula` evaluates to `false`, then `NewAssignment` is not partially consistent and we can immediately return `false`. Otherwise, `true` is returned.

3.4 Constraint Propagation

Once we choose a value for a variable, this choice influences the values that are still available for other variables. For example, suppose we place the queen in row 1 in the second column, then no other queen can be placed in that column. Additionally, the queen in row 2 can then not be placed in any of the first three columns. It turns out that elaborating this idea can enhance the performance of backtracking search considerably. Figure 3.9 on page 48 shows an implementation of *constraint propagation*. This implementation is also able to handle *unary constraints*, i.e. constraints that contain only a single variable.

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 `v` we need to know which values are admissible for `v`. To this end, we need a dictionary that contains the set of possible values for every variable `v`. Initially, this dictionary assigns the set `Values` to every variable. Next, we take care of the unary constraints and shrink these sets accordingly. Then we solve the binary constraints and shrink these sets even more once we assign values to variables.

1. The procedure `solve` receives a `csp`. This `csp` is first split into its three components.

2. The first task of `solve` is to create 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. Next, for every constraint `f`, the dictionary `Annotated` attaches the variables occurring in a constraint `f` to `f`.

4. In order to solve the unary constraints we first have to find them. The set `UnaryConstrs` contains all those pairs $[\texttt{f}, \texttt{V}]$ from the set of annotated constraints `Annotated` such that the set of variables `V` contains just a single variable.

```
solve := procedure(csp) {
    [Variables, Values, Constrs] := csp;
    ValuesPerVar  := { [v, Values] : v in Variables };
    Annotated     := { [f, collectVars(f)] : f in Constrs };
    UnaryConstrs  := { [f, V] : [f, V] in Annotated | #V == 1 };
    BinaryConstrs := { [f, V] : [f, V] in Annotated | #V == 2 };
    msg := "Constraints should be either unary or binary!";
    assert(UnaryConstrs + BinaryConstrs == Annotated, msg);
    for ([f, V] in UnaryConstrs) {
        var               := arb(V);
        ValuesPerVar[var] := solve_unary(f, var, ValuesPerVar[var]);
    }
    check {
        return bt_search({}, [Variables, ValuesPerVar, BinaryConstrs]);
    }
};
```

Figure 3.9: Constraint Propagation.

5. Similarly, the set `BinaryConstrs` contains those constraints that involve two variables.
6. We assume that the set of constraints `Annotated` only contains unary and binary constraints.
7. In order to solve the unary constraints, we iterate over all unary constraints and shrink the set of values associated with the variable occurring in the constraint accordingly.
8. Then, the dictionaries `VariablesPerVar` and `BinaryConstrs` are combined with the set of `Variables` into a triple that is used as the argument to the call of `bt_search`.

```
solve_unary := procedure(constraint, variable, Values) {
    LegalValues := {};
    for (value in Values) {
        Assignment := { [variable, value] };
        if (eval_constraint(Assignment, constraint, { variable })) {
            LegalValues += { value };
        }
    }
    return LegalValues;
};
```

Figure 3.10: Implementation of `solve_unary`.

The function `solve_unary` shown in Figure 3.10 on page 48 takes a unary `constraint`, a `variable` and the set of `Values` that can be assigned to this `variable`. It returns the subset of values that satisfy the given `constraint`.

1. To achieve its goal, `solve_unary` iterates over all possible `Values`.
2. Next, for every `value` in the set `Values`, an `Assignment` is created that assign the `value` to this `variable`.
3. Then the `constraint` is evaluated with this `Assignment`.

4. If the `constraint` is satisfied under this `Assignment`, the `value` is added to the set `LegalValues`, which is the set of values which satisfy the unary `constraint`.
5. Finally, the set of `LegalValues` is returned.

```
1  bt_search := procedure(Assignment, csp) {
2      [Variables, Values, Constraints] := csp;
3      if (#Assignment == #Variables) {
4          return Assignment;
5      }
6      x := most_constrained_variable(Assignment, Values);
7      for (val in Values[x]) {
8          check {
9              if (is_consistent(x, val, Assignment, Constraints)) {
10                 NewVals := propagate(x, val, Assignment, Constraints, Values);
11                 csp     := [Variables, NewVals, Constraints];
12                 return bt_search(Assignment + { [x, val] }, csp);
13             }
14         }
15     }
16     backtrack;
17 };
```

Figure 3.11: Implementation of `bt_search`.

The procedure `bt_search` shown in Figure 3.11 on page 49 is called with a partial `Assignment` that is guaranteed to be consistent and a `csp`. It tries to complete `Assignment` and thereby computes a solution to the `csp`.

1. First, it decomposes the `csp` into its components:
   (a) `Values` is a dictionary assigning to every variable `v` the set of values that can be used to instantiate `v`. On recursive invocations of `bt_search` these sets will shrink.
   (b) `Constraints` is a dictionary that assigns to every constraint `f` the set of variables occurring in `f`.
2. 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.
3. Otherwise, we choose a variable `x` such that the number of values that can still be used to instantiate `x` is minimal. This variable is computed using the procedure `most_constrained_variable` that is shown in Figure 3.12 on page 50.
4. Next, all values that are still available for `x` are tried. Note that since `Values[x]` is, in general, smaller than the set of all values of the `csp`, the `for`-loop in this version of backtracking search is more efficient than the version discussed in the previous section.
5. If is is consistent to assign `val` to the variable `x`, we propagate the consequences of this assignment using the procedure `propagate` shown in Figure 3.12 on page 50. This procedure updates the values that are still allowed for the variables of the `csp` once the value `val` has been assigned to the variable `x`.
6. Finally, the partial variable `Assignment` is updated to include the assignment of `val` to `x` and the recursive call to `bt_search` tries to complete this new assignment.

Figure 3.12 on page 50 show the implementation of the procedures `most_constrained_variable` and `propagate`. The procedure `most_constrained_variable` takes a partial `Assignment` and a dictionary `Values` returning the set of allowed values for all variables as input.

```
most_constrained_variable := procedure(Assignment, Values) {
    Unassigned := { [x, U] : [x, U] in Values | Assignment[x] == om };
    minSize    := min({ #U : [x, U] in Unassigned });
    return rnd({ x : [x, U] in Unassigned | #U == minSize });
};
propagate := procedure(x, v, Assignment, Constraints, Values) {
    NewAssignment := Assignment + { [x, v] };
    Values[x]     := { v };
    for ([Formula, Vars] in Constraints | x in Vars) {
        y := arb(Vars - { x });  // Assume binary constraints!!!
        if (!(y in domain(NewAssignment))) {
            NewValues := Values[y];
            for (w in Values[y]) {
                A2 := NewAssignment + { [y, w] };
                if (!eval_constraint(A2, Formula, Vars)) {
                    NewValues -= { w };
                }
            }
            if (NewValues == {}) { backtrack; }
            Values[y] := NewValues;
        }
    }
    return Values;
};
```

Figure 3.12: Constraint Propagation.

1. First, this procedure 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 $[\texttt{x}, \texttt{U}]$, where `U` is the set of allowable values for the variable `x`.
2. Next, it computes from all unassigned variables `x` the number of values `#U` that can be assigned to `x`. From these numbers, the minimum is computed and stored in `minSize`.
3. Finally, the set of variables that are maximally constrained is computed. These are all those variables `x` such that `Values`[x] has size `minSize`. From these variables, a random variable is returned.

The function `propagate` takes the following inputs:

(a) `x` is a variable.

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

(c) `Assignment` is a partial assignment that contains only assignments for variables that are different from `x`.

(d) `Constraints` is a set of annotated constraints, i.e. this set contains pairs of the form $[\texttt{Formula}, \texttt{Vars}]$, where `Formula` is a constraint and `Vars` is the set of variables occurring in `Formula`.

(e) `Values` is a dictionary assigning sets of values to the different variables.

The function `propagate` updates the dictionary `Values` by taking into account the consequences of assigning the value `v` to the variable `x`.

1. The partial `Assignment` is updated to `NewAssignment`, where `NewAssignment` maps `x` to `v`.
2. As `x` now has the value of `v`, the corresponding entry in the dictionary `Values` is changed accordingly.

3. Next, `propagate` iterates over all `Constraints` such that `x` occurs in `Formula`.
4. The implementation of `propagate` shown in Figure 3.12 on page 50 assumes that all constraints are *binary constraints*, i.e. if `Formula` is a constraint, then `Formula` contains exactly two different variables. As one of these variables is `x`, the other variable is called `y`.
5. If the variable `y` has already been assigned a value, then there is nothing to do because we already know that the assignment `NewAssignment` is consistent. This is ensured by the previous call to the function `is_consistent` in the body of the procedure `bt_search`.

