Duale Hochschule Baden-Württemberg Mannheim

# An Introduction to Artificial Intelligence

— Lecture Notes in Progress —

Prof. Dr. Karl Stroetmann

February 16, 2017

# 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. We will look at these different answers and discuss them.

1. *Artificial Intelligence* is the study of creating machines that <u>think</u> 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 knowledge of the human thought processes is just not sufficient.

2. *Artificial Intelligence* is the science of machines that <u>act</u> 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 plans 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 <u>think logically</u>.

   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. Technically, this approach is based on mathematical logic. The approach had 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) 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 the best decisions. 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 mathematical statistics. The shift from logic to statistics has been the most important reason for the success of Artificial Intelligence in the recent years.

Now that we have clarified the notion of artificial intelligence, we should set its goals. As we can never achieve more than 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 ··· |
| | 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 [2].

# 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, and
6. bidirectional A$^*$-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

   $$\texttt{nextState} : S \times A \rightarrow 2^S.$$

   The function `nextState` 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:

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

   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 $g$ 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. $s_1 = \texttt{start}$, 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. ◇

**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 $\textit{problem}(m, i)$ is `true` iff there 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

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

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

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

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

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 state of the search problem. The reason is that, in order to solve the problem, we do not need to define this set in order to solve the problem. If we want to, we can 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. ◇

Next, we want to develop an algorithm that can solve puzzles of the kind of the missionaries and cannibals problem automatically. The easiest algorithm to solve search problems is breadth first search.

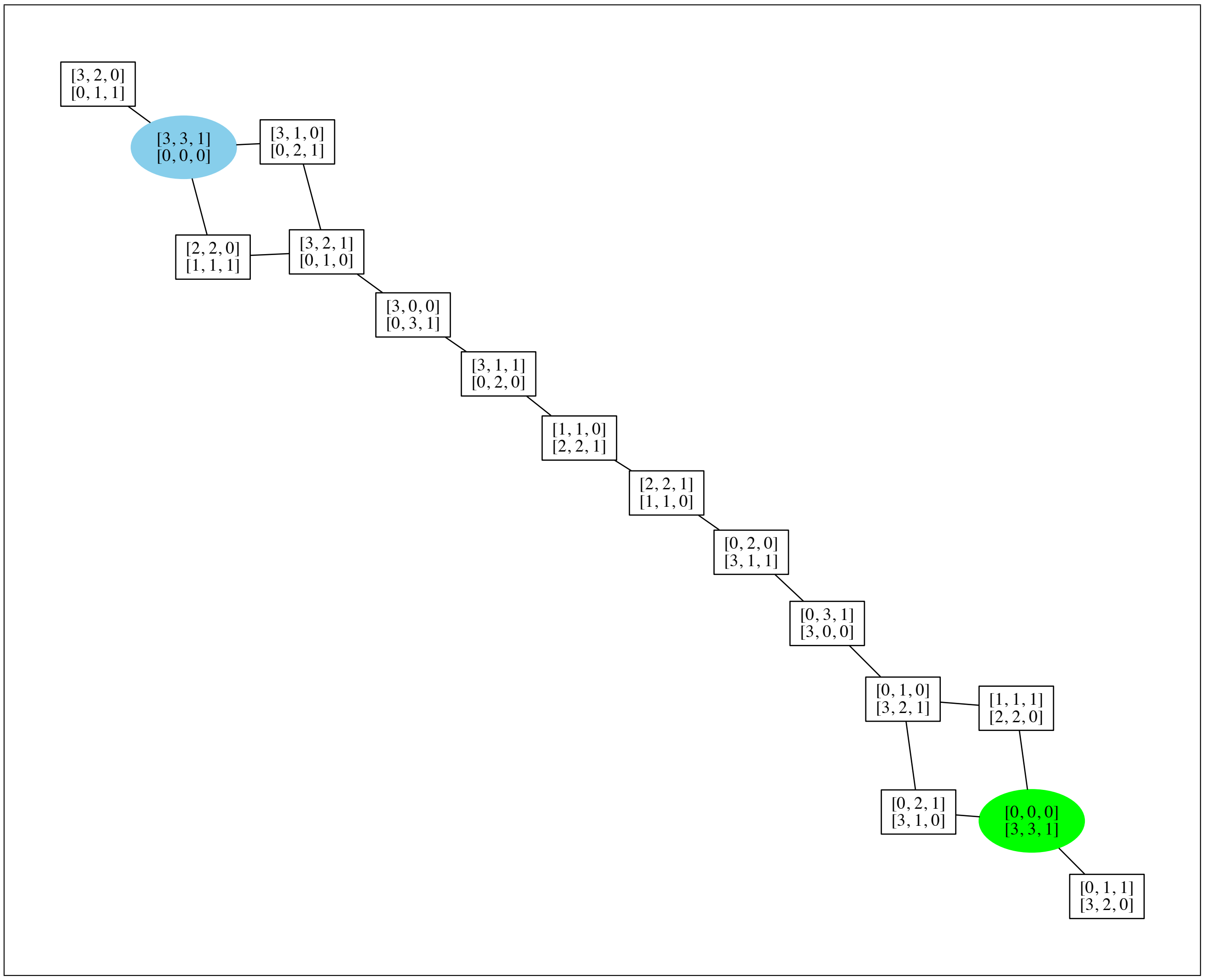

Figure 2.2: A graphical representation of the missionaries and cannibals problem.

## 2.1 The Sliding Puzzle

The $3 \times 3$ sliding puzzle is played on 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 $4 \times 4$ sliding puzzle is similar to the $3 \times 3$ sliding puzzle but it is played on a square board of length 4 instead. The $4 \times 4$ sliding puzzle is also known as the 15 puzzle.

In order to solve the $3 \times 3$ 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.

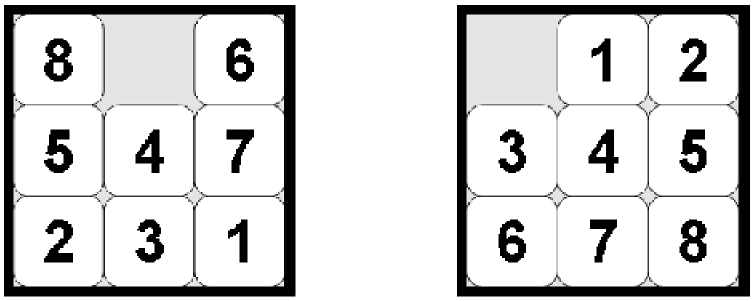

Figure 2.3: The $3 \times 3$ sliding puzzle.

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 $3 \times 3$ sliding puzzle, there are 9 squares that can be positioned in 9! different ways. It turns out that only half the positions are reachable from a given start state. Therefore, the effective number of states for the $3 \times 3$ sliding puzzle is

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

This is already a big number, but 181440 states can still be stored in a modern computer. However, the $4 \times 4$ 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 state. Hence we would need

$$16!/2 \cdots 8 = 83,691,159,552,000$$

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 $3 \times 3$ 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 $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ is moved into the position $\langle \texttt{row}, \texttt{col} \rangle$, while the tile at position $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ becomes empty.

3. Given a `state`, the procedure `newStates` computes the set of all states that can be reached in one step from `state`.

```
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 $3 \times 3$ sliding puzzle.

## 2.2 Breadth First Search

Informally, breadth first search 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`.

2. As long as the set `Frontier` does not contain the state `goal`, we extend this set by adding all states that can be reached in step from a state in `Frontier`.

In order to avoid loops, an implementation of breadth also 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

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

Figure 2.5 on page 11 shows an implementation of breadth first search in SetlX. We discuss this implementation line by line:

```
search := procedure(start, goal, nextStates) {
    Frontier := { start };
    Visited  := {}; // number of nodes expanded
    Parent   := {};
    while (Frontier != {}) {
        Visited += Frontier;
        NewFrontier := {};
        for (s in Frontier, ns in nextStates(s) | !(ns in Visited)) {
            NewFrontier += { ns };
            Parent[ns]  := s;
            if (ns == goal) {
                return pathTo(goal, Parent);
            }
        }
        Frontier := NewFrontier;
    }
};
```

Figure 2.5: Breadth first search.

1. `Frontier` is the set of all those states that have been encountered but whose neighbours have not yet been explored. Initially, it contains the state `start`.
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 11.
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`.

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

Figure 2.6: The procedure `pathTo`().

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 typical behaviour for breadth first search.

```
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 $3 \times 3$ 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 [2], breadth first search is often implemented using a queue data structure. Figure 2.8 on page 13 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 $3 \times 3$ sliding puzzle needs about 94 seconds, which is more than 10 times slower than our set based implementation that has been presented in Figure 2.5.

The solution of the $3 \times 3$ 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.

[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);
        }
        if (state in Visited) {
            continue;
        }
        Visited += { state };
        newStates := nextStates(state);
        for (ns in newStates | !(ns in Visited)) {
            Parent[ns] := state;
            Queue      += [ ns ];
        }
    }
};
```

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

2. The solution founnd 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. $\square$

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.

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

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

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

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

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

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

Figure 2.9: The first 24 steps in the solution of the $3 \times 3$ 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 15. The basic idea is to search a path up 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 ];
    Visited  := {}; // number of nodes expanded
    Parent   := {};
    while (Stack != []) {
        state := Stack[-1];
        Stack := Stack[..-2];
        if (state == goal) {
            return pathTo(state, Parent);
        }
        if (state in Visited) {
            continue;
        }
        Visited   += { state };
        newStates := nextStates(state);
        for (ns in newStates | !(ns in Visited)) {
            Parent[ns] := state;
            Stack      += [ns];
        }
    }
};
```

Figure 2.11: The depth first search algorithm.

When we test this idea with the 3 × 3 sliding puzzle, the solution is found in less than 2 seconds. This is more than four times faster than breadth first search. Furthermore, only 9569 states were explored. However, the solution that is found has a length of 9355 steps! As the shortest path from `start` to `goal` has 31 steps, the solution found by depth first search is highly redundant. If this redundancy is not an issue, depth first search is good choice as it is very easy to implement. For example, this is the case if we just want to know whether there is a path leading from `start` to `goal`. However, if we are interested in the path itself, then depth first search is simply not an option.

### 2.3.1 A Recursive Implementation of Depth First Search

Sometimes, the depth first search algorithm is presented as a recursive algorithm, since this leads to an implementation that is slightly shorter and more easy to understand. While this program works just fine for small problems like the missionaries and cannibals problem, it does not work at all for a problem of the size of the $3 \times 3$ sliding puzzle. The reason is that each recursive invocation of the function `dfs` needs to copy the parameters onto the stack. However, the parameters `Parent` and `Visited` grow linearly with the length of the path that is explored. We have already seen that the solution path that is found by depth first search has a length of more than 9000. The resulting overhead is prohibitive.

```
search := procedure(start, goal, nextStates) {
    return dfs(start, goal, nextStates, {}, {});
};
dfs := procedure(state, goal, nextStates, Parent, Visited) {
    if (state == goal) {
        return pathTo(goal, Parent);
    }
    Visited   += { state };
    newStates := nextStates(state);
    for (ns in newStates | !(ns in Visited)) {
        Parent[ns] := state;
        result     := dfs(ns, goal, nextStates, Parent, Visited);
        if (result != om) {
            return result;
        }
    }
};
```

Figure 2.12: A recursive implementation of depth first search.

