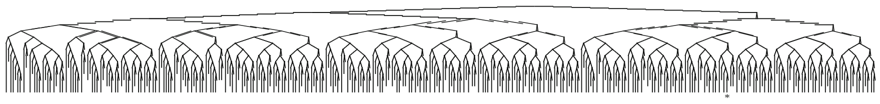
6 Search, Games and Problem Solving

6.1 Introduction

The search for a solution in an extremely large search tree presents a problem for nearly all inference systems. From the starting state there are many possibilities for the first inference step. For each of these possibilities there are again many possibilities in the next step, and so on. Even in the proof of a very simple formula from [Ert93] with three Horn clauses, each with at most three literals, the search tree for SLD resolution has the following shape:



The tree was cut off at a depth of 14 and has a solution in the leaf node marked by $*$. It is only possible to represent it at all because of the small branching factor of at most two and a cutoff at depth 14. For realistic problems, the branching factor and depth of the first solution may become significantly bigger.

Assume the branching factor is a constant equal to 30 and the first solution is at depth 50. The search tree has $30^{50} \approx 7.2 \times 10^{73}$ leaf nodes. But the number of inference steps is even bigger because not only every leaf node, but also every inner node of the tree corresponds to an inference step. Therefore we must add up the nodes over all levels and obtain the total number of nodes of the search tree

$$\sum_{d=0}^{50} 30^d = \frac{1-30^{51}}{1-30} = 7.4 \times 10^{73},$$

which does not change the node count by much. Evidently, nearly all of the nodes of this search tree are on the last level. As we will see, this is generally the case. But now back to the search tree with the 7.4×10^{73} nodes. Assume we had 10,000

W. Ertel, *Introduction to Artificial Intelligence*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-58487-4_6

computers which can each perform a billion inferences per second, and that we could distribute the work over all of the computers with no cost. The total computation time for all 7.4×10^{73} inferences would be approximately equal to

$$\frac{7.4 \times 10^{73} \text{ inferences}}{10000 \times 10^{9} \text{ inferences/sec}} = 7.4 \times 10^{60} \text{ sec} \approx 2.3 \times 10^{53} \text{ years},$$

which is about 10^{43} times as much time as the age of our universe. By this simple thought exercise, we can quickly recognize that there is no realistic chance of searching this kind of search space completely with the means available to us in this world. Moreover, the assumptions related to the size of the search space were completely realistic. In chess for example, there are over 30 possible moves for a typical situation, and a game lasting 50 half-turns is relatively short.

How can it be then, that there are good chess players—and these days also good chess computers? How can it be that mathematicians find proofs for theorems in which the search space is even much bigger? Evidently we humans use intelligent strategies which dramatically reduce the search space. The experienced chess player, just like the experienced mathematician, will, by mere observation of the situation, immediately rule out many actions as senseless. Through his experience, he has the ability to evaluate various actions for their utility in reaching the goal. Often a person will go by feel. If one asks a mathematician how he found a proof, he may answer that the intuition came to him in a dream. In difficult cases, many doctors find a diagnosis purely by feel, based on all known symptoms. Especially in difficult situations, there is often no formal theory for solution-finding that guarantees an optimal solution. In everyday problems, such as the search for a runaway cat in Fig. 6.1 on page 93, intuition plays a big role. We will deal with this kind of *heuristic search method* in Sect. 6.3 and additionally describe processes with which computers can, similarly to humans, improve their heuristic search strategies by learning.

First, however, we must understand how *uninformed search*, that is, blindly trying out all possibilities, works. We begin with a few examples.

Example 6.1 With the 8-puzzle, a classic example for search algorithms [Nil98, RN10], the various algorithms can be very visibly illustrated. Squares with the numbers 1 to 8 are distributed in a 3×3 matrix like the one in Fig. 6.2 on page 93. The goal is to reach a certain ordering of the squares, for example in ascending order by rows as represented in Fig. 6.2 on page 93. In each step a square can be moved left, right, up, or down into the empty space. The empty space therefore moves in the corresponding opposite direction. For analysis of the search space, it is convenient to always look at the possible movements of the empty field.

The search tree for a starting state is represented in Fig. 6.3 on page 94. We can see that the branching factor alternates between two, three, and four. Averaged over two levels at a time, we obtain an average branching factor[1] of $\sqrt{8} \approx 2.83$. We see

[1]The average branching factor of a tree is the branching factor that a tree with a constant branching factor, equal depth, and an equal amount of leaf nodes would have.