   However, if `y` is still unassigned, it is necessary to update `Values`[y].
6. To this end, all values `w` in `Values`[y] are tested. If it turns out that assigning the value `w` to the variable `y` violates the constraint `Formula`, then `Values`[y] must not contain the value `w` and, accordingly, this value is removed from `Values`[y].
7. If it turns out that `Values`[y] has been reduced to the empty set, then this means that the partial assignment `NewAssignment` can not be completed. Hence, the search has to `backtrack`.

I have tested the program described in this section using the n queens puzzle. I have found that the time needed to solve ten instances of 16 queens problem went down from 72 seconds to just 5 seconds. The procedure is even able to solve the 32 queens problem, taking about 4 seconds on average, while the version of backtracking search that does not use constraint propagation took more than 3 minutes on average.

Exercise 6: There are many different versions of the *zebra puzzle*. The version below is taken from *Wikipedia*. The puzzle reads 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. Who drinks water?
17. Who owns the zebra?

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

1. Each of the five houses is painted in a different color.

2. The inhabitants of the five houses are of different nationalities,
3. they own different pets,
4. they drink different beverages, and
5. they 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. You should also try to solve the puzzle using the program given in the previous section. Compare the results. ◇

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 does not apply. For example, consider the cryptarithmetic puzzle shown in Figure 3.13 on page 52. 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

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

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

Figure 3.13: 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:

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

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. In order to do better, we have to perform the addition in Figure 3.13 column by column, just as it is taught in elementary school. Figure 3.14 on page 53 shows how this can be implemented in SetlX.

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 do the addition of the last places of 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 reduce it to two equations that each involve only three variables as follows:

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

```
createCSP := procedure() {
    Variables   := { "S", "E", "N", "D", "M", "O", "R", "Y", "C1", "C2", "C3" };
    Values      := { 0 .. 9 };
    Constraints := allDifferent({ "S", "E", "N", "D", "M", "O", "R", "Y" });
    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" };
    return [Variables, Values, Constraints];
};
allDifferent := procedure(Vars) {
    return { "$x$ != $y$" : x in Vars, y in Vars | x != y };
};
```

Figure 3.14: Formulating "SEND + MORE = MONEY" as a CSP.

Here, the symbol "\" denotes integer division, e.g. we have $7 \backslash 3 = 2$. If we try to solve the cryptarithmetic puzzle as coded in Figure 3.14 on page 53 using the constraint solver developed in the previous section, we will be disappointed. The reason is that most constraints involve three variables and therefore the constraint propagation developed in the previous section is of no help. However, we can solve the problem in a few seconds 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 if our constraint solver would be able to figure out these constraints by itself. After all, since D and E are both less than 10, there 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

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

i.e. the variable x occurs in the constraint f and, furthermore, R is the set of all variables occurring in f that are different from x. Furthermore, assume that we have a dictionary AllowableValues such that for every variable y, the dictionary entry $\texttt{AllowableValues}[y]$ is the set of values that can be substituted for the variable y. Now *a value v is consistent for x with respect to the constraint f* iff the partial assignment $\{\langle x, v\rangle\}$ can be extended to an assignment A satisfying the constraint f, i.e. for every variable $y \in R$ we have to find a value $w \in \texttt{AllowableValues}[y]$ such that the resulting assignment A satisfies the equations

$$\texttt{evaluate}(f, A) = \texttt{true}.$$

Here, the function evaluate takes a formula f and an assignment A and evaluates f using the assignment A. Now, *consistency maintenance* works as follows.

1. The dictionary AllowableValues is initialized as follows:

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

   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:

$x :=$ `from(UncheckedVariables)`

4. We iterate over all constraints f such that x occurs in f.
   (a) For every value $v \in$ `AllowableValues`$[x]$ we check whether v is consistent with f.
   (b) If v is not consistent with f, then v is removed from `AllowableValues`$[x]$. Furthermore, if R is the set of all variables occurring in f that are different from x, then this set R is added to the set of `UncheckedVariables`.
5. Once `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 `AllowableValues`$[y]$ for some variable y. Now the set `UncheckedVariables` can grow during the algorithm, but the sets `AllowableValues`$[y]$ can never grow. However, every time the set `UncheckedVariables` grows, the set `AllowableValues`$[x]$ shrinks. As the sets `AllowableValues`$[y]$ are finite for all variables y, 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.

```
1  enforceConsistency := procedure(rw ValuesPerVar, Var2Formulas, Annotated, Connected) {
2      UncheckedVars := domain(Var2Formulas);
3      while (UncheckedVars != {}) {
4          variable    := from(UncheckedVars);
5          Constraints := Var2Formulas[variable];
6          Values      := ValuesPerVar[variable];
7          RemovedVals := {};
8          for (f in Constraints) {
9              OtherVars := Annotated[f] - { variable };
10             for (value in Values) {
11                 if(!existsValue(variable, value, f, OtherVars, ValuesPerVar)) {
12                     RemovedVals   += { value };
13                     UncheckedVars += Connected[variable];
14                 }
15             }
16         }
17         Remaining := Values - RemovedVals;
18         if (Remaining == {}) { backtrack; }
19         ValuesPerVar[variable] := Remaining;
20     }
21 };
```

Figure 3.15: Consistency maintenance in SetlX.

Figure 3.15 on page 54 shows how consistency maintenance can be implemented in SetlX. The procedure `enforceConsistency` 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 v, `Var2Formulas`$[v]$ is the set of those constraints f such that v occurs in f.

(c) `Annotated` is the set of annnotated constraints, i.e. this set contains pairs of the form $\langle f, V \rangle$ where f is a constraint and V is the set of all variables occurring in f. This last argument is needed only for efficiency: In order to avoid computing the set V of variables occurring in a constraint f every time the constraint f is

encountered, we compute these sets at the beginning of our computation and store them in the dictionary `Annnotated`.

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

The procedure `enforceConsistency` modifies the dictionary `ValuesPerVar` so that once the procedure has terminated, for every variable x the set $\texttt{ValuesPerVar}[x]$ is 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 $\texttt{ValuesPerVar}[x]$ has not been established.
3. Next, a `variable` is selected and removed from `UncheckedVars`.
4. `Constraints` is the set of all constraints f such that this `variable` occurs in f.
5. `Values` is the set of those values that can be assigned to `variable`.
6. `RemovedVals` is the subset of those values that are found to be inconsistent with some constraint.
7. We iterate over all constraints $\texttt{f} \in \texttt{Constraints}$.
8. `OtherVars` is the set of variables occurring in `f` that are different from the chosen `variable`.
9. We iterate over all $\texttt{value} \in \texttt{Values}$ that can be substituted for `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 `existsValue`.
10. If we do not find such values, then `value` is inconsistent for `variable` w.r.t. `f` and needs to be removed from the set $\texttt{ValuesPerVar}[\texttt{variable}]$. Furthermore, all variables that are connected to `variables` have to be added to the set `UncheckedVars`. The reason is that once a value is removed for `variable`, the value assigned to another variable y occurring in a constraint that mentions both `variable` and y might now become inconsistent.
11. If there are no consistent values for `variable` left, we have to backtrack.

Figure 3.16 on page 56 shows the implementation of the function `existsValue` that is used in the implementation of `enforceConsistency`. This procedure 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 `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 procedure checks whether the partial assignment $\{\langle \texttt{var}, \texttt{val}\rangle\}$ 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 `createAllAssignments`. 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`.

1. If the set of variables `Vars` is empty, the empty set can serve as a dictionary that assigns a value to every variable in `Vars`.

```
existsValue := procedure(var, val, f, Vars, ValuesPerVar) {
    AllVars := { var } + Vars;
    for (A in createAllAssignments(Vars, ValuesPerVar)) {
        if (eval_constraint(A + { [var, val] }, f, AllVars)) {
            return true;
        }
    }
    return false;
};
createAllAssignments := procedure(Vars, ValuesPerVar) {
    if (Vars == {}) {
        return { {} };  // set containing empty assignment
    }
    var         := from(Vars);
    Values      := ValuesPerVar[var];
    Assignments := createAllAssignments(Vars, ValuesPerVar);
    return { { [var, val] } + A : val in Values, A in Assignments };
};
```

Figure 3.16: The implementation of `existsValue`.

2. Otherwise, we remove a variable `var` from `Vars` and get the set of `Values` that can be assigned to `var`.
3. Recursively, we create the set of all `Assignments` that associate values with the remaining variables.
4. 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`.