## 2.4 Iterative Deepening

The fact that 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 2.13 on page 17 implements this strategy. We continue to discuss the details of this program.

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 depth first search is started. This process continues until either a solution is found or the sun rises in the west.

```
search := procedure(start, goal, nextStates) {
    limit := 1;
    while (true) {
        path := depthLimitedSearch(start, goal, nextStates, limit);
        if (path != om) {
            return path;
        }
        limit += 1;
    }
};
depthLimitedSearch := procedure(start, goal, nextStates, limit) {
    Stack    := [ start ];
    Distance := { [start, 0] };  // What is the distance to start?
    Parent   := {};
    while (Stack != []) {
        state := Stack[-1];
        Stack := Stack[..-2];
        if (state == goal) {
            return pathTo(state, Parent);
        }
        ds := Distance[state];            // ds:  distance state
        if (ds >= limit) {
            continue;
        }
        for (ns in nextStates(state)) {   // ns:  new state
            dns := Distance[ns];          // dns: distance new state
            if (dns != om && dns <= ds + 1) {
                continue;
            }
            Distance[ns] := ds + 1;
            Parent[ns]   := state;
            if (!(ns in Stack)) {
                Stack += [ns];
            }
        }
    }
};
```

Figure 2.13: Iterative deepening implemented in SetlX.

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 2.13 is stack based.

3. The stack is initialized to contain the state `start`.

4. For every state that is encountered in our search, we need to keep track of the distance of this state to the state `start`. This distance is stored in the dictionary `Distance`. Initially, only the distance of the node `start` is known. Of course, this node has a distance of 0 to the node `start`.

5. In contrast to the implementation of depth first search shown in Figure 2.11 on page 15, we do not keep track of the nodes that have been visited. Hence, there is no need for the variable `Visited`. The reason is that the depth limit already ensures that the function `dfs` does not loop. Furthermore, it actually might be necessary to revisit a given state: Suppose that there is a state $s$ and the current estimation of the

distance of $s$ from `start` is 5. Assume further that we now revisit the state $s$ on a path that would have a length of only 3. Finally, assume that we are using a depth `limit` of 6 and that there is a path of length 2 from the state $s$ to the state `goal`. If we would then discard further exploration of $s$ on the grounds that we have explored all its neighbours, we would then not be able to find the path of length 5 that connects `start` with `goal`!

6. Next, the first `state` is removed from the stack. If this `state` happens to be the `goal`, a path has been found and is returned.
7. Otherwise, we check the distance `ds` of `state`. If this distance is as big as the `limit`, then `state` can be discarded as we have already checked that it is not the goal.
8. Otherwise, the neighbours of `state` are computed. For every neighbour `ns` of `state`, we look up its distance `dns` from `start`. Now there are two cases.
   (a) If `dns` is less than or equal to `ds` $+ 1$, then there is no point in pursuing a path from `start` to `ns` that leads through `state`, as this path would have a length that is at least as long as the path from `start` to `ns` that has already been found.
   (b) If `dns` is either undefined or bigger than `ds` $+ 1$, then we have to update the distance of `ns`. Furthermore, if the state `ns` is not already present on the stack, we have to push it onto `Stack`.

   As in the original implementation of the depth first algorithm, this is process is iterated until the stack is exhausted.

When we run this program to solve the $3 \times 3$ sliding puzzle, the algorithm takes a little less than 5 minutes and visits $103,324$ states. 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 the 31 are essentially wasted.
2. When performing the computation for the limit of 30, all states that have a distance form start that is less than or equal to 30 have to be visited. This amounts to visiting $181,438$ states, because there are only two states that have a distance of 31 from `start`. Hence, the nearly the entire state space is visited.

**Exercise 1**: If there is no solution, the implementation of iterative deepening that is shown in Figure 2.13 does not terminate. The reason is that the function `dfs` does not distinguish whether it fails to find a solution because the depth limit is reached or because the `Stack` is exhausted. Improve the implementation so that it will always terminate 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. As the excessive length of the stack was the main culprit for the weak performance of our recursive implementation of depth first search, we can hope that a recursive implementation of iterative deepening is less disappointing than our recursive implementation of depth first has been. Figure 2.14 on page 19 shows a recursive implementation of iterative deepening. This implementation has several nice features:

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 considerably bigger than the running time of the stack base implementation: On my computer, the recursive implementation takes about 36 minutes!

```
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 2.14: A recursive implementation of iterative deepening.

## 2.5 Bidirectional Breadth First Search

The way breadth first search works it 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 generated. 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

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

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 will both have a size of approximately $d/2$. If this is the case, only

$$2 \cdot b^{d/2}$$

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

$$b^{d+1}$$

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

$$2^{40} = 1,099,511,627,776$$

in breadth first search, while we only have to explore

$2^{40/2} = 1,048,576$

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 average devices.

```
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 2.15: Bidirectional breadth first search.

Figure 2.15 on page 20 shows the implementation of bidirectional breadth first search. Essentially, we have to keep 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 2.16 on page 21.

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 $3 \times 3$ sliding puzzle in less than a second! However, bidirectional breadth first search is still not able to solve the $4 \times 4$ sliding puzzle since the portion of the search space that needs to be computed is still too big.

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

Figure 2.16: Combining two paths.

## 2.6 The A* Search Algorithm

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

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

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

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

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

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

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:

$$\begin{array}{lcll} h(s_{k+1}) & \leq & 1 + h(s_k) & \text{because } h \text{ is consistent} \\ & \leq & 1 + k & \text{because } h(s_k) \leq k \text{ by i.h.} \end{array}$$

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 9 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.

   We will later see that 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 coordinates 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 only allowed to move the tile horizontally or vertically. 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:

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

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 $3 \times 3$ 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 2.17 on page 23 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 $\langle \texttt{rowA}, \texttt{colA}\rangle$ in `stateA` is computed using the expression `stateA[rowA][colA]` and the corresponding position $\langle \texttt{rowB}, \texttt{colB}\rangle$ of this tile in state `stateB` is computed using the function `findTile`.

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

```
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 2.17: The Manhattan distance between two states.

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 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 describe how the A* algorithm uses its heuristic. 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 2.18 on page 24 and discussed below.

```
aStarSearch := procedure(start, goal, nextStates, heuristic) {
    Parent   := {};                     // back pointers, represented as dictionary
    Distance := { [start, 0] };
    estGoal  := heuristic(start, goal);
    Estimate := { [start, estGoal] };  // Estimated distance
    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 2.18: The A* search algorithm.

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. `next` is a function that takes a state as a parameter. For a state $s$,

   `next`$(s)$

   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

   `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`.

2. `Distance` is a dictionary that remembers for every state $s$ that is encountered during the search 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`.

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

$$[\texttt{stateEstimate}, \texttt{state}]$$

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 and is returned.

3. Otherwise, we check the length of 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. `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 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 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`.

It can be shown that the A* search algorithm is complete and that the computed solution is optimal.

When we run A* on the $3 \times 3$ sliding puzzle, it takes about 17 seconds to solve the instance shown in Figure 2.3 on page 9. 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.

## 2.7 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 2.19 on page 27 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.18 on page 24 as the algorithm for bidirectional search shown in Figure 2.15 on page 20 relates to breadth first search shown in Figure 2.5 on page 11. Hence, we will not discuss the details any further.

When we run bidirectional A* search for the $3 \times 3$ sliding puzzle shown in Figure 2.3 on page 9, the program takes 2 second but only uses $2,963$ states. Therefore, I have tried to solve the $4 \times 4$ sliding puzzle shown in Figure 2.20 on page 28 using bidirectional A* search. A solution of 44 steps was found in 65 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.

## 2.8 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*. Figure 2.21 on page 28 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

       `heuristic`$(s_1, s_2)$

       computes an estimate of the distance between $s_1$ and $s_2$.

2. The function `idaStarSearch` initialises `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`.

```
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 2.19: Bidirectional A* 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.

```
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 2.20: A start state and a goal state for the $4 \times 4$ sliding puzzle.

```
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 := 1.0 / 0.0;  // 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;
        }
        if (result < smallest) {
            smallest := result;
        }
    }
    return smallest;
};
```

Figure 2.21: Iterative deepening $A^*$ search.

We proceed to discuss the function `search`. This function takes 7 parameters, which we describe next.

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

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

   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 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 $3 \times 3$ sliding puzzle shown on Figure 2.3 on page 9, this algorithm takes about 2.6 seconds to solve the puzzle. For the $4 \times 4$ 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 does is not sufficient for the bidirectional A* algorithm.

## 2.9 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* does take longer, its memory consumption is much lower than the memory consumption of bidirectional A*. Hence, it is natural to try to combine these two algorithms. Concretely, the idea is to run an A* search from the start node until memory is more or less exhausted. Then, we start IDA* form the goal node and search until we find any of the nodes discovered by the A* search that has been started from the start node.

```
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 2.22: The A* – IDA* search algorithm, part I.

An implementation of the A*-IDA* algorithm is shown in Figure 2.22 on page 30 and Figure 2.23 on page 32. 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.

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, `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 and 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. However, $P$ does not include the node $s$. Hence, 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 returned.

Iterative deepening A*-IDA* is a complete search algorithm. On the instance of the $3 \times 3$ sliding puzzle shown on Figure 2.3 on page 9, this algorithm takes about 1.4 seconds to solve the puzzle. For the $4 \times 4$ sliding puzzle, if the algorithm is allowed to visit at most 3 000 states, the algorithm takes less than 9 seconds. However, there is one caveat: A*-IDA* search is not *guaranteed* to find an optimal path, although in practise it often does.

