An Introduction to Artificial Intelligence

— Lecture Notes in Progress —

Prof. Dr. Karl Stroetmann

February 16, 2017

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

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

The lecture notes are subject to continuous change. Provided the program `git` is installed on your computer, the repository containing the lecture notes can be cloned using the command

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

Once you have cloned the repository, the command

```
git pull
```

can be used to load the current version of these lecture notes from `github`.

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 $\cdots$
	You don't have to worry any more, I have fixed the problem. He is dead now.

May The Force be with us on achieving our goals!

1.2 Literature

The main sources of these lecture notes are the following:

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

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

2. The book

   Introduction to Artificial Intelligence

   written by Stuart Russel and Peter Norvig [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

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

   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. $s_n = \texttt{goal}$, 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. $\diamond$

Example: We illustrate the notion of a search problem with the following example, which is also known as the missionaries and cannibals problem: Three missionaries and three infidels have to cross a river that runs from the west to the east. Initially, they are on the northern shore. There is just one small boat and that boat has only room for at most two passengers. Both the missionaries and the infidels can steer the boat. However, if at any time the missionaries are confronted with a majority of infidels on either shore of the river, then the missionaries have a problem.

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

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

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

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

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

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.

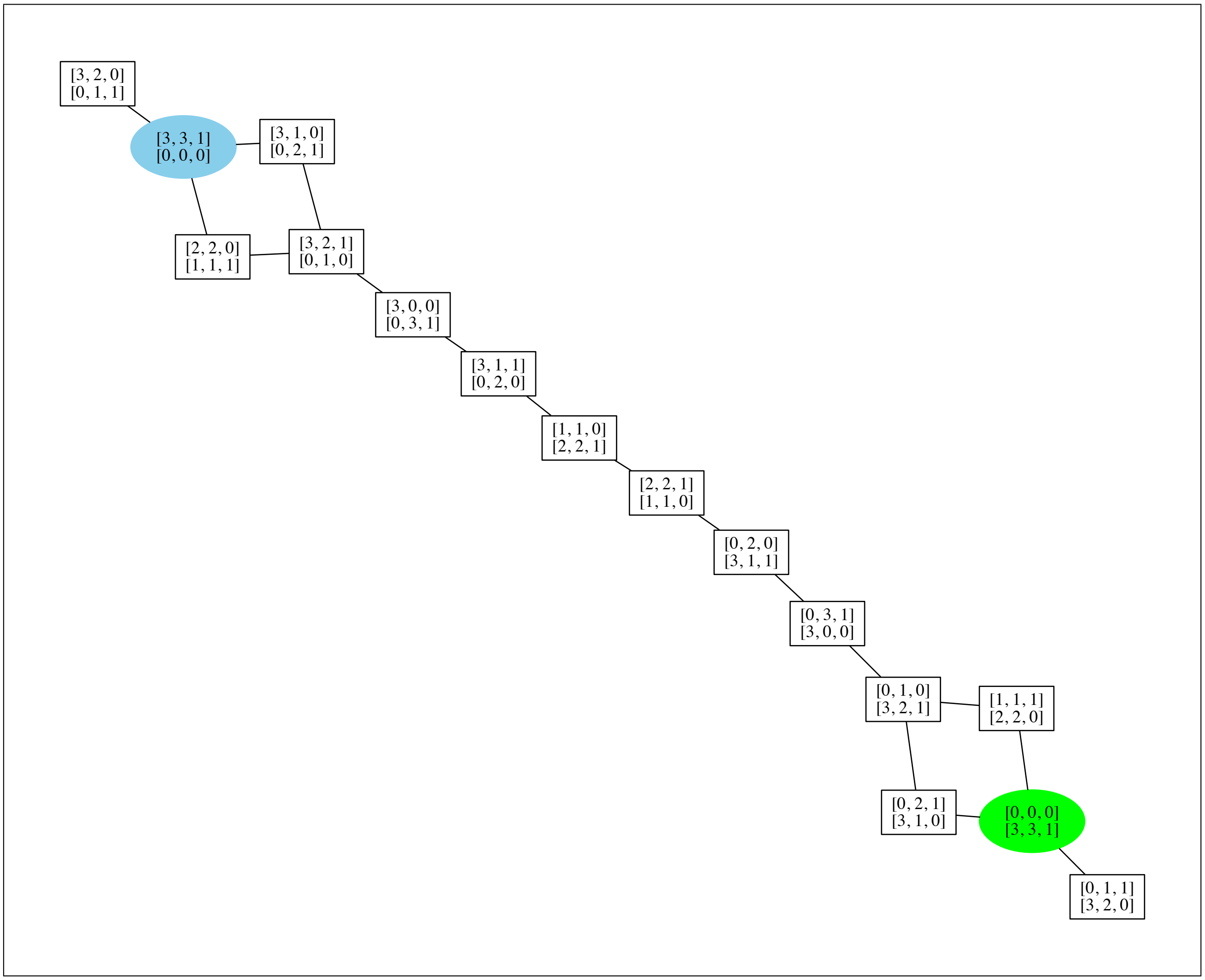

Figure 2.2: A graphical representation of the missionaries and cannibals problem.

2.1 The Sliding Puzzle

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

In order to solve the 3×3 sliding puzzle shown in Figure 2.3 we have to transform the state shown on the left of Figure 2.3 into the state shown on the right of this figure. The following operations are permitted when transforming a state of the sliding puzzle:

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

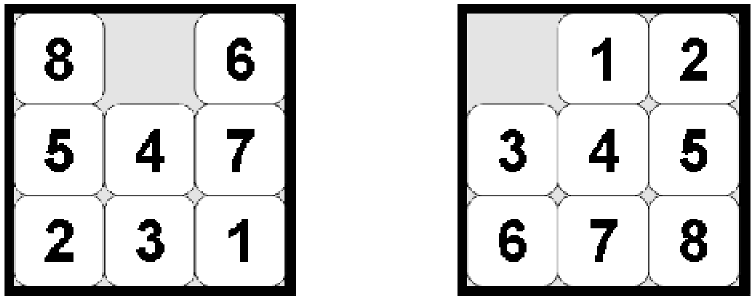

Figure 2.3: The 3×3 sliding puzzle.

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

http://mypuzzle.org/sliding.

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

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

This is already a big number, but 181440 states can still be stored in a modern computer. However, the 4×4 sliding puzzle has

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

different states reachable from a given start state. If a state is represented as matrix containing 16 numbers and we store every number using just 4 bits, we still need $16 \cdot 4 = 64$ bits or 8 bytes 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 3×3 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 3×3 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 10 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 10.