On one hand, consistency checking 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 propagation. On the other hand, many difficult constraint satisfaction problems can not be solved without consistency checking.

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. Initialize the values of the variables in `Variables` randomly.
2. If all `Constraints` are satisfied, return the solution.
3. For every $x \in$ `Variables`, count the number of unsatisfied constraints that involve the variable x.
4. Set `maxNum` to be the maximum of these numbers, i.e. `maxNum` is maximal number of unsatisfied constraints for any variable.
5. Compute the set `maxVars` of those variables that have `maxNum` unsatisfied constraints.
6. Randomly choose a variable x from the set `maxVars`.

[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 greatly. I have refrained from implementing this optimization because I did not want the code to get too complex.

7. Find a value $d \in$ `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.

8. Goto step 2 and repeat until either a solution is found or the sun rises in the west.

```
1  solve := procedure(n) {
2      Queens := [];
3      for (row in [1 .. n]) {
4          Queens[row] := rnd({1 .. n});
5      }
6      iteration := 0;
7      while (true) {
8          Conflicts   := { [numConflicts(Queens, row), row] : row in [1 ..n] };
9          [maxNum, _] := last(Conflicts);
10         if (maxNum == 0) {
11             return Queens;
12         }
13         if (iteration % 10 != 0) { // avoid infinite loops
14             row := rnd({ row : [num, row] in Conflicts | num == maxNum });
15         } else {
16             row := rnd({ 1 .. n });
17         }
18         Conflicts := {};
19         for (col in [1 .. n]) {
20             Board      := Queens;
21             Board[row] := col;
22             Conflicts  += { [numConflicts(Board, row), col] };
23         }
24         [minNum, _] := first(Conflicts);
25         Queens[row] := rnd({ col : [num, col] in Conflicts | num == minNum });
26         iteration   += 1;
27     }
28 };
```

Figure 3.17: Solving the n queens problem using local search.

Figure 3.17 on page 57 shows an implementation of these ideas in SetlX. Instead of solving an arbitrary constraint satisfaction problem, the program solves the n queens problem. We proceed to discuss this program line by line.

1. The procedure `solve` takes one parameter `n`, which is the size of the chess board. If the computation is successful, `solve`(`n`) returns a list of length `n`. Lets call this list `Queens`. For every row `r` $\in \{1, \cdots, n\}$, the value `Queens`[`r`] specifies that the queen that resides in row `r` is positioned in column `Queens`[`r`].

2. The `for` loop initializes the positions of the queens to random values from the set $\{1, \cdots, n\}$. Effectively, for every row on the chess board, this puts a queen in a random column.

3. The variable `iteration` counts the number of times that we need to reassign a queen in a given row.

4. All the remaining statements are surrounded by a `while` loop that is only terminated once a solution has been found.

5. The variable `Conflicts` is a set of pairs of the form $[c, r]$, where c is the number of times the queen in row r is attacked by other queens. Hence, c is the same as the number of unsatisfied conflicts for the variable specifying the column of the queen in row r.

6. `maxNum` is the maximum of the number of conflicts for any row.

7. If this number is 0, then all constraints are satisfied and the list `Queens` is a solution to the `n` queens problem.

8. Otherwise, we compute those rows that exhibit the maximal number of conflicts. From these rows we select one `row` arbitrarily.

9. The reason for enclosing the assignment to `row` in an `if` statement is explained later. On a first reading of this program, this `if` statement should be ignored.

10. Now that we have identified the `row` where the number of conflicts is biggest, we need to reassign `Queens[row]`. Of course, when reassigning this variable, we would like to have fewer conflicts after the reassignment. Hence, we test all columns to find the best column that can be assigned for the queen in the given `row`. This is done in a `for` loop that runs over all possible columns. The set `Conflicts` that is maintained in this loop is a set of pairs of the form $[k, c]$ where k is the number of times the queen in `row` would be attacked if it would be placed in column c.

11. We compute the minimum number of conflicts that is possible for the queen in `row` and assign it to `minNum`.

12. From those columns that minimize the number of violated constraints, we choose a column randomly and assign it for the specified `row`.

There is a technical issue, that must be addressed: It is possible there is just one row that exhibits the maximum number of conflicts. It is further possible that, given the placements of the other queens, there is just one optimal column for this row. In this case, the procedure `solve` would loop forever. To avoid this case, every 10 iterations we pick a random row to change.

```
1  numConflicts := procedure(Queens, row) {
2      n      := #Queens;
3      result := 0;
4      for (r in {1 .. n} | r != row) {
5          if ( Queens[r] == Queens[row]           ||
6               r - Queens[r] == row - Queens[row] ||
7               r + Queens[r] == row + Queens[row]
8             )
9          { result += 1; }
10     }
11     return result;
12 };
```

Figure 3.18: The procedure `numConficts`.

The procedure `numConficts` shown in Figure 3.18 on page 58 implements the function `numConficts`. Given a board `Queens` that specifies the positions of the queens on the board and a `row`, this function computes the number of ways that the queen in `row` is attacked by other queens. If all queens are positioned in different rows, then there are only three ways left that a queen can be attacked by another queen.

1. The queen in row `r` could be positioned in the same column as the queen in `row`.

2. The queen in row `r` could be positioned in the same falling or rising diagonal as the queen in `row`. These diagonals are specified by the linear equations given in line 6 and 7 of Figure 3.18.

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. Hence, it seems that local search is much better than the algorithms discussed previously. 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 similar performance as 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, many practically occurring constraint satisfaction problems can be solved today in a reasonably short time.

Chapter 4

Playing Games

One major success 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, today according to the computer chess rating list top performing chess programs like Stockfish have an Elo rating of 3392. To compare, according to Fide, the current World Chess Champion Magnus Carlsen has an Elo rating of just 2838. Hence, he wouldn't stand a chance to win a game against Stockfish. More recently, the computer program AlphaGo was able to beat Lee Sedol, who is considered to be the second best 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, at the beginning of 2017 the program Libratus was able to beat four professional poker players resoundingly.

In this chapter we want to investigate how a computer can play a game. To this end we define a *game* $\mathcal{G}$ as a six-tuple

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

where the components are interpreted as follows:

1. `States` is the set of all possible *states* of the game.
2. $s_0 \in \texttt{startState}$ is the *start state*.
3. `Players` is the set of *players* of the game.
4. `nextStates` is a function that takes a state and a player p and returns the set of states that can be reached if p has to make a move in the game. Hence, the signature of `nextStates` is given as follows:

   $$\texttt{nextStates} : \texttt{State} \times \texttt{Player} \rightarrow 2^{\texttt{States}}.$$

5. `finished` is a function that takes a state s and decides whether the games is finished. Therefore

   $$\texttt{finished} : \texttt{States} \rightarrow \mathbb{B}.$$

   Here, $\mathbb{B}$ is the set of Boolean values, i.e. we have $\mathbb{B} := \{\texttt{true}, \texttt{false}\}$.

   Using the function `finished`, we define the set `TerminalStates` as the set of those states such that the game has finished, i.e. we define

   $$\texttt{TerminalStates} := \{s \in \texttt{States} \mid \texttt{finished}(s)\}.$$

6. `utility` is a function that takes a state $s \in \texttt{TerminalStates}$ and a player p. If returns the *value* that the game has for player p. In all of our examples, this value will be an element from the set $\{-1, 0, +1\}$. The value -1 indicates that player s has lost the game, if the value is $+1$ the player p has won the game and if this value is 0 then the game is drawn. Hence the signature of `utility` is

   $$\texttt{utility} : \texttt{TerminalStates} \times \texttt{Players} \rightarrow \{-1, 0, +1\}.$$

Exercise 7: The definition given above does not capture all types of games. In particular, the definition only captures *deterministic games*: A game is called *deterministic* iff there is no randomness involved. Games like chess and go are certainly deterministic. However, other games like dice chess involve randomness. In dice chess a pair of dice is rolled at the beginning of each turn. The outcome of this roll determines which pieces may be moved. Extend the definition of a game so that also a games like *dice chess* be modelled. ◇

In this chapter we will only consider so called *two person zero sum games*. This means that the set `Players` has exactly two elements. If we call these players A and B, i.e. if we have

$$\texttt{Players} = \{\mathrm{A}, \mathrm{B}\},$$

then the game is called a *zero sum game* iff we have

$$\forall s \in \texttt{TerminalStates} : \texttt{utility}(s, \texttt{A}) + \texttt{utility}(s, \texttt{B}) = 0,$$

i.e. the losses of player A are compensated by the wins of player `B` and vice versa. Games like go and chess are two person zero sum games.