```
deepeningSearch := procedure(g, s, nextStates, heuristic, Distance) {
    limit := 0;
    while (true) {
        Path := search(g, s, nextStates, 0, limit, heuristic, [g], Distance);
        if (isList(Path)) {
            return Path;
        }
        limit := Path;
    }
};
search := procedure(g, s, nextStates, d, l, heuristic, Path, Dist) {
    total := d + heuristic(g, s);
    if (total > l) {
        return total;
    }
    if (Dist[g] != om) {
        return [g, Path[2..]];
    }
    smallest := 1.0 / 0.0;  // infinity
    for (ns in nextStates(g) | !(ns in Path)) {
        result := search(ns, s, nextStates, d+1, l, heuristic, [ns]+Path, Dist);
        if (isList(result)) {
            return result;
        }
        if (result < smallest) {
            smallest := result;
        }
    }
    return smallest;
};
```

Figure 2.23: The $A^* - IDA^*$ search algorithm, part II.

# Chapter 3

# Constraint Satisfaction

In this chapter we discuss *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* for the variables in `Vars`.
3. `Constraints` is a set of mathematical formulae. Each of these formulae is called a *constraint* of $\mathcal{P}$.

Given a constraint satisfaction problem $\mathcal{P} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$, a *variable assignment* for $\mathcal{P}$ is a function

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

A variable assignment $A$ is a *solution* of the constraint satisfaction problem $\mathcal{P}$ if, given the assignment $A$, all constraints of $\mathcal{P}$ are satisfied. We proceed to illustrate the definitions given so far with two examples.

### 3.0.1 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 34 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:

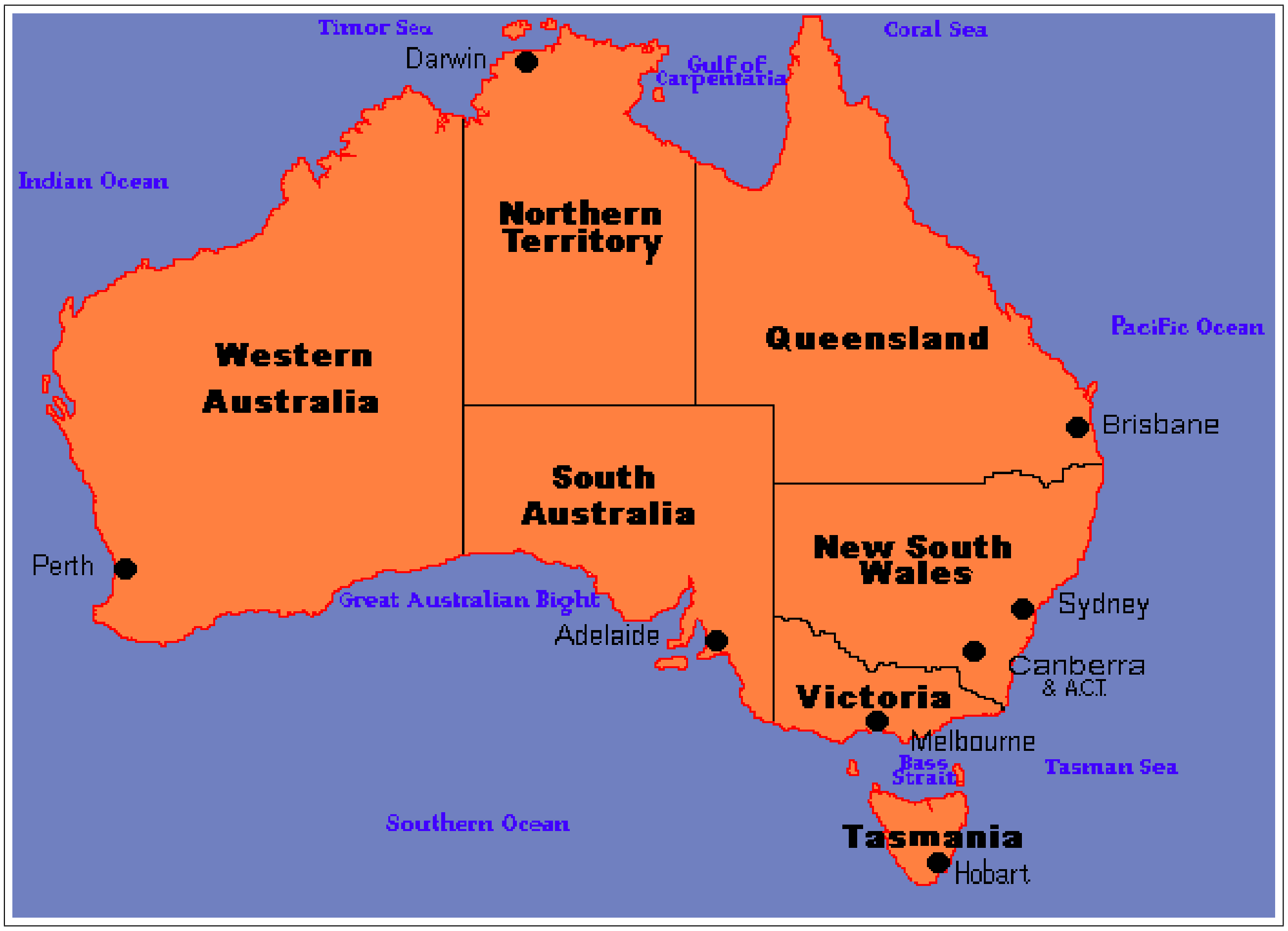

Figure 3.1: A map of Australia.

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. $\texttt{Constraints} := \big\{\text{WT} \neq \text{NT}, \text{WT} \neq \text{SA}, \text{NT} \neq \text{SA}, \text{NT} \neq \text{Q}, \text{SA} \neq \text{Q}, \text{SA} \neq \text{NSW}, \text{SA} \neq \text{V}, \text{V} \neq \text{T}\big\}$

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 $\mathcal{P}$.

## 3.0.2 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 `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 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

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

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 36.

Figure 3.3 on page 36 shows a SetlX program that can be used to solve the eight queens puzzle. The code shown in this figure is more general than the eight queens puzzle: Given a natural number $n$, the function call $\texttt{queensCSP}(n)$ 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 $n$.

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 constraint satisfaction problem like the one produced by the program shown in Figure 3.3 and that is then capable of computing a solution.

## 3.0.3 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

```
---------------------------------
|   |   |   |   |   |   | Q |   |
---------------------------------
|   |   |   | Q |   |   |   |   |
---------------------------------
|   | Q |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   | Q |
---------------------------------
|   |   |   |   |   | Q |   |   |
---------------------------------
| Q |   |   |   |   |   |   |   |
---------------------------------
|   |   | Q |   |   |   |   |   |
---------------------------------
|   |   |   |   | Q |   |   |   |
---------------------------------
```

Figure 3.2: A solution of the *eight queens problem*.

```
queensCSP := procedure(n) {
    Variables   := { "V$i$" : i in {1..n} };
    Values      := { 1 .. n };
    Constraints := {};
    for (i in [1..n], j in [1..i-1]) {
        Constraints += { "V$i$ != V$j$" };
        Constraints += { "$i$ + V$i$ != $j$ + V$j$" };
        Constraints += { "$i$ - V$i$ != $j$ - V$j$" };
    }
    return [Variables, Values, Constraints];
};
```

Figure 3.3: SetlX code to create the CSP representing the eight-queens puzzle.

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.

## 3.1 Backtracking Search

## 3.2 Non-Chronological Backtracking

## 3.3 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

$$\mathcal{C} := \langle V, C, D\rangle,$$

local search works as follows:

1. Initialize the values of the variables in $V$ randomly.
2. If all constraints are satisfied, return the solution.
3. For every variable $x \in V$, count the number of unsatisfied constraints that involve the variable $x$.
4. Set `maxNum` to be the biggest number of unsatisfied constraints for a single variable.
5. Compute the set `maxVars` of those variables that have `maxNum` unsatisfied constraints.
6. Randomly choose a variable $x$ from the set `maxVars`.
7. Find a value $d \in D$ such that by assigning $d$ to the variable $x$, the number of unsatisfied constraints 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 a solution is found.

```
 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 3.4: Solving the $n$ queens problem using local search.

Figure 3.4 on page 37 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`, hich 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 $\texttt{r} \in \{1, \cdots, \texttt{n}\}$, 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 $\{1, \cdots, \texttt{n}\}$. 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 enclsing 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.

The procedure `numConficts` shown in Figure 3.5 on page 39 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 3.5.

Using the program discussed in this section, the n queens problem can be solved for a $\texttt{n} = 1000$ 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.

```
numConflicts := procedure(Queens, row) {
    n      := #Queens;
    result := 0;
    for (r in {1 .. n} | r != row) {
        if ( Queens[r] == Queens[row]           ||
             r - Queens[r] == row - Queens[row] ||
             r + Queens[r] == row + Queens[row]
           )
        { result += 1; }
    }
    return result;
};
```

Figure 3.5: The procedure `numConficts`.

# Chapter 4

# Planning

**Definition 4 (State Transition System)** A deterministic *state transition system* is a triple $\Sigma = \langle S, A, \gamma \rangle$ such that:

- $S$ is the set of *states*,
- $A$ is the set of *actions*, and
- $\gamma : S \times A \to S$ is the state transition function. $\diamond$

Executing an action $a \in A$ in a state $s_1$ yields a new state $s_2 = \gamma(s_1)$. The function $\gamma$ is extended to a function $\gamma^*$ that takes lists of actions as its second argument. This definition is given inductively.

1. $\gamma^*(s, []) := s$,
2. $\gamma^*(s, [a] + r) := \gamma^*\bigl(\gamma(s, a), r\bigr)$.

In order to simplify the notation we do not distinguish between $\gamma^*$ and $\gamma$, i.e. we write $\gamma(s, l)$ instead of $\gamma^*(s, l)$.

**Definition 5 (Planning Problem)** A *planning problem* is a triple $\mathcal{P} = \langle \Sigma, s_0, g \rangle$ such that:

1. $\Sigma = \langle S, A, \gamma \rangle$ is a state transition system,
2. $s_0 \in S$ is the *initial state*, and
3. $g \in S$ is the *goal* state.

A search problem is more abstract than a planning problem: In a search problem, states are connected in a way described by a relation $R$ that tells us whether two states are connected. We can go from state $s_1$ to state $s_2$ iff $\langle s_1, s_2 \rangle \in R$. In a planning problem, we additionally have actions that tell us exactly what causes the state $s_1$ to be transformed into the state $s_2$.

There are various ways to represent planning problems. The three most common ones are as follows:

1. The *set-theoretic* representation uses sets of propositional variables to represent a state.
2. The *classical* representation is based on first order logic.
3. The *state-variable* representation represents states by specifying the values that certain variables take.

In this lecture, we will only have time to discuss the classical representation of the planning problem. The other representations are discussed in [1].

## 4.1 The Classical Representation of a Planning Problem

A *first-order language* $\mathcal{L}$ is triple

$$\mathcal{L} = \langle P, C, V \rangle, \quad \text{where}$$

1. $P$ is the set of *predicate symbols*,
2. $C$ is the set of *constant symbols*, and
3. $V$ is the set of *variable symbols.*

In the classical representation of a planning problem, a state is a set of *ground atoms* of the language $\mathcal{L}$. Here, a *ground atom* is an atomic formula that does not contain variables. An atomic formula $p$ is true is a state $s$ iff $p \in s$. Goals will be represented by sets of literals. A goal $g$ is *satisfied* in a state $s$ iff there is a variable substitution $\sigma$ such that every positive literal of $g\sigma$ is in $s$, while no negative literal of $g\sigma$ is in $s$. Here, we have used the so called *closed world assumption*: If $s$ is a state, i.e. a set of ground atoms, then every ground atom that is not an element of $s$ is assumed to be false.