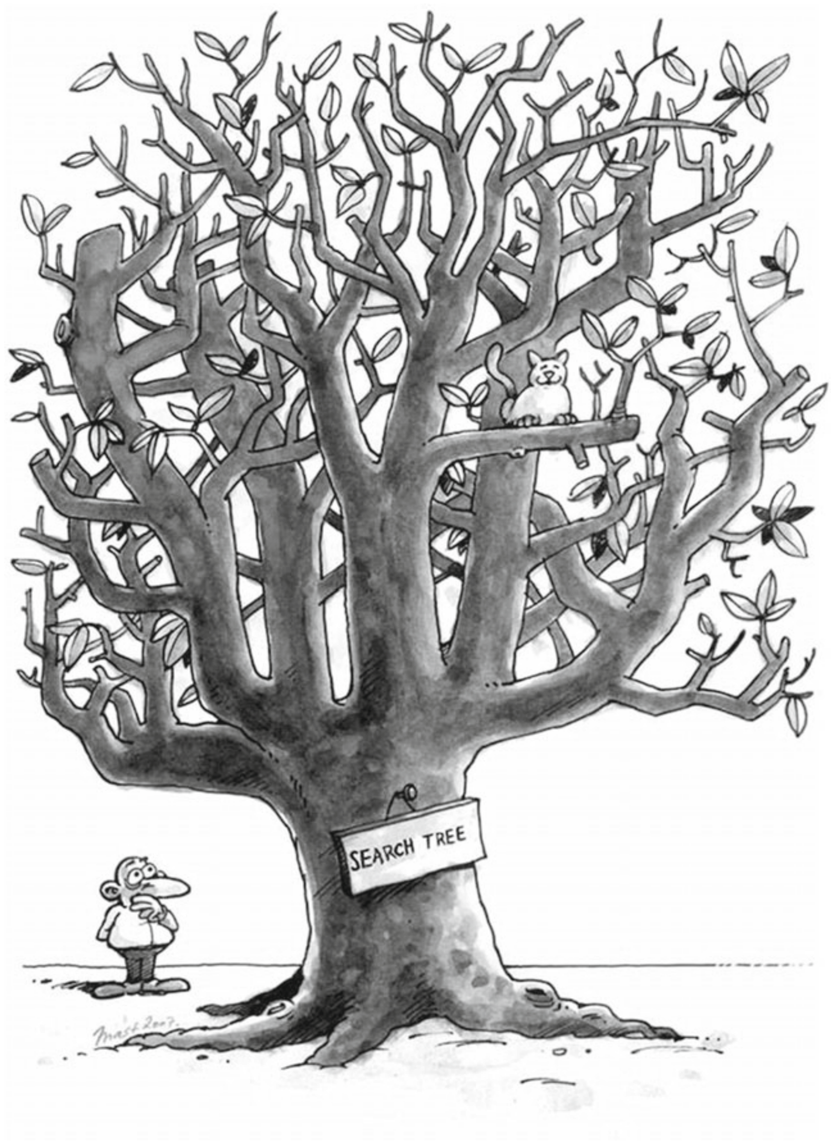
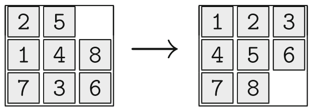


Fig. 6.1 A heavily trimmed search tree—or: “Where is my cat?”

Fig. 6.2 Possible starting and goal states of the 8-puzzle



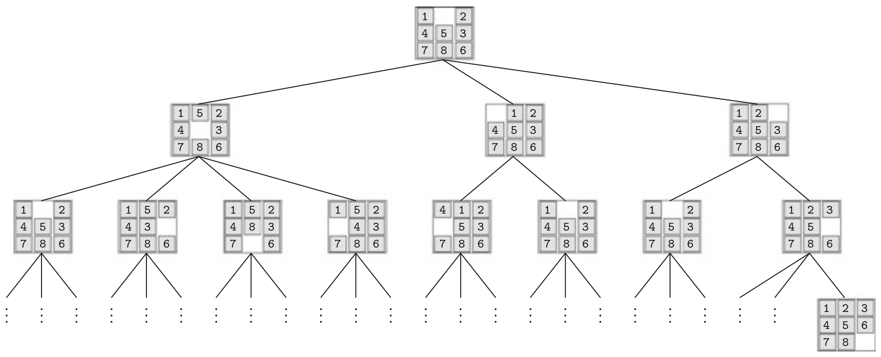


Fig. 6.3 Search tree for the 8-puzzle. *Bottom right* a goal state in depth 3 is represented. To save space the other nodes at this level have been omitted