Example: The game tic-tac-toe is played on a square board of size 3×3. On every turn, player A puts an "`X`" on one of the free squares of the board, while player B puts an `O` onto a free square. If player A has three `X`s in a row, column, or diagonal, he has won the game. Similarly, if player B has three `O`s in a row, column, or diagonal, player B is the winner. Otherwise, the game is drawn.

Figure 4.1 on page 62 shows a SetlX implementation of tic-tac-toe.

1. The function `players` returns the set `Players`. Traditionally, the players in tic-tac-toe are called "`X`" and "`O`".

2. The function `startState` returns the start state, which is an empty board. States are represented as list of lists. The entries in these lists are the characters "`X`", "`O`", and " ". As the `StartState` is the empty board, it is represented as as follows:

   ```
   [ [" ", " ", " "],
     [" ", " ", " "],
     [" ", " ", " "]
   ].
   ```

   The function `startState` receives one optional argument `n`. By default `n` is equal to 3. This argument specifies the size of the board. Although traditionally tic-tac-toe is played on a 3×3 board, it can also be played on a bigger board.

3. The function `nextStates` takes a `State` and a `player` and computes the set 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. Every position is represented as pair of the form [`row`, `col`] where `row` specifies the row and `col` specifies the column of the position. A position is *empty* iff

   $$[\texttt{row}, \texttt{col}] = \texttt{" "}.$$

   The computation of the empty position has been sourced out to the function `find_empty`. The function `nextStates` then iterates over the set of 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 set `Result` and returned.

4. The function `find_empty` takes a `State` and returns the set of empty positions.

5. The function `utility` takes a `State` and a `player`. If the game is finished in the given `State`, it returns the value that this `State` has for the current `player`. If the game is not yet decided, Ω is returned instead.

   In order to achieve its goal, the procedure first computes the set of all *lines*. Given a `State`, a *line* is a set of those markers that either form a horizontal, vertical, or diagonal line in `State`, e.g. the set

```
players := procedure() { return { "X", "O" }; };
startState := procedure(n) {
    L := [1 .. n];
    return [ [ " " : col in L] : row in L];
};
nextStates := procedure(State, player) {
     Empty  := find_empty(State);
    Result := {};
    for ([row, col] in Empty) {
        NextState           := State;
        NextState[row][col] := player;
        Result              += { NextState };
    }
    return Result;
};
find_empty := procedure(State) {
    n := #State;
    L := [1 .. n];
    return { [row, col] : row in L, col in L | State[row][col] == " " };
};
utility := procedure(State, player) {
    Lines := all_lines(State);
    for (line in Lines) {
        if (#{ x : x in line } == 1 && line != { " " }) {
            if (line == { player }) { return  1; } else { return -1; }
        }
    }
    if (find_empty(State) == {}) { return 0; }
};
all_lines := procedure(State) {
    n := #State;
    L := [1 .. n];
    Lines := { { State[row][col] : col in L } : row in L };
    Lines += { { State[row][col] : row in L } : col in L };
    Lines += { { State[idx][idx] : idx in L } };
    Lines += { { State[idx][n - (idx - 1) ] : idx in L } };
    return Lines;
};
finished := procedure(State) {
    player := "X";
    if (utility(State, player) != om) { return true; } else { return false; }
};
```

Figure 4.1: A SetlX description of tic-tac-toe.

$$\{\texttt{State}[1,1], \texttt{State}[2,2], \texttt{State}[3,3]\}$$

is the set of entries in a diagonal line. The game is decided if all entries in this set are either "`X`" or "`O`". In this case, the set has exactly one element which is different from the blank. If this element is the same as `player`, then the game is won by `player`, otherwise it is lost.

If there are no empty squares left, then the game is a draw.

6. The auxiliary procedure `all_lines` takes a `State` and computes the sets of all *lines*.

7. The procedure `finished` takes a `State` and checks whether the game is finished. To this end it computes the `utility` of the state for the player "`X`". If this `utility` is different from Ω, the game is finished. Note that it makes no difference whether we take the utility of the state for the player "`X`" or for the player "`O`" since if the game is finished for "`X`" it is also finished for "`O`" and vice versa.

4.1 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 easiest algorithm that works is the minimax algorithm. This algorithm is based on the notion of the *value* of a state. To this end, we define a function

$$\texttt{value} : \texttt{States} \times \texttt{player} \to \{-1, 0, +1\}$$

that takes a state s and a player p and returns the value that s has if both the player p and his opponent play perfectly. The easiest way to define this function is via recursion. The base case is simple:

$$\texttt{finished}(s) \to \texttt{value}(s, p) = \texttt{utility}(s, p).$$

If the game is not yet finished, assume that o is the opponent of p. Then we define

$$\neg\texttt{finished}(s) \to \texttt{value}(s, p) = \max\bigl(\bigl\{-\texttt{value}(s, o) \mid n \in \texttt{nextStates}(s, p)\bigr\}\bigr).$$

The reason is that if the game is not finished yet, the player p has to evaluate all possible moves. Then the player will choose the best move. In order to do so, the player computes the set $\texttt{nextStates}(s, p)$ of all states that can be reached in one move from the state s. Now if n is a state that results from player p making a move, the other player o has to make his move in the state n. As we are investigation zero sum games, we have $\texttt{value}(n, p) = -\texttt{value}(n, o)$. Figure 4.2 on page 64 shows an implementation of this strategy.

1. The implementation of the function `value` follows the reasoning outlined above. However, note that we have implemented `value` as a `cachedProcedure`, i.e. as a procedure that memorizes its results. Hence, when it is called a second time with the same arguments, it does not recompute the value but rather the value is looked up. To understand why this is important, let us consider how many states would be explored in the case of tic-tac-toe if we would not use memoization. In this case, we have 9 moves for player "`O`" from the start state, then 8 moves for player "`X`", then again 7 moves for player "`X`". If we disregard the fact that some games are decided after fewer than 9 moves, the function `value` needs to consider

   $$9 \cdot 8 \cdot 7 \cdot \ldots \cdot 2 \cdot 1 = 9! = 362880$$

   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

   $$\binom{9}{5} = \frac{9!}{5! \cdot 4!} = 126$$

   possibilities, i.e. there are a factor of $5! \cdot 4! = 2880$ less states to evaluate!

2. The function `best_move` takes a `State` and a `player` and returns a state s that is optimal for the `player` and such that s can be reached in one step from `State`.

   (a) To this end, it first computes the set `AllStates` of all states that can be reached from the given `State` in one step if `player` is to move next.

   (b) Since there are only two players, `other` is the opponent of `player`.

   (c) `bestValue` is the best value that `player` can achieve.

   (d) The function returns randomly one of those states `s` $\in$ `AllStates` such that the value of s is maximal, i.e. is equal to `bestValue`.

```
value := cachedProcedure(State, player) {
    if (finished(State)) {
        return utility(State, player);
    }
    other := arb(players() - { player });
    return max({ -value(s, other) : s in nextStates(State, player) });
};
best_move := procedure(State, player) {
    AllStates := nextStates(State, player);
    other     := arb(players() - { player });
    bestValue := max({ -value(s, other) : s in AllStates });
    return rnd({ s : s in AllStates | -value(s, other) == bestValue });
};
play_game := procedure(n) {
    State := startState(n);
    print(stateToString(State));
    while (true) {
        State := best_move(State, "O");
        print("My move:");
        print(stateToString(State));
        if (final_msg(State)) { return; }
        State := getMove(State);
        print(stateToString(State));
        if (final_msg(State)) { return; }
    }
};
```

Figure 4.2: The Minimax algorithm.

3. The function `play_game` is used to play a game.
   (a) Since we would prefer to have
   (b) Initially, `State` is the `startState`.
   (c) As long as the game is not finished, the procedure keeps running.
   (d) We assume that the computer is player "`O`" and that the computer goes first. Hence, the function `best_move` is used to compute the state that results from the best move of player "`O`".
   (e) After that, it is checked whether the game is finished.
   (f) If the game is not yet finished, the human is asked to make its move via the function `getMove` that takes a `State`, displays it, and asks the user to enter a move. The state resulting from this move is then returned and displayed.
   (g) Next, we have to check whether the game has finished.
   (h) The `while`-loop keeps iterating until the game is decided.