# Chapter 5

# Markov Decision Processes

*Markov Decision Processes* are a means to deal with uncertainty when searching.

**Definition 6 (Markov Decision Process)** A *Markov decision process* (abbreviated as MDP) is a 5-tuple

$$M := \langle S, A, T, R, s_0, f \rangle$$

where

1. $S$ is the set of states.
2. $A$ is the set of actions.
3. $T$ is the transition probability. For all states $s_1, s_2 \in S$ and every action $a \in A$,
   $$T(s_1, a, s_2)$$
   is the probability that, provided the system is in state $s_1$ and action $a$ is executed, the system will be in state $s_2$ afterwords. Formally, we have
   $$T : S \times A \times S \to [0, 1].$$
4. $R$ is the reward function. For all states $s_1, s_2 \in S$ and every action $a \in A$,
   $$R(s_1, a, s_2)$$
   is the reward that the system reaps if the system starts in state $s_1$, executes the action $a$ and thereby reaches the state $s_2$. Formally, we have
   $$R : S \times A \times S \to \mathbb{R}.$$
5. $f$ is the final state. Therefore, we have $f \in S$.

The goal of a Markov decision process is to reach the final state and earn the highest possible reward while doing so. Therefore, a Markov decision process formalizes search under uncertainty. ◇

**Definition 7 (Policy)** Given a Markov decision process $\langle S, A, T, R, s_0, f \rangle$, a policy $\pi$ is a function

$$\pi : S \to A$$

that takes a state and computes an action. A policy is optimal if it maximizes the expected utility of an agent following it.

**Definition 8 (Transition)** Given a Markov decision process $\langle S, A, T, R, s_0, f \rangle$, a *transition*

$$\tau = \langle s_1, a, s_2 \rangle$$

is a triple consisting of a state $s_1$, an action $a$, and a state $s_2$. The interpretation of this transition is that in state $s_1$ the agent executes the action $a$ and this results in the agent being in state $s_2$. ◇

## 5.1 The Bellmann Equation

The Bellmann equation is a fixpoint equation that determines the *value of a state*, where the value of a state is the discounted payoff of the expected reward that is expected to be achieved from the given state if the agent always performs the action that maximizes his expected rewards. If the value of a state $s$ is denoted as $V^*(s)$, then $V^*(s)$ satisfies the equation

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \cdot \bigl(R(s,a,s') + \gamma \cdot V^*(s')\bigr),$$

where $\gamma$ is the discount factor satisfying $0 < \gamma \leq 1$. If $\gamma = 1$, then there is no discounting of future values. If $\gamma < 1$, the Bellmann equation can be solved by fixed point iteration. In this case, for every state $s$ we define a sequence of values $\bigl(V_k(s)\bigr)_{s \in S}$ as follows:

1. $V_0(s) := 0$ for all $s \in S$,
2. $V_{k+1}(s) := \max_{a \in A} \sum_{s' \in S} T(s,a,s') \cdot \bigl(R(s,a,s') + \gamma \cdot V_k^*(s')\bigr)$ for all $s \in S$ and all $k \in \mathbb{N}$.

It can be shown that the sequence $\bigl(V_k(s)\bigr)_{s \in S}$ converges provided $\gamma < 1$. This process is known as *value iteration.*

## 5.2 Q-Value Iteration

Instead finding the values $V^*(s)$ of a given state $s$ we instead try to compute the value a state $s$ has once an action $a$ for that state has been fixed. These values are calles Q-values and they are denoted as $Q(s,a)$. The Bellmann equation for Q-values is

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s') \cdot \bigl(R(s,a,s') + \gamma \cdot \max_{a' \in A} Q^*(s',a')\bigr),$$

This equation can also be solved via a fixpoint iteration as follows:

1. $Q_0(s,a) := 0$ for all $s \in S$ and $a \in A$,
2. $Q_{k+1}(s,a) := \sum_{s' \in S} T(s,a,s') \cdot \bigl(R(s,a,s') + \gamma \cdot \max_{a' \in A} Q_k^*(s',a')\bigr)$
   for all $s \in S$, $a \in A$ and all $k \in \mathbb{N}$.

## 5.3 Q-Learning

Q-value iteration can only be used if the transition probability $T(s,a,s')$ and the reward $R(s,a,s')$ is given. In Q-learning, we have an unknown Markov decision process and attempt to learn the optimal strategy. In this case, we update our estimates of the Q-values via the equation

$$Q_{k+1}(s,a) := (1-\alpha) \cdot Q_k(s) + \alpha \cdot \bigl(R(s,a,s') + \gamma \cdot \max_{a' \in A} Q_k^*(s',a')\bigr).$$

Here, $s'$ is the state that has actually been reached the last time we executed the action $a$ in state $s$. The variable $\alpha$ is the *learning rate* and we have $0 < \alpha < 1$. If $\alpha$ close to 1, the learning rate is high and the past is unimportant. Usually, $\alpha$ is decreased over time because, initially, the values of $Q_k(s,a)$ are mostly wrong but as we keep learning these values get better. Ideally, the learning rate should be slowly decreased over time.

## 5.4 Approximate Q-Learning

Compute the Q-values via a number $m$ different features in a linear way as follows:

$$Q(s,a) = \sum_{k=1}^{m} w_k \cdot f_k(s,a).$$

Here, the numbers $w_k$ are the weights while the functions $f_k$ are the features. Update the weights using the equation

$$w_k := w_k + \alpha \cdot \bigl(r + \gamma \cdot \max_{a'} Q(s'.a') - Q(s,a)\bigr) \cdot f_k(s,a).$$

Here, $r$ is the reward when taking the action $a$ in state $s$ and $s'$ is the resulting state.

# Chapter 6

# Boolean Satisfiability

## 6.1 Chaff

1. *Implication*: Propositional variable forced to be true by the current assignment of the propositional variables.
2. *Decision*: Assignment of setting a variable to either `true` or `false`.
3. *Decision Stack*: Stack of all decisions.
4. *Decision Level*: Integer tag identifying the last *decision*.
5. *Decision Assignment*: Which variable is assigned next, and which value do we pick?
6. *implied clause*: A clause is implied iff all but one of the literals of the clause have a value of `true` given the current variable assignment.

### 6.1.1 Watch Lists

In a typical DP SAT solver, 90% of the computation time is spend in Boolean constraint propagation. Therefore, it is essential to get Boolean constraint propagation fast.

In every clause, two literals are *watched*. If any of these two literals evaluates to `true` because of a variable assignment, we do not need to watch the clause any more, as it evaluates as `true`. If one of these variables is assigned `false`, there are two cases:

1. If all the unwatched literals in the variable have been assigned `false`, the remaining watched literal is assigned `true`.
2. If at least one of the unwatched literals in the variable have been assigned `true`, the clause is `true` and does not need to be watched anymore.
3. If there is an unwatched literal that is not yet assigned, we watch this literal instead of the literal that evaluates as `false` now.

The choice of which variables to watch is not important. If a variable is unassigned, we do not need to change the watch lists.

Data Structure: A map that maps propositional variables to lists of clauses.

### 6.1.2 Decaying Sum Heuristic

1. Each literal has a counter that is initialized to 0.
2. When a clause is added to the database, the counter of each literal occurring in the clause is incremented.

3. Choose the unassigned variable with the highest count.
4. Break ties randomly.
5. Periodically, all counters are divided by a constant. (This is the decaying process.)

### 6.1.3 Restarts

# Chapter 7

# Linear Regression

## 7.1 Simple Linear Regression

Assume we have a *linear hypothesis*

$$h_\theta(x) := \theta_1 \cdot x + \theta_0$$

We have $m$ observations $\langle x^{(1)}, y^{(1)}\rangle, \cdots, \langle x^{(m)}, y^{(m)}\rangle$. Our goal is to minimize the *squared error*

$$E(\theta) = \sum_{i=1}^{m} h_\theta\bigl(x^{(i)} - y^{(i)}\bigr)^2$$

## 7.2 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

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

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

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

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

$$E := \frac{1}{n} \cdot \sum_{i=1}^{n} \Bigl(F\bigl(\mathbf{x}^{(i)}\bigr) - y^{(i)}\Bigr)^2. \tag{7.1}$$

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 (7.1) can be rewritten as follows:

$$E(\mathbf{w}) = \frac{1}{n} \cdot \sum_{i=1}^{n} \Big( (\mathbf{x}^{(i)})^T \cdot \mathbf{w} - y^{(i)} \Big)^2. \tag{7.2}$$

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:

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

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

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

Taking the square of the vector $X \cdot \mathbf{w} - \mathbf{y}$ we discover that we can rewrite equation (7.2) as follows:

$$E(\mathbf{w}) = \frac{1}{n} \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big)^T \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big). \tag{7.3}$$

## 7.3 Some Useful Gradients

In the last section, we have computed the squared error $E(\mathbf{w})$ using equation (7.3). 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.

### 7.3.1 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 2**: Prove this equation.

## 7.4 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 (7.3):

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

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

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

Hence we have shown that

$$E(\mathbf{w}) = \frac{1}{n} \cdot \Bigl(\mathbf{w}^T \cdot \bigl(X^T \cdot X\bigr) \cdot \mathbf{w} - 2 \cdot \mathbf{y}^T \cdot X \cdot \mathbf{w} + \mathbf{y}^T \cdot \mathbf{y}\Bigr) \tag{7.4}$$

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

$$\bigl(X^T \cdot X\bigr)^T = X^T \cdot \bigl(X^T\bigr)^T = X^T \cdot X.$$

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

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

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

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

This equation can be rewritten as

$$\bigl(X^T \cdot X\bigr) \cdot \mathbf{w} = X^T \cdot \mathbf{y}. \tag{7.5}$$

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

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

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.

# Chapter 8

# Classification

One of the earliest application of artificial intelligence is *classification*. A good example of classification is spam detection. A system for spam detection classifies an email as either spam or not spam. To do so, it first computes various *features* of the email and then uses these features to determine whether the email is likely to be spam. For example, a possible feature would be the number of occurrences of the word "`pharmacy`" in the text of the email.