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 3×3 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 12 shows an implementation of breadth first search that uses a queue to store the set `Frontier`. However, when we run this version, it turns out that the solution of the 3×3 sliding puzzle needs about 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 3×3 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 3×3 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 14. 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 3×3 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 16 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 14, we do <u>not</u> 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 3×3 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 18 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

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

At least this is true in the beginning of the search. As the size of the memory that is needed is the most constraining factor when searching, it is important to cut down this size. 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 19 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 20.

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

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

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

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

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

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

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

Hence, the heuristic is admissible iff it is optimistic: An admissible heuristic must never overestimate the distance to the goal, but it is free to underestimate this distance.

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

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

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 8 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:

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

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

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

The Manhattan heuristic measure the number of moves that would be needed if we wanted to put every tile of s into its final positions and if we were allowed to slide tiles over each other. Figure 2.17 on page 22 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 $h(s_1) \leq 1 + h(s_2)$. $\square$

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 23 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 3×3 sliding puzzle, it takes about 17 seconds to solve the instance shown in Figure 2.3 on page 8. 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 26 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.18 on page 23 as the algorithm for bidirectional search shown in Figure 2.15 on page 19 relates to breadth first search shown in Figure 2.5 on page 10. Hence, we will not discuss the details any further.

When we run bidirectional A* search for the 3×3 sliding puzzle shown in Figure 2.3 on page 8, the program takes 2 second but only uses $2,963$ states. Therefore, I have tried to solve the 4×4 sliding puzzle shown in Figure 2.20 on page 27 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 27 shows an implementation of IDA* in SetlX. We proceed to discuss this program.

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

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

   computes an estimate of the distance between s_1 and s_2.

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 4×4 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 3×3 sliding puzzle shown on Figure 2.3 on page 8, this algorithm takes about 2.6 seconds to solve the puzzle. For the 4×4 sliding puzzle, the algorithm takes about 518 seconds. Although this is more than the time needed by bidirectional A* search, the good news is that the IDA* algorithm does not need much memory since basically only the path discovered so far is stored in memory. Hence, IDA* is a viable alternative if the available memory 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 29 and Figure 2.23 on page 31. 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 $[d, s] \in \texttt{Frontier}$, then the state s has been encountered in the A* search and it is estimated that a path leading from `start` to `goal` and passing through s would have a length of d.