4.2 α-β-Pruning

The minimax algorithm can be improved by if the function `value` gets two additional parameters, α and β. The idea is that

`valueAlphaBeta`(`State`, `player`, `alpha`, `beta`)

still computes the value of `State` as long as this value is strictly between α and β, i.e. we have

$$v = \texttt{value}(\texttt{State}, \texttt{player}) \land \alpha < v \land v < \beta \rightarrow \texttt{valueAlphaBeta}(\texttt{State}, \texttt{player}, \texttt{alpha}, \texttt{beta}) = v$$

However, if $v \leq \alpha$ the value returned by `valueAlphaBeta` is less than α, while if $v \geq \beta$ it will be bigger than β. The idea is that if v is outside the interval $[\alpha, \beta]$, then we essentially do not care about the exact value that is returned and, therefore, are able to shortcircuit the evaluation of the given `State`. This can result in significant speed improvements. Figure **??** on page **??** shows an implementation of the function `valueAlphaBeta` that works along this line of thinking.

```
1  valueAlphaBeta := cachedProcedure(State, player, alpha := -1, beta := 1) {
2      if (finished(State)) {
3          return utility(State, player);
4      }
5      other  := arb(players() - { player });
6      maxVal := -1;
7      for (s in nextStates(State, player)) {
8          val := -value(s, other, -beta, -alpha);
9          if (val >= beta) {
10             return val + 1/2;
11         }
12         maxVal := max({val, maxVal});
13     }
14     return maxVal;
15 };
```

Figure 4.3: α-β-Pruning.

1. If `State` is a terminal state, the function returns the `utility` of the given `State` with respect to `player`.
2. Otherwise, we iterate over all successor states $\texttt{s} \in \texttt{nextStates}(\texttt{State}, \texttt{player})$.
3. As we want to find the successor state with the highest value, we initialize `maxVal` to -1 since this is the lowest value that is possible and therefore -1 we have

   $$\max(v, -1) = v \quad \text{for all values } v \text{ that can occur.}$$

Chapter 5

Linear Regression

A great deal of the current success of artificial intelligence is due to recent advances in machine learning. *Linear regression* is one of the most basic algorithms that are applied in machine learning. It is also the foundation for more advanced forms of machine learning like *logistic regression* and *neural networks*. Furthermore, linear regression is surprisingly powerful. Finally, many of the fundamental problems of machine learning can already be illustrated with linear regression. Therefore it is only natural that we begin our study of machine learning with the study of linear regression.

5.1 Simple Linear Regression

Assume we want to know how the engine displacement of a car engine relates to the fuel consumption of the car. Of course, we could try to create a theoretical model that relates the engine displacement to its fuel consumption. However, due to our lack of understanding of engine physics, this is not an option for us. Instead, we could try to take a look at different engines and compare their engine displacement with the corresponding fuel consumption. This way, we would collect a set of m observations of the form $\langle x_1, y_1\rangle, \cdots, \langle x_m, y_m\rangle$ where x_i is the engine displacement of the engine in the i-th car, while y_i is the consumption of the i-th car. We call x is the *independent variables*, while y is the *dependent variable*. In linear regression, we use a *linear hypothesis* and assume that the dependent variable y_i is related to the independent variable x_i via an equation of the form

$$y_i := \vartheta_1 \cdot x_i + \vartheta_0.$$

Of course, we do not expect this equation to hold exactly. The reason is that there are many other factors that influence the fuel consumption. For example, the weight of a car certainly influences the fuel consumption. We want to calculate those values ϑ_0 and ϑ_1 such that the *mean squared error*, which is defined as

$$\texttt{MSE}(\vartheta_0, \vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2, \tag{5.1}$$

is minimized. It can be shown that the solution to this minimization problem is given as follows:

$$\vartheta_1 = r_{x,y} \cdot \frac{s_y}{s_x} \quad \text{and} \quad \vartheta_0 = \bar{y} - \vartheta_1 \cdot \bar{x}. \tag{5.2}$$

Here, $\bar{x}$ and $\bar{y}$ are the *sample mean values* of x and y respectively, i.e. we have

$$\bar{x} = \frac{1}{m} \cdot \sum_{i=1}^{m} x_i \quad \text{and} \quad \bar{y} = \frac{1}{m} \cdot \sum_{i=1}^{m} y_i.$$

Furthermore, s_x and s_y are the *sample standard deviations* of x and y, i.e. we have

$$s_x = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(x_i - \bar{x}\big)^2} \quad \text{and} \quad s_y = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(y_i - \bar{y}\big)^2}.$$

Finally, $r_{x,y}$ is the *sample correlation coefficient* that is defined as

$$r_{x,y} = \frac{1}{(m-1)\cdot s_x \cdot s_y} \cdot \sum_{i=1}^{m} \big(x_i - \bar{x}\big) \cdot \big(y_i - \bar{y}\big).$$

The number $r_{x,y}$ is also known as the Pearson correlation coefficient or *Pearson's r*. It is named after Karl Pearson (1857 – 1936). Note that the formula for the parameter ϑ_1 can be simplified to

$$\vartheta_1 = \frac{\sum\limits_{i=1}^{m} \big(x_i - \bar{x}\big) \cdot \big(y_i - \bar{y}\big)}{\sum\limits_{i=1}^{m} \big(x_i - \bar{x}\big)^2} \tag{5.3}$$

This latter formula should be used to calculate ϑ_1. However, the previous formula is also useful because it shows the the correlation coefficient is identical to the coefficient ϑ_1, provided the variables x and y have been normalized so that their standard deviation is 1.

Exercise 8: Proof Equation 5.2 and Equation 5.3.

Hint: Take the partial derivatives of $\texttt{MSE}(\vartheta_0, \vartheta_1)$ with respect to ϑ_0 and ϑ_1. In order to minimize the expression $\texttt{MSE}(\vartheta_0, \vartheta_1)$ with respect to ϑ_0 and ϑ_1, these derivatives have to be set to 0. ◇

5.1.1 Accessing the Quality of Linear Regression

Assume that we have been given a set of m observations of the form $\langle x_1, y_1\rangle, \cdots, \langle x_m, y_m\rangle$ and that we have calculated the parameters ϑ_0 and ϑ_1 according to Equation 5.2 and Equation 5.3. Provided that not all x_i have the same value, these formulæ will returns two numbers. The question then is how good this linear approximation realy is. In order to judge the quality of the linear model that is given by

$$y = \vartheta_0 + \vartheta_1 \cdot x$$

we can compute the mean squared error according to Equation 5.1. However, the mean squared error is an absolute number that, by itself, is difficult to interpret. The reason is that the variable y might be inherently noisy and we have to relate this noise to the mean squared error. Now the noise contained in y can be measured by the sample variance of y and is given by the formula

$$\texttt{Var}(y) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(y_i - \bar{y}\big)^2. \tag{5.4}$$

If we compare this formula to the formula for the mean squared error

$$\texttt{MSE}(\vartheta_0, \vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2,$$

we see that the sample variance of y is an upper bound for the mean squared error since we have

$$\texttt{Var}(y) = \texttt{MSE}(\bar{y}, 0),$$

i.e. the sample variance is the value that we would get for the mean squared error if we would set ϑ_0 to the average value of y and ϑ_1 to zero. Since ϑ_0 and ϑ_1 are chosen to minimize the mean squared error, we have

$$\texttt{MSE}(\vartheta_0, \vartheta_1) \leq \texttt{Var}(y).$$

The mean squared error is an absolute value and, therefore, difficult to interpret. The fraction

$$\frac{\texttt{MSE}(\vartheta_0, \vartheta_1)}{\texttt{Var}(y)}$$

is called the portion of the *unexplained variance* because it is the variance that is still left if we use our linear model to predict the values of y given the values of x. The *explained variance* which is also known as the $\texttt{R}^2$

statistic is defined as

$$\mathtt{R}^2 := \frac{\mathtt{Var}(y) - \mathtt{MSE}}{\mathtt{Var}(y)} = 1 - \frac{\mathtt{MSE}}{\mathtt{Var}(y)}. \tag{5.5}$$

Here MSE is short for $\mathtt{MSE}(\vartheta_0, \vartheta_1)$ where we have substituted the values from equation 5.2 and 5.3. The $\mathtt{R}^2$ statistic measures the quality of our model: If it is small, then our model does not explain the variation of the value of y when the value of x changes. On the other hand, if it is near a 100%, then our model does a good job in explaining the variation of y when x changes.