## 8.1 Introduction

Formally, the classification problem in machine learning can be stated a s follows. We are given a set of objects $S := \{o_1, \cdots, o_n\}$ and a set of classes $C := \{c_1, \cdots, c_k\}$. Furthermore, there exists a function

$$\texttt{classify} : S \to C$$

that assigns a class $\texttt{classify}(o)$ to every object $o \in S$. The set $S$ is called the *sample space*. In the example of spam detection, $S$ is the set of all emails that we might receive, i.e. $S$ is the set of all strings of the Ascii alphabet, while

$$C = \{\texttt{spam}, \texttt{ham}\}.$$

If $\texttt{classify}(o) = \texttt{spam}$, we consider the email $o$ to be spam. Our goal is to compute the function `classify`. In order to do this, we use an approach known as supervised learning: We take a subset $S_{\mathit{Train}} \subseteq S$ of emails where we already know whether the emails are spam or not. This set $S_{\mathit{Train}}$ is called the *training set*. Next, we define a set of $D$ *features* for every $o \in S$. These features have to be computable, i.e. we must have a function

$$\texttt{feature} : S \times \{1, \cdots, D\} \to \mathbb{R}$$

such that $\texttt{feature}(o, j)$ computes the $j$-th feature and we have to be able to implement this function with reasonable efficiency. In general, the values of the features are real values. However, there are cases where these values are just Booleans. If

$$\texttt{feature}(o, j) \in \mathbb{B} \text{ for all } o \in S,$$

then the $j$-th feature is a *binary feature*. For example, in the case of spam detection, the first feature could be the occurrence of the string "`pharmacy`". In this case, we would have

$$\texttt{feature}(o, 1) := (\texttt{pharmacy} \in o),$$

i.e. the first feature would be to check whether the email $o$ contains the string "`pharmacy`".

If we want to be more precise, we can instead define the first feature as

$$\texttt{feature}(o, 1) := \texttt{count}(\texttt{"pharmacy"}, o),$$

i.e. we would count the number of occurrences of the string "`pharmacy`" in our email $o$. As $\texttt{count}(\texttt{"pharmacy"}, o)$ is always a natural number, in this case the first feature would be a *discrete* feature. However, we can be even more precise than just counting the number of occurrences of "`pharmacy`". After all, there is a difference if the string “`pharmacy`” occurs once in an email containing but a hundred characters or whether is occurs once in an email with a length of several thousand characters. To this end, we would define the first feature as

$$\texttt{feature}(o, 1) := \frac{\texttt{count("pharmacy"}, o)}{\texttt{\#}o},$$

where $\texttt{\#}o$ defines the number of characters in the string $o$. In this case, the first feature would be a *continuous* feature and as this is the most general case, unless stated otherwise, we deal with the continuous case.

Having defined the features, we next need a *model* of the function `classify` that tries to approximate the function `classify` via the features. This model is given by a function

$$\texttt{model} : \mathbb{R}^D \to C$$

such that

$$\texttt{model}\bigl(\texttt{feature}(o, 1), \cdots, \texttt{feature}(o, D)\bigr) \approx \texttt{classify}(o).$$

Using the function `model`, we can than approximate the function classify using a function `guess` that is defined as

$$\texttt{guess}(o) := \texttt{model}\bigl(\texttt{feature}(o, 1), \cdots, \texttt{feature}(o, D)\bigr)$$

Most of the time, the function `guess` will only approximate the function `classify`, i.e. we will have

$$\texttt{guess}(o) = \texttt{classify}(o)$$

for most objects of $o \in S$ but not for all values. The *accuracy* of our model is then defined as the fraction of those objects that are classified correctly, i.e.

$$\texttt{accuracy} := \frac{\texttt{\#}\{o \in S \mid \texttt{guess}(o) = \texttt{classify}(o)\}}{\texttt{\#}S}.$$

If the set $S$ is infinite, this equation has to be interpreted as a limit, i.e. we can define $S_n$ as the set of strings that have a length of at most $n$. Then, the accuracy can be defined as follows:

$$\texttt{accuracy} := \lim_{n\to\infty} \frac{\texttt{\#}\{o \in S_n \mid \texttt{guess}(o) = \texttt{classify}(o)\}}{\texttt{\#}S_n}.$$

The function `model` is usually determined by a set of *parameters* or *weights* $\mathbf{w}$. In this case, we have

$$\texttt{model}(\mathbf{x}) = \texttt{model}(\mathbf{x}; \mathbf{w})$$

where $\mathbf{x}$ is the vector of features, while $\mathbf{w}$ is the vector of weights. We will assume that the number of weights is the same as the number of features. When it comes to the choice of the model, it is important to understand that, at least in practical applications, <u>all</u> models are wrong. Nevertheless, <u>some</u> models are useful. There are two reasons for this:

1. We do not fully understand the function `classify` that we want to approximate by the function `model`.
2. The function `classify` is so complex, that even if we could compute it exactly, the resulting model would be much too complicated.

The situation is similar in physics: Let us assume that we intend to model the fall of an object. A model that is a hundred percent accurate would have to include not only air friction, but also the possible effects of tidal forces or, in case we have a metallic object, the effects of the magnetic field of the earth have to be taken into account. On top of that we need some corrections from relativistic physics and even quantum physics. A model of this kind would be so complicated that it would be useless.

Let us summarize our introductory discussion of machine learning in general and classification in particular. A set $S$ of objects and a set $C$ of classes are given. Our goal is to approximate a function

$$\texttt{classify} : S \to C$$

using certain *features* of our objects. The function `classify` is then approximated using a function `model` as follows:

$$\texttt{model}\bigl(\texttt{feature}(o, 1), \cdots, \texttt{feature}(o, D); \mathbf{w}\bigr) \approx \texttt{classify}(o).$$

The model depends on a vector of parameters $\mathbf{w}$. In order to *learn* these parameters, we are given a *training set* $S_{\mathit{Train}}$ that is a subset of $S$. As we are dealing with *supervised learning*, the function classify is known for all objects $o \in S_{\mathit{Train}}$. Our goal is to determine the parameters $\mathbf{w}$ such that the number of mistakes we make on the training set is minimized.

### 8.1.1 Notation

We conclude this introductory section by fixing some notation. Let us assume that the objects $o \in S_{\mathit{Train}}$ are numbered from 1 to $N$, while the features are numbered from 1 to $D$. Then we define

1. $\mathbf{x}_i := [\texttt{feature}(o_i, 1), \cdots, \texttt{feature}(o_i, D)] \quad$ for all $i \in \{1, \cdots, N\}$.

   i.e. $\mathbf{x}_i$ is a $D$-dimensional vector that collects the features of the $i$-th training object.
2. $x_{i,j} := \texttt{feature}(o_i, j) \quad$ for all $i \in \{1, \cdots, N\}$ and $j \in \{1, \cdots, D\}$.

   i.e. $x_{i,j}$ is the $j$-th feature of the $i$-th object.
3. $y_i := \texttt{classify}(o_i) \quad$ for all $i \in \{1, \cdots, N\}$

   i.e. $y_i$ is the class of the $i$-th object.

Mathematically, our goal is now to maximize the accuracy of our model as a function of the parameters $\mathbf{w}$.

### 8.1.2 Applications of Classification

Besides spam detection, there are many other classification problems that can be solved using machine learning. To give just one more example, imagine a general practitioner that receives a patient and examines his symptoms. In this case, the symptoms can be seen as the features of the patient. For example, these features could be

1. body temperature,
2. blood pressure,
3. heart rate,
4. body weight,
5. breathing difficulties,
6. age,

to name but a few of the possible features. Based on these symptoms, the general practitioner would then decide on an illness, i.e. the set of classes for the classification problem would be

$$\{\texttt{commonCold}, \texttt{pneumonia}, \texttt{asthma}, \texttt{flu}, \cdots, \texttt{unknown}\}.$$

Hence, the task of disease diagnosis is a classification problem. This was one of the earliest problem that was tackled by artificial intelligence. As of today, computer-aided diagnosis has been used for more than 40 years in many hospitals.

## 8.2 Digression: The Method of Gradient Ascent

In machine learning, it is often the case that we have to find either the maximum or the minimum of a function

$$f : \mathbb{R}^n \to \mathbb{R}.$$

For example, when we discuss *logistic regression* in the next section, we will have to find the maximum of a function. First, let us introduce the argmax function. The idea is that

$$\widehat{\mathbf{x}} = \underset{\mathbf{x} \in \mathbb{R}}{\operatorname{argmax}} f(\mathbf{x})$$

is that value of $\mathbf{x} \in \mathbb{R}$ that maximizes $f(\mathbf{x})$. Formally, we have

$$\forall \mathbf{x} \in \mathbb{R} : f(\mathbf{x}) \leq f\Bigl(\operatorname*{argmax}_{\mathbf{x}\in\mathbb{R}} f(\mathbf{x})\Bigr).$$

Of course, the function argmax is only defined when the maximum is unique. If the function $f$ is differentiable, we know that a necessary condition for a vector $\widehat{\mathbf{x}} \in \mathbb{R}^n$ to satisfy

$$\widehat{\mathbf{x}} = \arg\max_{\mathbf{x}\in\mathbb{R}} f(\mathbf{x}) \quad \text{is that we must have} \quad \nabla f(\widehat{\mathbf{x}}) = \mathbf{0},$$

i.e. the *gradient* of $f$, which we will write as $\nabla f$, vanishes at the maximum. Remember that the gradient of $f$ is defined as

$$\nabla f := \begin{pmatrix} \dfrac{\partial f}{\partial x_1} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{pmatrix}.$$

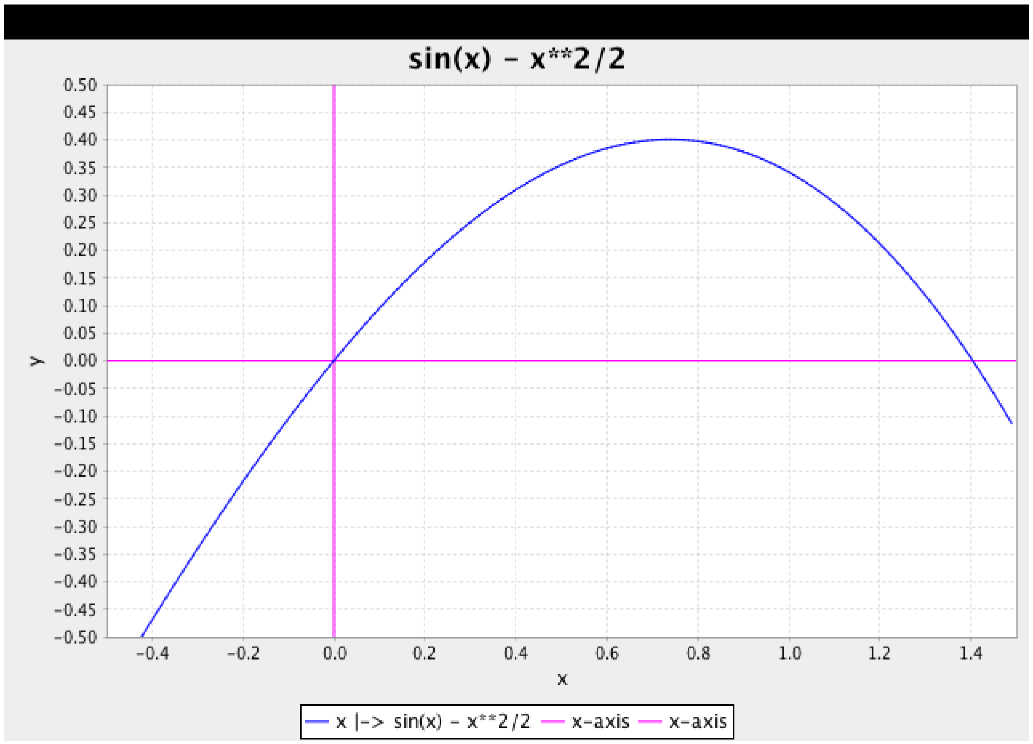

Figure 8.1: The function $x \mapsto \sin(x) - \frac{1}{2} \cdot x^2$.