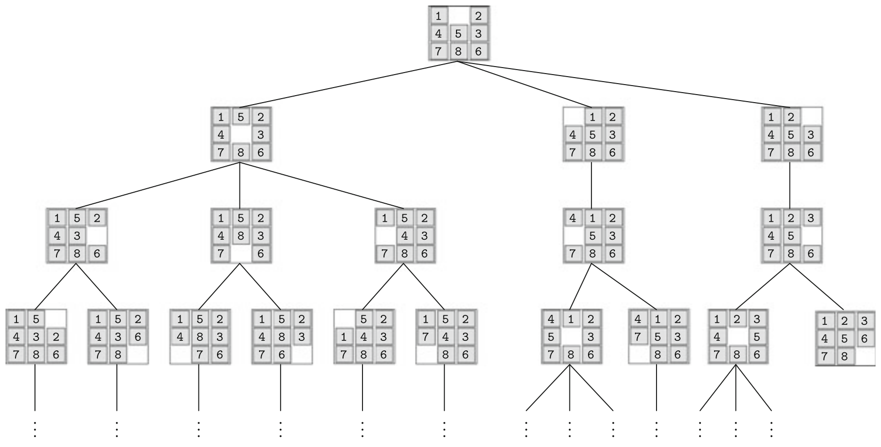


Fig. 6.4 Search tree for an 8-puzzle without cycles of length 2

that each state is repeated multiple times two levels deeper because in a simple uninformed search, every action can be reversed in the next step.

If we disallow cycles of length 2, then for the same starting state we obtain the search tree represented in Fig. 6.4. The average branching factor is reduced by about 1 and becomes 1.8.[2]

Before we begin describing the *search algorithms*, a few new terms are needed. We are dealing with discrete search problems here. Being in state s, an *action* a_1 leads to a new state s'. Thus $s' = a_1(s)$. A different action may lead to state s'', in

[2]For an 8-puzzle the average branching factor depends on the starting state (see Exercise 6.2 on page 122).

other words: $s'' = a_2(s)$. Recursive application of all possible actions to all states, beginning with the starting state, yields the *search tree*.

Definition 6.1 A search problem is defined by the following values
State: Description of the state of the world in which the search agent finds itself.
Starting state: The initial state in which the search agent is started.
Goal state: If the agent reaches a goal state, then it terminates and outputs a solution (if desired).
Actions: All of the agents allowed actions.
Solution: The path in the search tree from the starting state to the goal state.
Cost function: Assigns a cost value to every action. Necessary for finding a cost-optimal solution.
State space: Set of all states.

Applied to the 8-puzzle, we get

State: 3×3 matrix S with the values 1, 2, 3, 4, 5, 6, 7, 8 (once each) and one empty square.
Starting state: An arbitrary state.
Goal state: An arbitrary state, e.g. the state given to the right in Fig. 6.2 on page 93.
Actions: Movement of the empty square S_{ij} to the left (if $j \neq 1$), right (if $j \neq 3$), up (if $i \neq 1$), down (if $i \neq 3$).
Cost function: The constant function 1, since all actions have equal cost.
State space: The state space is degenerate in domains that are mutually unreachable (Exercise 6.4 on page 122). Thus there are unsolvable 8-puzzle problems.

For analysis of the search algorithms, the following terms are needed:

Definition 6.2
- The number of successor states of a state s is called the *branching factor* $b(s)$, or b if the branching factor is constant.
- The *effective branching factor* of a tree of depth d with n total nodes is defined as the branching factor that a tree with constant branching factor, equal depth, and equal n would have (see Exercise 6.3 on page 122).
- A search algorithm is called *complete* if it finds a solution for every solvable problem. If a complete search algorithm terminates without finding a solution, then the problem is unsolvable.

For a given depth d and node count n, the effective branching factor can be calculated by solving the equation

$$n = \frac{b^{d+1} - 1}{b - 1} \tag{6.1}$$

for b because a tree with constant branching factor and depth d has a total of

$$n = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} \tag{6.2}$$

nodes.