Since the formulæ for $\mathtt{Var}(y)$ and $\mathtt{MSE}(\vartheta_0, \vartheta_1)$ have the same denominator $m - 1$, this denominator can be cancelled when the $\mathtt{R}^2$ statistic is computed. To this end we define the *total sum of squares* TSS as

$$\mathtt{TSS} := \sum_{i=1}^{m} \big(y_i - \bar{y}\big)^2 = (m-1) \cdot \mathtt{Var}(y)$$

and the *residual sum of squares* RSS as

$$\mathtt{RSS} := \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2 = (m-1) \cdot \mathtt{MSE}(\vartheta_0, \vartheta_1).$$

Then the formula for the $\mathtt{R}^2$ statistic can be written as

$$\mathtt{R}^2 = 1 - \frac{\mathtt{RSS}}{\mathtt{TSS}}.$$

This is the formula that we will use when we implement simple linear regression.

5.1.2 Putting the Theory to the Test

In order to get a better feeling for linear regression, we want to test it to investigate the factors that determine the fuel consumption of cars. Figure 5.1 on page 68 shows the head of the data file "`cars.csv`" which I have adapted from the file

`http://www-bcf.usc.edu/~gareth/ISL/Auto.csv`.

Figure 5.1 on page 68 shows the column headers and the first ten data entries contained in this file. Altogether, this file contains data of 392 different car models.

```
 1   mpg, cyl, displacement,    hp, weight,  acc, year, name
 2  18.0,   8,        307.0, 130.0, 3504.0, 12.0,   70, chevrolet chevelle malibu
 3  15.0,   8,        350.0, 165.0, 3693.0, 11.5,   70, buick skylark 320
 4  18.0,   8,        318.0, 150.0, 3436.0, 11.0,   70, plymouth satellite
 5  16.0,   8,        304.0, 150.0, 3433.0, 12.0,   70, amc rebel sst
 6  17.0,   8,        302.0, 140.0, 3449.0, 10.5,   70, ford torino
 7  15.0,   8,        429.0, 198.0, 4341.0, 10.0,   70, ford galaxie 500
 8  14.0,   8,        454.0, 220.0, 4354.0,  9.0,   70, chevrolet impala
 9  14.0,   8,        440.0, 215.0, 4312.0,  8.5,   70, plymouth fury iii
10  14.0,   8,        455.0, 225.0, 4425.0, 10.0,   70, pontiac catalina
11  15.0,   8,        390.0, 190.0, 3850.0,  8.5,   70, amc ambassador dpl
```

Figure 5.1: The head of the file `cars.csv`.

The file "`cars.csv`" is part of the data set accompanying the excellent book Introduction to Statistical Learning by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani [1]. The file "`cars.csv`" contains the fuel consumption of a number of different cars that were in widespread use during the seventies and eighties of the last century. The first column of this data set contains the *miles per gallon*, which is the reciprocal of the fuel consumption. Furthermore, for every car the following other parameters are listed:

1. `cyl` is the number of cylinders,
2. `displacement` is the engine displacement in cubic inches,
3. `hp` is the horse power,
4. `weight` is the weight in pounds,
5. `acc` is the acceleration given as the time in seconds needed to accelerate from 0 miles to 60 miles,
6. `year` is the year in which the model was introduced, and
7. `name` is the name of the model.

Our aim is to determine which of the parameters can be used to explain the fuel consumption of a car. To this end, I have written the program shown in Figure 5.2 on page 69.

```
1  simple_linear_regression := procedure(fileName, types, name) {
2      csv    := readTable(fileName, types);
3      data   := csv.getData();
4      number := #data;
5      index  := find_index(name, csv.getColumnNames());
6      Y      := [];
7      X      := [];
8      L      := [1 .. number];
9      for (i in L) {
10         Y[i] := 1 / data[i][1];
11         X[i] := data[i][index];
12     }
13     xMean  := +/ X / number;
14     yMean  := +/ Y / number;
15     theta1 :=   (+/ [(X[i]-xMean)*(Y[i]-yMean) : i in L])
16                / (+/ [(X[i]-xMean)**2 : i in L]);
17     theta0 := yMean - theta1 * xMean;
18     TSS    := +/ [(Y[i] - yMean) ** 2 : i in L];
19     RSS    := +/ [(theta0 + theta1 * X[i] - Y[i]) ** 2 : i in L];
20     R2     := 1 - RSS / TSS;
21     return R2;
22 };
23 find_index := procedure(name, List) {
24     index := 0;
25     for (s in List) {
26         index += 1;
27         if (s == name) {
28             return index;
29         }
30     }
31     assert(false, "$name$ not found in $List$");
32 };
```

Figure 5.2: Simple Linear Regression

The procedure `simple_linear_regression` take three arguments:

(a) `fileName` is the name of the file containing the data.

It is assumed that this file is a `csv`-file containing the data that need to be analyzed.

(b) `types` is a list of type names of the columns present in the `csv` file.

(c) `name` is the column name that is used for linear regression, i.e. it specifies the column that is used as the values for the independent variable x. The dependent variable is assumed to be given in the first column of the `csv`-file.

The implementation of the procedure `simple_linear_regression` works as follows:

1. The function `readTable` is used to read the files specified in `fileName`. It returns the object `csv` which is of class `Table`. This class and the implementation of the function `readTable` is shown in Figure 5.3 on page 71. The object `csv` contains both the column names as well as the data that is present in the given file. The list `types` is needed in order to convert the strings that are read into the proper data types.
2. The function `getData` extracts the `data` from the object `csv`. Technically, `data` is a list of lists. `data` has the same length as there are lines in the file specified by `fileName`. Every single line in this file is converted into a list of entries of the appropriate types.
3. `number` is the number of data lines in the file specified by `fileName`.
4. `index` is the index of the column that has the given `name`.
5. The values of the dependent variable y are stored in the list `Y` and the values of the independent variable x are stored in the list `X`.
6. The list `L` is used as an abbreviation so that iterations over all lists stored in `data` are simplified.
7. The `for`-loop fills the lists `X` and `Y`. Since the file "`cars.csv`" contains the miles per gallon in the first column labelled `mpg`, we need to convert this data into fuel consumption. As the fuel consumption is inverse to the miles per gallon, the list `Y` is filled with the reciprocal of the first column of the file "`cars.csv`".
8. `xMean` is the mean value $\bar{x}$ of the independent variable x.
9. `yMean` is the mean value $\bar{y}$ of the dependent variable y.
10. The coefficient `theta1` is computed according to Equation 5.3, which is repeated here for convenience:
$$\vartheta_1 = \frac{\sum_{i=1}^{m} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sum_{i=1}^{m} (x_i - \bar{x})^2}.$$
11. The coefficient `theta0` is computed according to Equation 5.2, which reads
$$\vartheta_0 = \bar{y} - \vartheta_1 \cdot \bar{x}.$$
12. `VarY` computes the sample variance of the dependent variable y.
13. `TSS` is the total sum of squares and computed using the formula
$$\texttt{TSS} = \sum_{i=1}^{m} (y_i - \bar{y})^2.$$
14. `RSS` is the residual sum of squares and computed as
$$\texttt{RSS} := \sum_{i=1}^{m} (\vartheta_1 \cdot x_i + \vartheta_0 - y_i)^2.$$
15. `R2` is the $\texttt{R}^2$ statistic and measures the portion of the explained variance. It is computed using the formula
$$\texttt{R}^2 = \frac{\texttt{TSS} - \texttt{RSS}}{\texttt{TSS}}.$$

Figure 5.3 on page 71 shows the implementation of the class table and the function `readTable`. This figure is only given for completeness.

```
class Table(columnNames, types, data) {
    mColumnNames := columnNames;
    mTypes       := types;
    mData        := data;
  static {
      getColumnNames := [ ] |-> mColumnNames;
      getTypes       := [ ] |-> mTypes;
      getData        := [ ] |-> mData;
      getRow         := [r] |-> mData[r];
      getLength      := [ ] |-> #mData;

      head := procedure(limit := 10) {
          print(mColumnNames);
          print(mTypes);
          for (i in [1 .. limit]) {
              print(mData[i]);
          }
      };
  }
}
readTable := procedure(fileName, types) {
    allData     := readFile(fileName);  // list of lines
    columnNames := split(allData[1], ',\s*');
    data        := [];
    for (i in [2 .. #allData]) {
        row       := split(allData[i], ',\s*');
        data[i-1] := [evalType(type, s) : [type, s] in types >< row];
    }
    return Table(columnNames, types, data);
};
evalType := procedure(type, s) {
    switch {
        case     type == "double": return double(s);
        case     type == "int"   : return int(s);
        case     type == "string": return s;
        default: abort("unknown type $type$ in evalType");
    }
};
```

Figure 5.3: The class `Table`.