Unfortunately, in many cases the equation

$$\nabla f(\widehat{\mathbf{x}}) = \mathbf{0}$$

can not be solved explicitly. This is already true in the one-dimensional case, i.e. if $n = 1$. For example, consider the function $f : \mathbb{R} \to \mathbb{R}$ that is defined as

$$f(x) := \sin(x) - \frac{1}{2} \cdot x^2.$$

This function is shown in Figure 8.1 on page 55. It is obvious that this function has a maximum somewhere between 0.6 and 0.8. In order to compute this maximum, we can compute the derivative of $f$. This derivative is given as

$$f'(x) = \cos(x) - x$$

As it happens, the equation $\cos(x) - x = 0$ does not seem to have a solution in closed terms. We can only approximate the solution numerically.

The method of *gradient ascent* is a numerical method that can be used to find the maximum of a function

$$f : \mathbb{R}^n \to \mathbb{R}.$$

The basic idea is to take a vector $\mathbf{x}_0 \in \mathbb{R}^n$ as start value and define a sequence of vectors $\bigl(\mathbf{x}_n\bigr)_{n\in\mathbb{N}}$ such that we have

$$f(\mathbf{x}_{n+1}) \geq f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N}.$$

Hopefully, this sequence will converge against $\widehat{x} = \arg\max\limits_{\mathbf{x}\in\mathbb{R}} f(\mathbf{x})$. If we do not really know where to start our search, we define $\mathbf{x}_0 := \mathbf{0}$. In order to compute $\mathbf{x}_{n+1}$ given $\mathbf{x}_n$, the idea is to move from $\mathbf{x}_n$ in that direction where we have the biggest change in the values of $f$. This direction happens to be the gradient of $f$ at $\mathbf{x}_n$. Therefore, the definition of $\mathbf{x}_{n+1}$ is given as follows:

$$\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha \cdot \nabla f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N}_0.$$

Here, $\alpha$ is called the step size. It determines by how much we move in the direction of the gradient. In practice, it is best to adapt the step size dynamically during the iteration. Figure 8.2 shows how this is done.

```
findMaximum := procedure(f, gradF, start, eps) {
    x     := start;
    fx    := f(x);
    alpha := 1.0;
    while (true) {
        [xOld, fOld] := [x, fx];
        x  += alpha * gradF(x);
        fx := f(x);
        if (fx < fOld) {
            alpha *= 0.5;
            [x, fx] := [xOld, fOld];
            continue;
        } else {
            alpha *= 1.2;
        }
        if (abs(fx - fOld) <= abs(fx) * eps) {
            return [x, fx];
        }
    }
};
```

Figure 8.2: The gradient ascent algorithm.

The function `findMaximum` takes four arguments:

1. `f` is the function that is to be maximized. It is assumed that `f` takes a vector $\mathtt{x} \in \mathbb{R}^n$ as its input and that it returns a real number.

2. `gradF` is the gradient of `f`. It takes a vector $\mathtt{x} \in \mathbb{R}^n$ as its input and that it returns the vector $\nabla\mathtt{f}(\mathtt{x})$.

3. `start` is the a vector from $\mathbb{R}^n$ that is used as the value of $\mathbf{x}_0$. In practice, we will often use $\mathbf{0} \in \mathbb{R}^n$ as the start vector.

4. `eps` is the precision that we need for the maximum. We will have to say more on how `eps` is exactly related to the precision later. As we are using double precision floating point arithmetic, it won't make sense to use a value for `eps` that is smaller than $10^{-15}$.

Next, let us discuss the implementation of gradient ascent.

1. `x` is initialized with the parameter `start`. Hence, `start` is really the same as $\mathbf{x}_0$.
2. `fx` is the value that the function $f$ takes for the argument `x`.
3. `alpha` is the step size $\alpha$. We initialize `alpha` as 1.0. It will be adapted dynamically.
4. The `while` loop starting in line 5 executes the iteration.
5. In this iteration, we store the values of $\mathbf{x}_n$ and $f(\mathbf{x}_n)$ in the variables `xOld` and `fOld`.
6. Next, we compute $\mathbf{x}_{n+1}$ in line 7 and compute the corresponding value $f(\mathbf{x}_{n+1})$ in line 8.
7. If we are unlucky, $f(\mathbf{x}_{n+1})$ is smaller than $f(\mathbf{x}_n)$. This happens if the step size $\alpha$ is too large. Hence, in this case we decrease the value of $\alpha$, discard both $\mathbf{x}_{n+1}$ and $f(\mathbf{x}_{n+1})$ and start over again.
8. Otherwise, $\mathbf{x}_{n+1}$ is a better approximation of the maximum than $\mathbf{x}_n$. In order to increase the speed of the convergence we will then increase the step size $\alpha$ by 20%.
9. The idea of our implementation is to stop the iteration when the function values of $f(\mathbf{x}_{n+1})$ and $f(\mathbf{x}_n)$ do not differ by more than $\varepsilon$ percent, or, to be more precise, if

   $$f(\mathbf{x}_{n+1}) < f(\mathbf{x}_n) \cdot (1 + \varepsilon).$$

   As the sequence $\bigl(f(\mathbf{x}_n)\bigr)_{n\in\mathbb{N}}$ will be monotonically increasing, i.e. we have

   $$f(\mathbf{x}_{n+1}) \geq f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N},$$

   the condition given above is sufficient. Now, if the increase of $f(\mathbf{x}_{n+1})$ is less than $f(\mathbf{x}_n) \cdot (1+\varepsilon)$ we assume that we have reached the maximum with the required precision. In this case we return both the value of `x` and the corresponding function value $f(\mathtt{x})$.

The implementation of gradient ascent given above is not the most sophisticated variant of this algorithm. It should also be noted that there are algorithms that are more powerful than gradient ascent. The first of these methods is the conjugate gradient method. A refinement of this method is the BFGS-algorithm that has been invented by Broyden, Fletcher, Goldfarb, and Shanno. Unfortunately, we do not have the time to discuss this algorithm. However, our implementation of gradient ascent is sufficient for our applications and as this is not a course on numerical analysis but rather on artificial intelligence we will not delve deeper into this topic but, instead, we refer readers interested in more efficient algorithms to the literature [3].

## 8.3 Logistic Regression

In logistic regression we use a linear model that is combined with the *sigmoid function*. Before we can discuss the details of logistic regression we need define this function and state some of its properties.

### 8.3.1 The Simoid Function

**Definition 9 (Sigmoid Function)** The *sigmoid function* $S : \mathbb{R} \to [0,1]$ is defined as

$$S(t) = \frac{1}{1 + \exp(-t)}.$$

Figure 8.3 on page 58 shows the sigmoid function. ◇

Let us note some immediate consequences of the definition of the sigmoid function. As we have

$$\lim_{x\to-\infty} \exp(-x) = \infty, \quad \lim_{x\to+\infty} \exp(-x) = 0, \quad \text{and} \quad \lim_{x\to\infty} \frac{1}{x} = 0,$$

the sigmoid function has the following properties:

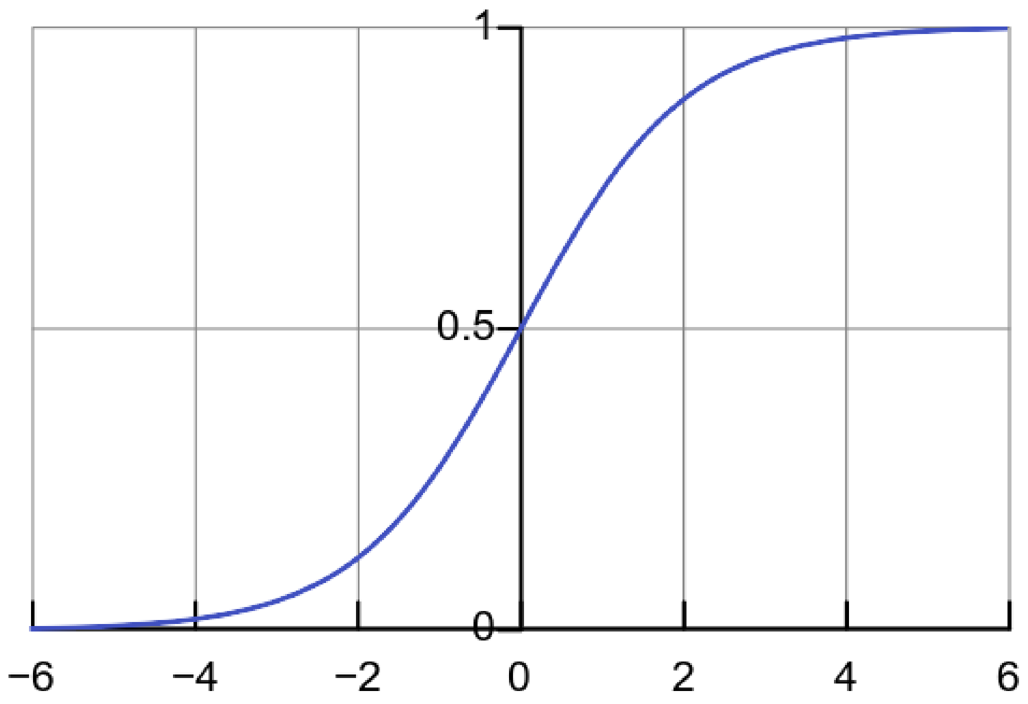

Figure 8.3: The sigmoid function.

$$\lim_{t \to -\infty} S(t) = 0 \quad \text{and} \quad \lim_{t \to +\infty} S(t) = 1.$$

Another important property is the symmetry of the sigmoid function. Figure 8.3 shows that if the sigmoid function is shifted down by $\frac{1}{2}$, the resulting function is *centrally symmetric*, i.e. we have

$$S(-t) - \frac{1}{2} = -\Big(S(t) - \frac{1}{2}\Big).$$

Adding $\dfrac{1}{2}$ on both sides of this equation shows that this is equivalent to the equation

$$\boxed{S(-t) = 1 - S(t),}$$

The proof of this fact runs as follows:

$$\begin{aligned}
1 - S(t) &= 1 - \frac{1}{1+\exp(-t)} && \text{by definition of } S(t) \\
&= \frac{1+\exp(-t)-1}{1+\exp(-t)} && \text{common denominator} \\
&= \frac{\exp(-t)}{1+\exp(-t)} && \text{simplify} \\
&= \frac{1}{1+\exp(+t)} && \text{expand fraction by } \exp(t) \\
&= S(-t). \quad \square && \text{by definition of } S(-t)
\end{aligned}$$