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

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

Iterative deepening A*-IDA* is a complete search algorithm. On the instance of the 3×3 sliding puzzle shown on Figure 2.3 on page 8, this algorithm takes about 1.4 seconds to solve the puzzle. For the 4×4 sliding puzzle, if the algorithm is allowed to visit at most 3 000 states, the algorithm takes less than 9 seconds. 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.

Exercise 2: Assume that you have 3 water buckets: The first bucket can hold 12 liters of water, the second bucket can hold 8 liters, while the last bucket can hold 3 liters. There are three types of action:

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

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

Chapter 3

Constraint Satisfaction

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

1. The first section defines the notion of a constraint satisfaction problem. In order to illustrate this concept, two examples of constraint satisfaction problems are presented. Finally, we present applications of constraint satisfaction problems.

2. The simplest algorithm to solve a constraint satisfaction problem is via *Backtracking Search*. This is presented in Section 2.

3. Backtracking Search can be refined to *Non-Chronological Backtracking*, which is presented in Section 3.

4. Finally, the last section shows how constraint satisfaction problems can be solved via *local search*.

3.1 Formal Definition of Constraint Satisfaction Problems

Formally, we define a constraint satisfaction problem as a triple

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

where

1. `Vars` is a set of strings which serve as *variables*,

2. `Values` is a set of *values* for the variables in `Vars`.

3. `Constraints` is a set of formulae from *first order logic*. Each of these formulae is called a *constraint* of $\mathcal{P}$.

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

In the following, the abbreviation CSP is short for *constraint satisfaction problem*. Given a CSP