Figure 5.4 on page 72 shows how to call the procedure `simple_linear_regression`. We need to define the types of the various columns that are present in the file "`cars.csv`". Next, for all those column names present in this file, we use this variable to compute the explained variance. The resulting values are shown in Table 5.1. It seems that, given the data in the file "`cars.csv`", the best indicator for the fuel consumption is the `weight` of a car. The `displacement`, the power `hp` of an engine, and the number of cylinders `cyl` are also good predictors. But notice that the `weight` is the real cause of fuel consumption. Now if a car has a big weight, it will also need a more powerful engine. Hence the variable `hp` is correlated with the variable `weight` and will therefore also provide a reasonable explanation of the fuel consumption, although it is not the real cause.

```
test := procedure() {
    types := ["double", "int", "double", "double", "double", "double", "int", "string"];
    for (varName in ["displacement", "cyl", "hp", "weight", "acc", "year"]) {
        R2 := simple_linear_regression("cars.csv", types, varName);
        print("The explained variance for $varName$ vs fuel consumption is $R2$.");
    }
};
```

Figure 5.4: Calling the procedure `simple_linear_regression`.

dependent variable	explained variance
`displacement`	0.75
`cyl`	0.70
`hp`	0.73
`weight`	0.78
`acc`	0.21
`year`	0.31

Table 5.1: Explained variance for various dependent variables.

5.1.3 Testing the Statistical Significance

In this section we answer the question how to access the statistical significance of our results. In the case of simple linear regression, the null hypothesis is given as follows:

$\mathtt{H}_0$: There is no relationship between the variables x and y.

In order to compute the probability that the null hypothesis is true, we have to compute the *t-statistic*, which is given by the following formula:

$$t := |\vartheta_1| \cdot \sqrt{\frac{(m-2)}{\mathtt{RSS}} \cdot \sum_{i=1}^{m} \big(x_i - \bar{x}\big)^2}\,. \tag{5.6}$$

The t statistics is distributed according to Student's t-distribution with $m-2$ degrees of freedom. In SetlX, the cumulative Student's t-distribution can be computed via the function

`stat_studentCDF`(t, ν),

where t is the value of the t-statistic and ν is the number of degrees of freedom. If we substitute $\nu := m-2$, then this function computes the probability that the null hypothesis is true. This probability is also known as the *p-value*. If the p-value is small, e.g. less than 0.01, then the null hypothesis is refuted and we can conclude that there is some relationship between x and y. In the example previously discussed, i.e. the example relating the fuel consumption of a car to various other parameters, the p-values are all very small, i.e. less than 10^{-16}. This is due to the fact that we have had enough data at our disposal: As a rule of thumb we can say that once we have more than 30 pairs $\langle x_i, y_i \rangle$ and there is indeed a relationship between x and y, then simple linear regression will detect this relationship, even if the relationship is only small.

5.2 General Linear Regression

In practise, it is rarely the case that a given observed variable y only depends on a single variable x. To take the example of the fuel consumption of a car further, in general we would expect that the fuel consumption of a car does depend not only on the mass of the car but is also related to the other parameters. These kinds of problems are addressed by general linear regression. In a *general regression problem* we are given a list of

n pairs of the form $\langle \mathbf{x}^{(i)}, y^{(i)} \rangle$ where $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and $y^{(i)} \in \mathbb{R}$ for all $i \in \{1, \cdots, n\}$. These pairs are called the *training examples*. Our goal is to compute a linear function

$$F : \mathbb{R}^m \to \mathbb{R}$$

such that $F\big(\mathbf{x}^{(i)}\big)$ approximates $y^{(i)}$ as precisely as posssible for all $i \in \{1, \cdots, n\}$, i.e. we want to have

$$\forall i \in \{1, \cdots, n\} : F\big(\mathbf{x}^{(i)}\big) \approx y^{(i)}.$$

In order to make the notation $F\big(\mathbf{x}^{(i)}\big) \approx y^{(i)}$ more precise, we define the *squared error*

$$E := \frac{1}{n} \cdot \sum_{i=1}^{n} \Big(F\big(\mathbf{x}^{(i)}\big) - y^{(i)} \Big)^2. \tag{5.7}$$

Then, given the list of training examples $[\langle \mathbf{x}^{(1)}, y^{(1)} \rangle, \cdots, \langle \mathbf{x}^{n}, y^{(n)} \rangle]$, our goal is to minimize the squared error E. In order to proceed, we need to have a model for the function F. The simplest model is a linear model, i.e. we assume that F is given as

$$F(\mathbf{x}) = \sum_{i=1}^{m} w_i \cdot x_i + b = \mathbf{x}^T \cdot \mathbf{w} + b \quad \text{where } \mathbf{w} \in \mathbb{R}^m \text{ and } b \in \mathbb{R}.$$

Here, the expression $\mathbf{x}^T \cdot \mathbf{w}$ denotes the matrix product of the vector $\mathbf{x}^T$, which is viewed as a 1-by-m matrix, and the vector $\mathbf{w}$. At this point you might wonder why it is useful to introduce matrix notation here. The reason is that this notation shortens the formula and, furthermore, is more efficient to implement since most programming languages used in machine learning have special library support for matrix operations.

The definition of F given above is the model used in linear regression. Here, $\mathbf{w}$ is called the *weight vector* and b is called the *bias*. It turns out that the notation can be simplified if we extend the m-dimensional vector $\mathbf{x}$ to an $m+1$-dimensional vector $\mathbf{x}'$ such that

$$x'_j := x_j \quad \text{for all } j \in \{1, \cdots, m\} \quad \text{and} \quad x'_{m+1} := 1.$$

To put it in words, the vector $\mathbf{x}'$ results from the vector $\mathbf{x}$ by appending the number 1:

$$\mathbf{x}' = \langle x_1, \cdots, x_m, 1 \rangle^T \quad \text{where } \langle x_1, \cdots, x_m \rangle = \mathbf{x}^T.$$

Furthermore, we define

$$\mathbf{w}' := \langle w_1, \cdots, w_m, b \rangle^T \quad \text{where } \langle w_1, \cdots, w_m \rangle = \mathbf{w}^T.$$

Then we have

$$F(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = \mathbf{w}' \cdot \mathbf{x}'.$$

Hence, the bias has been incorporated into the weight vector at the cost of appending a 1 at the input vector. As we want to use this simplification, from now on we assume that the input vectors $\mathbf{x}^{(i)}$ have all been extended so that there last component is 1. Using this assumption, we define the function F as

$$F(\mathbf{x}) := \mathbf{x}^T \cdot \mathbf{w}.$$

Now equation (5.7) can be rewritten as follows:

$$E(\mathbf{w}) = \frac{1}{n} \cdot \sum_{i=1}^{n} \Big((\mathbf{x}^{(i)})^T \cdot \mathbf{w} - y^{(i)} \Big)^2. \tag{5.8}$$

Our aim is to rewrite the sum appearing in this equation as a scalar product of a vector with itself. To this end, we first define the vector $\mathbf{y}$ as follows:

$$\mathbf{y} := \langle y^{(1)}, \cdots, y^{(m)} \rangle^T.$$

Note that $\mathbf{y} \in \mathbb{R}^n$ since it has a component for all of the n training examples. Next, we define the matrix X as follows:

$$X := \begin{pmatrix} \big(\mathbf{x}^{(1)}\big)^T \\ \vdots \\ \big(\mathbf{x}^{(n)}\big)^T \end{pmatrix}$$

Defined this way, the row vectors of the matrix X are the vectors $\mathbf{x}^{(i)}$ transposed. Now we have the following:

$$X \cdot \mathbf{w} - \mathbf{y} = \begin{pmatrix} \big(\mathbf{x}^{(1)}\big)^T \\ \vdots \\ \big(\mathbf{x}^{(n)}\big)^T \end{pmatrix} \cdot \mathbf{w} - \mathbf{y} = \begin{pmatrix} \big(\mathbf{x}^{(1)}\big)^T \cdot \mathbf{w} - y_1 \\ \vdots \\ \big(\mathbf{x}^{(n)}\big)^T \cdot \mathbf{w} - y_n \end{pmatrix}$$

Taking the square of the vector $X \cdot \mathbf{w} - \mathbf{y}$ we discover that we can rewrite equation (5.8) as follows:

$$E(\mathbf{w}) = \frac{1}{n} \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big)^T \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big). \tag{5.9}$$

5.2.1 Some Useful Gradients

In the last section, we have computed the squared error $E(\mathbf{w})$ using equation (5.9). Our goal is to minimize the $E(\mathbf{w})$ by choosing the weight vector $\mathbf{w}$ appropriately. A necessary condition for $E(\mathbf{w})$ to be minimal is

$$\nabla E(\mathbf{w}) = \mathbf{0},$$

i.e. the gradient of $E(\mathbf{w})$ need to be zero. In order to prepare for the computation of $\nabla E(\mathbf{w})$, we first compute the gradient of two simpler functions.