The exponential function can be expressed via the sigmoid function. Let us start with the definition of the sigmoid function.

$$S(t) = \frac{1}{1+\exp(-t)}$$

Multiplying this equation with denominator yields

$$S(t) \cdot \big(1+\exp(-t)\big) = 1.$$

Dividing both sides by $S(t)$ gives:

$$\begin{aligned} & 1+\exp(-t) = \frac{1}{S(t)} \\ \Leftrightarrow \quad & \exp(-t) = \frac{1}{S(t)} - 1 \\ \Leftrightarrow \quad & \exp(-t) = \frac{1-S(t)}{S(t)} \end{aligned}$$

We highlight this formula, as we need it later

$$\boxed{\exp(-t) = \frac{1-S(t)}{S(t)}.}$$

If we take the reciprocal of both sides of this equation, we have

$$\exp(t) = \frac{S(t)}{1-S(t)}.$$

Applying the natural logarithm on both sides of this equation yields

$$t = \ln\left(\frac{S(t)}{1-S(t)}\right).$$

This shows that the inverse of the sigmoid function is given as

$$\boxed{S^{-1}(y) = \ln\left(\frac{y}{1-y}\right).}$$

This function is known as the *logit function*. Next, let us compute the derivative of $S(t)$, i.e. $S'(t) = \frac{\mathrm{d}S}{\mathrm{d}t}$. We have

$$\begin{aligned} S'(t) &= -\frac{-\exp(-t)}{\big(1+\exp(-t)\big)^2} \\ &= \exp(-t) \cdot S(t)^2 \\ &= \frac{1-S(t)}{S(t)} \cdot S(t)^2 \\ &= \big(1-S(t)\big) \cdot S(t) \end{aligned}$$

We have shown

$$\boxed{S'(t) = \big(1-S(t)\big) \cdot S(t).}$$

We will later need the derivative of the logarithm of the logistic function. We define

$$L(t) := \ln\big(S(t)\big).$$

Then we have

$$\begin{aligned} L'(t) &= \frac{S'(t)}{S(t)} && \text{by the chain rule} \\ &= \frac{(1-S(t))\cdot S(t)}{S(t)} \\ &= 1 - S(t) \\ &= S(-t) \end{aligned}$$

As this is our most important result, we highlight it:

$$\boxed{L'(t) = S(-t) \quad \text{where} \quad L(t) := \ln\bigl(S(t)\bigr).}$$

### 8.3.2 The Model of Logistic Regression

We use the following model to compute the probability that an object with features $\mathbf{x}$ will be of the given class:

$$P(y = +1 \mid \mathbf{x}; \mathbf{w}) = S(\mathbf{x} \cdot \mathbf{w}).$$

Here $\mathbf{x} \cdot \mathbf{w}$ denotes the dot product of the vectors $\mathbf{x}$ and $\mathbf{y}$. Seeing this model the first time you might think that this model is not very general and that it can only be applied in very special circumstances. However, the fact is that the features can be functions of arbitrary complexity and hence this model is much more general than it appears on first sight.

As $y$ can only take the values $+1$ or $-1$ and complementary probabilities add up to 1, we have

$$P(y = -1 \mid \mathbf{x}; \mathbf{w}) = 1 - P(y = +1 \mid \mathbf{x}; \mathbf{w}) = 1 - S(\mathbf{x} \cdot \mathbf{w}) = S(-\mathbf{x} \cdot \mathbf{w}).$$

Hence, we can combine the equations for $P(y = -1 \mid \mathbf{x}; \mathbf{w})$ and $P(y = +1 \mid \mathbf{x}; \mathbf{w})$ into a single equation

$$\boxed{P(y \mid \mathbf{x}; \mathbf{w}) = S\bigl(y \cdot (\mathbf{x} \cdot \mathbf{w})\bigr).}$$

Given $N$ objects $o_1, \cdots, o_n$ with feature vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ we want to determine the weight vector $\mathbf{w}$ such that the likelihood $\ell(\mathbf{X}, \mathbf{y})$ of all of our observations is maximized. This approach is called the maximum likelihood estimation of the weights. As we assume the probabilities of different objects to be independent, the individual probabilities have to be multiplied to compute the overall likelihood $\ell(\mathbf{X}, \mathbf{y}; \mathbf{w})$ of a given training set:

$$\ell(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i; \mathbf{w}).$$

Here, we have combined the different attribute vectors $\mathbf{x}_i$ into the matrix $\mathbf{X}$. Since it is easier to work with sums than with products, instead of maximizing the function $\ell(\mathbf{X}, \mathbf{y}; \mathbf{w})$ we maximize the function

$$\ell\ell(\mathbf{X}, \mathbf{y}; \mathbf{w}) := \ln\bigl(\ell(\mathbf{X}, \mathbf{y}; \mathbf{w})\bigr).$$

As the natural logarithm is a monotone function, the functions $\ell(\mathbf{X}, \mathbf{y}; \mathbf{w})$ and $\ell\ell(\mathbf{X}, \mathbf{y}; \mathbf{w})$ take their maximum at the same value of $\mathbf{w}$. Since we have

$$\ln(a \cdot b) = \ln(a) + \ln(b),$$

the natural logarithm of the likelihood is

$$\boxed{\ell\ell(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \sum_{i=1}^{N} \ln\Bigl(S\bigl(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\bigr)\Bigr) = \sum_{i=1}^{N} L\bigl(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\bigr).}$$

Our goal is to maximize the likelihood. Since this is the same as maximizing the log-likelihood, we need to determine those values of the coefficients $\mathbf{w}$ that satisfy

$$\frac{\partial}{\partial\, w_j}\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w}) = 0.$$

In order to compute the partial derivative of $\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w})$ with respect to the coefficients $\mathbf{w}$ we need to compute the partial derivative of the dot product $\mathbf{x}_i \cdot \mathbf{w}$. We define

$$h(\mathbf{w}) := \mathbf{x}_i \cdot \mathbf{w} = \sum_{j=1}^{D} x_{i,j} \cdot w_j.$$

Then we have

$$\frac{\partial}{\partial\, w_j} h(\mathbf{w}) = x_{i,j}.$$

Now we are ready to compute the partial derivative of $\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w})$ with respect to $\mathbf{w}$:

$$\begin{aligned}
&\frac{\partial}{\partial\, w_j}\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w}) \\
=\ &\frac{\partial}{\partial\, w_j}\sum_{i=1}^{N} L\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big) \\
=\ &\sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S\big(-y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big)
\end{aligned}$$

Hence, the partial derivative of the log-likelihood function is given as follows:

$$\boxed{\frac{\partial}{\partial\, w_j}\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w}) = \sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w})}$$

Next, we have to find the value of $\mathbf{w}$ such that

$$\sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w}) = 0 \quad \text{for all } j \in \{1, \cdots, D\}.$$

These are $D$ equation for the $D$ variables $w_1, \cdots w_D$. Due to the occurrence of the sigmoid function, these equations are nonlinear. In order to find the value of $\mathbf{w}$ that maximizes the likelihood, we will instead use the method of gradient ascent to compute the best value of $\mathbf{w}$. This method has been outlined in the previous section.

### 8.3.3 Implementing Logistic Regression

In order to implement logistic regression we need a data structure for tabular data. Figure 8.4 on page 62 shows the class table that can be used to administer this kind of data. Figure 8.4 shows an example of tabular data that is stored in a csv file. In this case, the data stores the hours a student has learned for a particular exam and the fact whether the student has passed of failed. The first column stores pass or fail, where a pass is coded using the number 1, while a fail is coded as 0. The second column stores the number of hours that the student has learned in order to pass the exam.

There is no need for us to discuss every detail of the implementation of the class `table`. The important thing to note is that the data is stored as a list of lists in the member variable `mData`. Each of the inner lists corresponds to one row of the csv file. This member variable can be accessed using the function getData. The function `readTable` has the responsibility to read a csv file and to convert it into an object of class `table`. In order to do this, it has to be called with two arguments. The first argument is the file name, the second argument is a list of the types of each column in the csv file. For example, to read the file “`exam.csv`” we would call `readTable` as follows:

```
readTable("exam.csv", ["int", "double"]).
```

```
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) {
    all := readFile(fileName);
    columnNames := split(all[1], ',\s*');
    data := [];
    for (i in [2 .. #all]) {
        row := split(all[i], ',\s*');
        data[i-1] := [eval("$type$($s$)") : [type, s] in types >< row];
    }
    return table(columnNames, types, data);
};
```

Figure 8.4: A class to represent tabular data.

The program shown in Figure 8.6 on page 64 implements logistic regression. As there are a number of subtle points that might easily be overlooked otherwise, we proceed to discuss this program line by line.

1. First, we have to load both the class `table` and our implementation of gradient ascent that has already been discussed in Section 8.2.
2. Line 4 implements the sigmoid function
$$S(x) = \frac{1}{1+\exp(-x)}.$$
3. Line 5 starts the implementation of the natural logarithm of the sigmoid function, i.e. we implement
$$L(x) = \ln\bigl(S(X)\bigr) = \ln\Bigl(\frac{1}{1+\exp(-x)}\Bigr) = -\ln\bigl(1+\exp(-x)\bigr).$$
The implementation is more complicated than you might expect. The reason has to do with overflow. Consider values of $x$ that are smaller than, say, $-1000$. The problem is that the expression `exp`$(1000))$ evaluates to `Infinity`, which represents the mathematical value $\infty$. But then $1 + $ `exp`$(1000))$ is also `Infinity` and finally `log(1 + exp(1000))` is `Infinity`. However, in reality we have
$$\ln\bigl(1+\exp(1000)\bigr) \approx 1000.$$

```
Pass, Hours
0,    0.50
0,    0.75
0,    1.00
0,    1.25
0,    1.50
0,    1.75
1,    1.75
0,    2.00
1,    2.25
0,    2.50
1,    2.75
0,    3.00
1,    3.25
0,    3.50
1,    4.00
1,    4.25
1,    4.50
1,    4.75
1,    5.00
1,    5.50
```

Figure 8.5: Results of an exam.

The argument works as follows:

$$\begin{array}{lcll}
\ln\bigl(1+\exp(x)\bigr) & = & \ln\bigl(\exp(x)\cdot(1+\exp(-x))\bigr) & \\
 & = & \ln\bigl(\exp(x)\bigr)+\ln\bigl(1+\exp(-x)\bigr) & \\
 & = & x+\ln\bigl(1+\exp(-x)\bigr) & \\
 & \approx & x+\ln(1)+\exp(-x) & \text{Taylor expansion of } \ln(1+x) \\
 & = & x+0+\exp(-x) & \\
 & \approx & x & \text{since } \exp(-x)\approx 0 \text{ for large } x
\end{array}$$

This is the reason that `logSigmoid` returns `x` if `x` is less than $-100$.