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

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

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

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

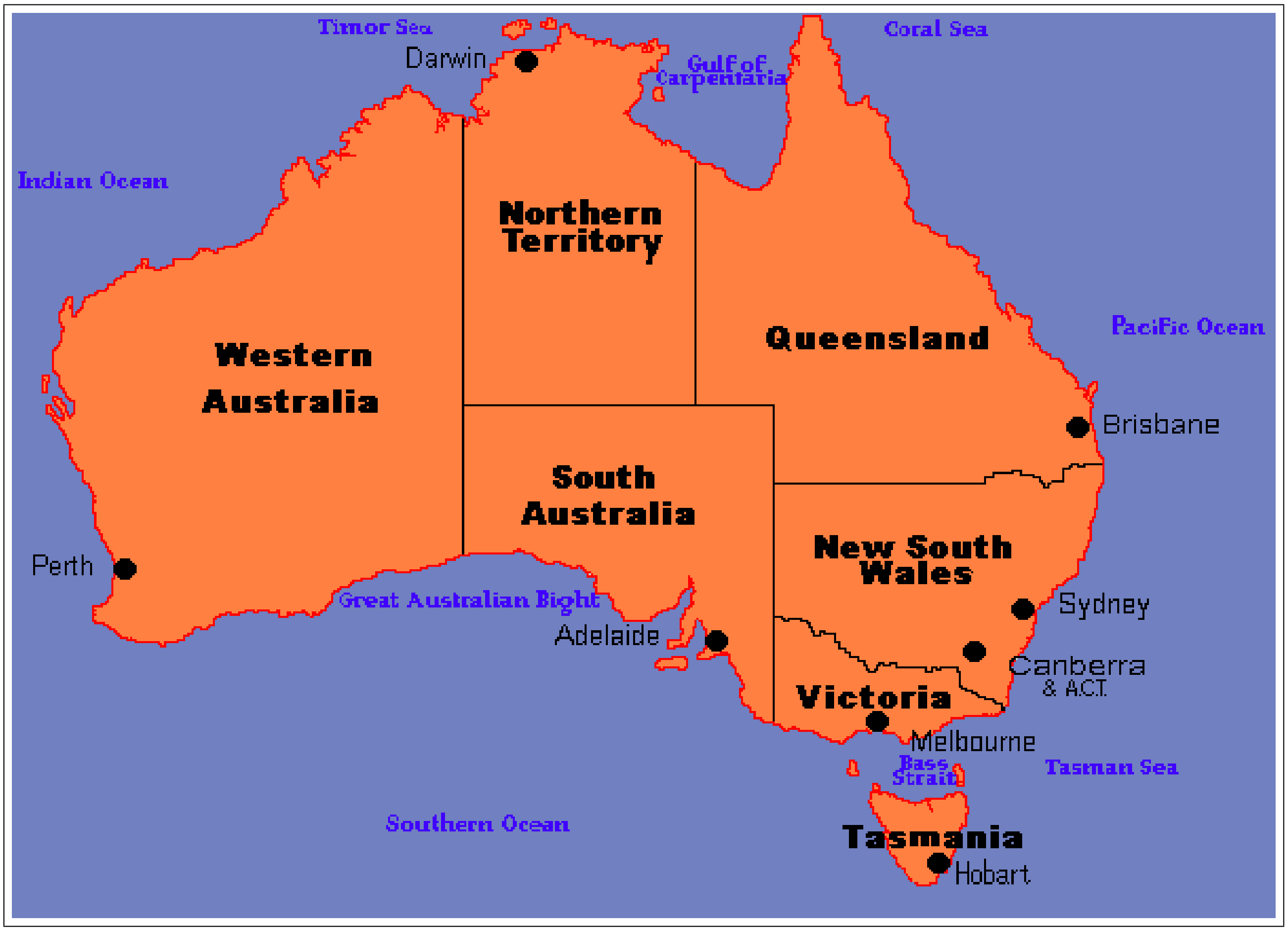

Figure 3.1: A map of Australia.

3.1.1 Example: Map Coloring

In map colouring a map showing different state borders is given and the task is to colour the different states such that no two states that have a common border share the same colour. Figure 3.1 on page 33 shows a map of Australia. There are seven different states in Australia:

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

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

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

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

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

3.1.2 Example: The Eight Queens Puzzle

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

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

where for $i \in \{1, \cdots, 8\}$ the variable $\texttt{V}i$ specifies the column of the queen that is placed in row i. As the columns run from one to eight, we define the set 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

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

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

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

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

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

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

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 35.

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

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

Figure 3.3 on page 35 shows a SetlX program that can be used to create the eight queens puzzle as a Csp. The code shown in this figure is more general than the eight queens puzzle: Given a natural number n, the function call `queensCSP`(n) creates a constraint satisfaction problem $\mathcal{P}$ that generalizes the eight queens problem to the problem of putting n queens on a board of size n times n.

```
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.

The beauty of constraint programming is the fact that we will be able to develop a so called *constraint solver* that takes as input a CSP like the one produced by the program shown in Figure 3.3 and that is then capable of computing a solution.

3.1.3 Applications

Besides the toy problems discussed so far, there are a number of industrial applications of constraint satisfaction problems. The most important application seem to be variants of scheduling problems. A simple example of a scheduling problem is the problem of generating a time table for a school. A school has various teachers, each of which can teach some subjects but not others. Furthermore, there are a number of classes that must be taught in different subjects. The problem is then to assign teachers to classes and to create a time table.

3.2 Backtracking Search

3.3 Non-Chronological Backtracking

3.4 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.

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.

```
solve := procedure(n) {
    Queens := [];
    for (row in [1 .. n]) {
        Queens[row] := rnd({1 .. n});
    }
    iteration := 0;
    while (true) {
        Conflicts   := { [numConflicts(Queens, row), row] : row in [1 ..n] };
        [maxNum, _] := last(Conflicts);
        if (maxNum == 0) {
            return Queens;
        }
        if (iteration % 10 != 0) { // avoid infinite loops
            row := rnd({ row : [num, row] in Conflicts | num == maxNum });
        } else {
            row := rnd({ 1 .. n });
        }
        Conflicts := {};
        for (col in [1 .. n]) {
            Board      := Queens;
            Board[row] := col;
            Conflicts  += { [numConflicts(Board, row), col] };
        }
        [minNum, _] := first(Conflicts);
        Queens[row] := rnd({ col : [num, col] in Conflicts | num == minNum });
        iteration   += 1;
    }
};
```

Figure 3.4: Solving the n queens problem using local search.

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.

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

Figure 3.5: The procedure `numConficts`.

The procedure `numConficts` shown in Figure 3.5 on page 38 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 $n = 1000$ 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.

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.