Computing the Gradient of $f(\mathbf{x}) = \mathbf{x}^T \cdot C \cdot \mathbf{x}$

Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$f(\mathbf{x}) := \mathbf{x}^T \cdot C \cdot \mathbf{x} \quad \text{where } C \in \mathbb{R}^{n \times n}.$$

If we write the matrix C as $C = (c_{i,j})_{\substack{i=1,\cdots,n \\ j=1,\cdots,n}}$ and the vector $\mathbf{x}$ as $\mathbf{x} = \langle x_1, \cdots, x_n \rangle^T$, then $f(\mathbf{x})$ can be computed as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i \cdot \sum_{j=1}^{n} c_{i,j} \cdot x_j = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i \cdot c_{i,j} \cdot x_j.$$

We compute the partial derivative of f with respect to x_k and use the product rule:

$$\begin{aligned} \frac{\partial f}{\partial x_k} &= \sum_{i=1}^{n} \sum_{j=1}^{n} \Big(\frac{\partial x_i}{\partial x_k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \frac{\partial x_j}{\partial x_k} \Big) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \Big(\delta_{i,k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \delta_{j,k} \Big) \\ &= \sum_{j=1}^{n} c_{k,j} \cdot x_j + \sum_{i=1}^{n} x_i \cdot c_{i,k} \\ &= \big(C \cdot \mathbf{x}\big)_k + \big(C^T \cdot \mathbf{x}\big)_k \end{aligned}$$

Hence we have shown that

$$\nabla f(\mathbf{x}) = (C + C^T) \cdot \mathbf{x}.$$

If the matrix C is symmetric, i.e. if $C = C^T$, this simplifies to

$$\nabla f(\mathbf{x}) = 2 \cdot C \cdot \mathbf{x}.$$

If the function $g : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$g(\mathbf{x}) := \mathbf{b}^T \cdot A \cdot \mathbf{x}, \quad \text{where } \mathbf{b} \in \mathbb{R}^n \text{ and } A \in \mathbb{R}^{n \times n},$$

then a similar calculation shows that

$$\nabla g(\mathbf{x}) = A^T \cdot \mathbf{b}.$$

Exercise 9: Prove this equation.

5.3 Deriving the Normal Equation

Next, we derive the so called *normal equation* for linear regression. To this end, we first expand the product in equation (5.9):

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{n} \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big)^T \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big) \\ &= \frac{1}{n} \cdot \big(\mathbf{w}^T \cdot X^T - \mathbf{y}^T\big) \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big) && \text{since } (A \cdot B)^T = B^T \cdot A^T \\ &= \frac{1}{n} \cdot \big(\mathbf{w}^T \cdot X^T \cdot X \cdot \mathbf{w} - \mathbf{y}^T \cdot X \cdot \mathbf{w} - \mathbf{w}^T \cdot X^T \cdot \mathbf{y} + \mathbf{y}^T \cdot \mathbf{y}\big) \\ &= \frac{1}{n} \cdot \big(\mathbf{w}^T \cdot X^T \cdot X \cdot \mathbf{w} - 2 \cdot \mathbf{y}^T \cdot X \cdot \mathbf{w} + \mathbf{y}^T \cdot \mathbf{y}\big) && \text{since } \mathbf{w}^T \cdot X^T \cdot \mathbf{y} = \mathbf{y}^T \cdot X \cdot \mathbf{w} \end{aligned}$$

The fact that

$$\mathbf{w}^T \cdot X^T \cdot \mathbf{y} = \mathbf{y}^T \cdot X \cdot \mathbf{w}$$

might not be immediately obvious. This is equation is true because the result of the matrix product $\mathbf{w}^T \cdot X^T \cdot \mathbf{y}$ is a real number. Now the transpose r^T of a real number r is the number itself, i.e. $r^T = r$ for all $r \in \mathbb{R}$. Therefore, we have

$$\mathbf{w}^T \cdot X^T \cdot \mathbf{y} = \big(\mathbf{w}^T \cdot X^T \cdot \mathbf{y}\big)^T = \mathbf{y}^T \cdot X \cdot \mathbf{w}.$$

Hence we have shown that

$$E(\mathbf{w}) = \frac{1}{n} \cdot \Big(\mathbf{w}^T \cdot \big(X^T \cdot X\big) \cdot \mathbf{w} - 2 \cdot \mathbf{y}^T \cdot X \cdot \mathbf{w} + \mathbf{y}^T \cdot \mathbf{y}\Big) \tag{5.10}$$

holds. The matrix $X^T \cdot X$ used in the first term is symmetric because

$$\big(X^T \cdot X\big)^T = X^T \cdot \big(X^T\big)^T = X^T \cdot X.$$

Using the results from the previous section we can now compute the gradient of $E(\mathbf{w})$ with respect to $\mathbf{w}$. The result is

$$\nabla E(\mathbf{w}) = \frac{2}{n} \cdot \Big(X^T \cdot X \cdot \mathbf{w} - X^T \cdot \mathbf{y}\Big).$$

If the squared error $E(\mathbf{w})$ has a minimum for the weights $\mathbf{w}$, then we must have

$$\nabla E(\mathbf{w}) = \mathbf{0}.$$

This leads to the equation

$$\frac{2}{n} \cdot \Big(X^T \cdot X \cdot \mathbf{w} - X^T \cdot \mathbf{y}\Big) = \mathbf{0}.$$

This equation can be rewritten as

$$\big(X^T \cdot X\big) \cdot \mathbf{w} = X^T \cdot \mathbf{y}. \tag{5.11}$$

This equation is called the *normal equation*. Now, if the matrix $X^T \cdot X$ is invertible, then this equation can be rewritten as

$$\mathbf{w} = (X^T \cdot X)^{-1} \cdot X^T \cdot \mathbf{y}.$$

In this case, we define

$$X^+ := (X^T \cdot X)^{-1} \cdot X^T.$$

The expression X^+ is called the *Moore-Penrose pseudoinverse* of the matrix X. We can understand in what sense X^+ is an inverse of X by noting that if X itself were invertible, we could solve the equation

$$X \cdot \mathbf{w} = \mathbf{y}$$

by defining

$$\mathbf{w} := X^{-1} \cdot \mathbf{y}.$$

In that case, the squared error would be zero. In general, X will not be invertible for the simple reason that the matrix X is an $n \times m$ matrix and hence is not even a square matrix if $m \neq n$. In practical applications of linear regression the number n of training examples is bigger than the dimension m of the vectors $\mathbf{x}_i$ that make up the training examples. Hence there is no hope of X being invertible. The next best thing is then to minimize the squared error. To do this, we take the previous equation for the weight vector $\mathbf{w}$ and replace X^{-1} with the pseudoinverse of X. This gives the equation

$$\mathbf{w} = (X^T \cdot X)^{-1} \cdot X^T \cdot \mathbf{y}$$

derived previously.

At this point you might ask whether the matrix $X^T \cdot X$ is always invertible. The short answer is yes, provided the regression problem is well posed. By this I mean that there are much more training examples than there are different weights. After all, if you have only 10 training examples but want to determine a 100 different weights, then it is obvious that the problem is not well posed. Now if you have enough training examples you have to be sure that the training examples are independent of each other. For example, lets say you have three training examples $\langle \mathbf{x}_1, y^{(1)} \rangle$, $\langle \mathbf{x}_2, y^{(2)} \rangle$, and $\langle \mathbf{x}_3, y^{(3)} \rangle$ and you decide that you define a fourth training example $\langle \mathbf{x_4}, y^{(4)} \rangle$ as

$$\mathbf{x}_4 := \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 \quad \text{and} \quad y^{(4)} := y^{(1)} + y^{(2)} + y^{(3)},$$

then you have not generated any new information and the fourth training example will not help in making the matrix $X^T \cdot X$ invertible.

5.3.1 Testing the Statistical Significance

$$F = \frac{\texttt{TSS} - \texttt{RSS}}{\texttt{RSS}} \cdot \frac{m - p - 1}{p} \tag{5.12}$$

The F-statistic is distributed according to the Fisher-Snedecor-distribution with p degrees of freedom in the nominator and $m - p - 1$ degrees of freedom in the denominator.

Bibliography

[1] Trevor Hastie Gareth James, Daniela Witten and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R.* Springer, 2014.

[2] 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.

[3] 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.

[4] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Limited, 3rd edition, 2009.