4. The function `ll`$(\mathbf{x},\mathbf{y},\mathbf{w})$ computes the log-lokelihood $\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w}) = \sum\limits_{i=1}^{N} L\bigl(y_i\cdot(\mathbf{x}_i\cdot\mathbf{w})\bigr)$. Here $L$ denotes the natural logarithm of the sigmoid of the argument. It is assumed that `x` is a matrix. Every observation corresponds to a row in this matrix, i.e. the vector $\mathbf{x}_i$ is the feature vector containing the features of the $i$-th observation. $\mathbf{y}$ is a vector describing the outcomes, i.e. the elements of this vector are either $+1$ or $-1$. Finally, $\mathbf{w}$ is the vector of coefficients.

5. The function `gradLL`$(\mathbf{x},\mathbf{y},\mathbf{w})$ computes the gradient of the log-lokelihood according to the formula

$$\frac{\partial}{\partial\, w_j}\ell\ell(\mathbf{X},\mathbf{y};\mathbf{w}) = \sum_{i=1}^{N} y_i\cdot x_{i,j}\cdot S(-y_i\cdot\mathbf{x}_i\cdot\mathbf{w}).$$

   The different components of this gradient are combined into a vector. The arguments are the same as the arguments to the log-lokelihood.

6. Finally, the function `logisticRegressionFile` takes two arguments. The first argument is the name of the csv file containing the data, while the second argument is a list specifying the types of the columns.

```
load("table.stlx");
load("gradient-ascent.stlx");

sigmoid := procedure(x) { return 1.0 / (1.0 + exp(-x)); };
logSigmoid := procedure(x) {
    if (x >= -100) {
        return -log(1.0 + exp(-x));
    } else {
        return x;
    }
};
ll := procedure(x, y, w) {
    result := 0;
    for (i in [1 .. #x]) {
        result += logSigmoid(y[i] * (x[i] * w));
    }
    return result;
};
gradLL := procedure(x, y, w) {
    result := [];
    for (j in [1 .. #x[1]]) {
        result[j] := 0;
        for (i in [1 .. #x]) {
            result[j] += y[i] * x[i][j] * sigmoid((-y[i]) * (x[i] * w));
        }
    }
    return la_vector(result);
};
logisticRegressionFile := procedure(fileName, types) {
    csv    := readTable(fileName, types);
    data   := csv.getData();
    number := #data;
    dmnsn  := #data[1];
    yList  := [];
    xList  := [];
    for (i in [1 .. number]) {
        yList[i] := data[i][1];
        xList[i] := la_vector([1.0] + data[i][2..]);
    }
    x := la_matrix(xList);
    y := la_vector([2 * y - 1 : y in yList]);
    start := la_vector([0.0 : i in [1 .. dmnsn]]);
    eps   := 10 ** -15;
    f     := w |=> ll(x, y, w);
    gradF := w |=> gradLL(x, y, w);
    return findMaximum(f, gradF, start, eps)[1];
};
```

Figure 8.6: An implementation of logistic regression.

The elements of this list have to be either `"int"` or `"double"`. The task of this function is to read the csv file, convert the data in the matrix `x` and the vector `y`, and then use gradient ascent to find the best coefficients.

If we run the function `logisticRegressionFile` using the data shown in Figure 8.5 via the command

```
logisticRegressionFile("exam.csv", ["int", "double"]);
```

the resulting coefficients are:

```
<<-4.077649741107752 1.5046211108850898>>
```

This shows that the probability $P(h)$ that a student who has studied for $h$ hours will pass the exam is given approximately as follows:

$$P(h) \approx \frac{1}{1 + \exp(4.1 - 1.5 \cdot h)}$$

Figure 8.7 shows a plot of the probability $P(x)$. This figure has been taken from the Wikipedia article on logistic regression. It has been created by Michaelg2015.

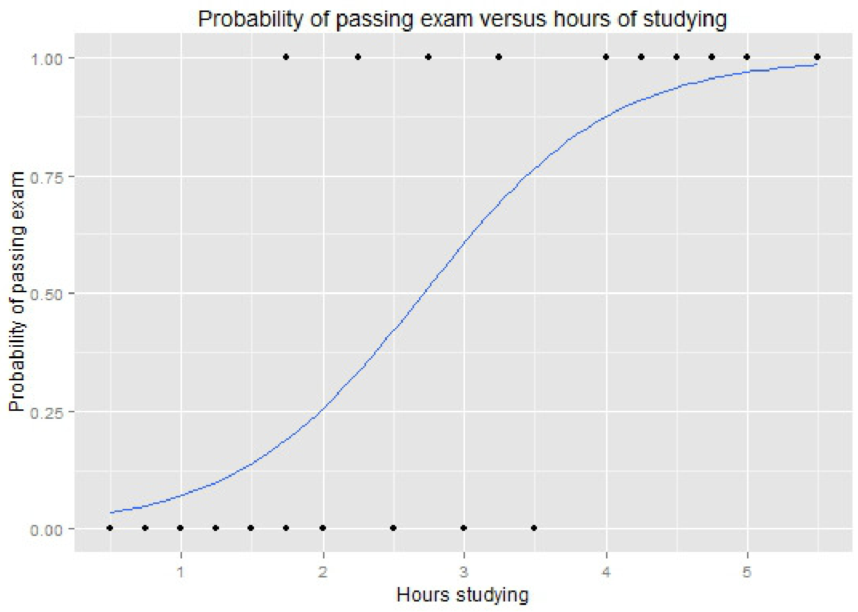

Figure 8.7: Probability of passing an exam versus hours of studying.

# Chapter 9

# Neural Networks

A neural network is built from *neurons.* At the abstraction level that we are looking at neural networks, a single neuron with $n$ inputs is defined as a pair $\langle \mathbf{w}, b \rangle$ where vector $\mathbf{w} \in \mathbb{R}^m$ is called the *weight vector* and $b \in \mathbb{R}$ is called the *bias.* Conceptually, a neuron is a function $p$ that maps a vector $\mathbf{x} \in \mathbb{R}^m$ into the interval $[0, 1]$. This function is defined as follows:

$$p(\mathbf{x}; \mathbf{w}) := a(\mathbf{x} \cdot \mathbf{w} + b),$$

where $a$ is called the *activation function.* In our applications, we will use the sigmoid function as our activation function, i.e. we have

$$a(t) := S(t) = \frac{1}{1 + \exp(-t)}.$$

The function $p$ modeling the neuron can also be written using index notation. If

$$\mathbf{w} = \langle w_1, \cdots, w_m \rangle^T$$

is the weight vector and

$$\mathbf{x} = \langle x_1, \cdots, x_m \rangle^T$$

is the input vector, then

$$p(\mathbf{x}; \mathbf{w}) = S\left(\sum_{i=1}^{m} x_i \cdot w_i + b\right)$$

if we use the sigmoid function as our activation function. If you compare $p(\mathbf{x}; \mathbf{w})$ to a similar function appearing in the last chapter, you will notice that so far a neuron works just like logistic regression. The only difference is that the bias $b$ is now explicit.

A neural network is a layered network of neurons. Formally, the *topology* of a neural network is given by a number $L \in \mathbb{N}$ and a list of $[m(1), \cdots, m(L)]$ of $L$ natural numbers. The number $L$ is called the number of layers and for $i \in \{2, \cdots, L\}$ the number $m(i)$ is the number of neurons in the $l$-th layer. The first layer is called the input layer, the last layer (i.e. the layer with index $L$) is called the output layer and the remaining layers are called *hidden layers.* If there is more than one hidden layer, the neural network is called a *deep neural network.*

As the first layer is the input layer, the *input dimension* is defined as $m(1)$. Similarly, the *output dimension* is defined as $m(L)$. Every node in the $l$-th layer is connected to every node in the $(l+1)$-th layer. The weight $w_{j,k}^{(l)}$ is the weight of the connection from the $k$-th neuron in layer $l-1$ to the $j$-th neuron in layer $l$. The weights in layer $l$ are combined into the weight matrix of layer $W^{(l)}$ which is defined as

$$W^{(l)} := \left(w_{j,k}^{(l)}\right).$$

Note that $W^{(l)}$ is an $m(l-1) \times m(l)$ matrix, i.e. we have

$$W^{(l)} \in \mathbb{R}^{m(l-1) \times m(l)}.$$

The $j$-th neuron in layer $l$ has the bias $b_j^{(l)}$. Then, the activation of the $j$-th neuron in layer $l$ is denoted as $a_j^{(l)}$ and is defined as

$$a_j^{(l)} := S\left(\sum_{k=1}^{n(l-1)} w_{j,k}^{(l)} \cdot a^{(l-1)} + b_j^{(l)}\right).$$

The output of our neural network for an input $\mathbf{x}$ is given by the neuron in the output layer, i.e. the output vector $\mathbf{o}(\mathbf{x}) \in \mathbb{R}^{(m(L))}$ is defined as

$$\mathbf{o}(\mathbf{x}) = \langle a_1^{(L)}(\mathbf{x}), \cdots, a_{m(L)}^{(L)}(\mathbf{x})\rangle.$$

We assume that we have $n$ training examples

$$\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\rangle \quad \text{for } i = 1, \cdots, n$$

such that

$$\mathbf{x}^{(i)} \in \mathbb{R}^{m(1)} \text{ and } \mathbf{y}^{(i)} \in \mathbb{R}^{m(L)}$$

The *quadratic error cost function* is defined as

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{2 \cdot m} \cdot \sum_{i=1}^{n} \Big(\mathbf{o}\big(\mathbf{x}^{(i)}\big) - \mathbf{y}^{(i)}\Big)^2.$$

Note that the cost function is additive in the training examples $\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\rangle$. In order to simplify the notation we therefore define

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big) := \frac{1}{2} \cdot \Big(\mathbf{o}(\mathbf{x}) - \mathbf{y}\Big)^2.$$

Then, we have

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{m} \cdot \sum_{i=1}^{n} C_{\mathbf{x}^{(\mathbf{i})},\mathbf{y}^{(\mathbf{i})}}\Big(W^{(2)}, \cdots W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big).$$

As the notation

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big)$$

is far too heavy, we will abbreviate this term as $C$ in the following discussion of the backpropagation algorithm.

## 9.1 Backpropagation

# Chapter 10

# Clustering and Retrieval

The *inverse document frequency* $\texttt{idf}(w)$ of a word $w$ with respect to a corpus $C$ is defined as follows:

$$\texttt{idf}(w) := \ln\left(\frac{\texttt{\#}C}{1 + \texttt{count}(w, C)}\right).$$

Here, $\texttt{\#}C$ is the number of documents that are present in the corpus $C$, while $\texttt{count}(w, C)$ yields the number of documents in the corpus $C$ that contain the word $w$, i.e. we have

$$\texttt{count}(w, C) := \texttt{\#}\{d \in C \mid w \in d\}.$$

# Bibliography

[1] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

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

[3] Jan A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer Publishing, 2005.