For the practical application of search algorithms for finite search trees, the last level is especially important because

Theorem 6.1 *For heavily branching finite search trees with a large constant branching factor, almost all nodes are on the last level.*

The simple proof of this theorem is recommended to the reader as an exercise (Exercise 6.1 on page 122).

Example 6.2 We are given a map, such as the one represented in Fig. 6.5, as a graph with cities as nodes and highway connections between the cities as weighted edges with distances. We are looking for an optimal route from city *A* to city *B*. The description of the corresponding schema reads
State: A city as the current location of the traveler.
Starting state: An arbitrary city.
Goal state: An arbitrary city.
Actions: Travel from the current city to a neighboring city.
Cost function: The distance between the cities. Each action corresponds to an edge in the graph with the distance as the weight.

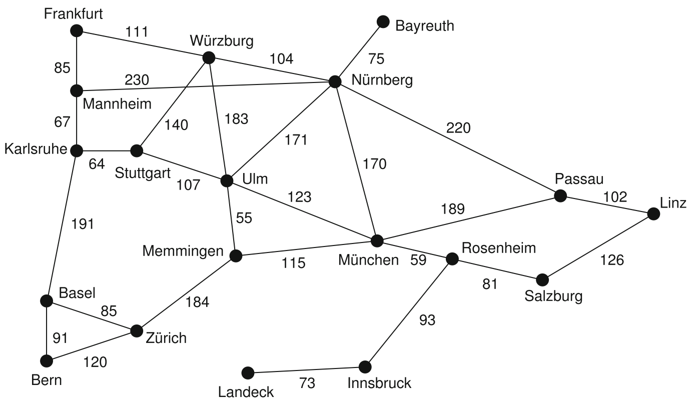


Fig. 6.5 The graph of southern Germany as an example of a search task with a cost function

State space: All cities, that is, nodes of the graph.
To find the route with minimal length, the costs must be taken into account because they are not constant as they were in the 8-puzzle.

Definition 6.3 A search algorithm is called *optimal* if it, if a solution exists, always finds the solution with the lowest cost.

The 8-puzzle problem is *deterministic*, which means that every action leads from a state to a unique successor state. It is furthermore *observable*, that is, the agent always knows which state it is in. In route planning in real applications both characteristics are not always given. The action “*Drive from Munich to Ulm*” may—for example because of an accident—lead to the successor state “*Munich*”. It can also occur that the traveler no longer knows where he is because he got lost. We want to ignore these kinds of complications at first. Therefore in this chapter we will only look at problems that are deterministic and observable.

Problems like the 8-puzzle, which are deterministic and observable, make action planning relatively simple because, due to having an abstract model, it is possible to find action sequences for the solution of the problem without actually carrying out the actions in the real world. In the case of the 8-puzzle, it is not necessary to actually move the squares in the real world to find the solution. We can find optimal solutions with so-called *offline algorithms*. One faces much different challenges when, for example, building robots that are supposed to play soccer. Here there will never be an exact abstract model of the actions. For example, a robot that kicks the ball in a specific direction cannot predict with certainty where the ball will move because, among other things, it does not know whether an opponent will catch or deflect the ball. Here *online algorithms* are then needed, which make decisions based on sensor signals in every situation. *Reinforcement learning*, described in Chap. 10, works toward optimization of these decisions based on experience.

6.2 Uninformed Search

6.2.1 Breadth-First Search

In breadth-first search, the search tree is explored from top to bottom according to the algorithm given in Fig. 6.6 on page 98 until a solution is found. First every node in the node list is tested for whether it is a goal node, and in the case of success, the program is stopped. Otherwise all successors of the node are generated. The search is then continued recursively on the list of all newly generated nodes. The whole thing repeats until no more successors are generated.

This algorithm is generic. That is, it works for arbitrary applications if the two application-specific functions “GoalReached” and “Successors” are provided.

```
BREADTHFIRSTSEARCH(NodeList, Goal)
NewNodes = $\emptyset$
For all Node $\in$ NodeList
    If GoalReached(Node, Goal)
        Return(“Solution found”, Node)
    NewNodes = Append(NewNodes, Successors(Node))
If NewNodes $\neq \emptyset$
    Return(BREADTH-FIRST-SEARCH(NewNodes, Goal))
Else
    Return(“No solution”)
```

Fig. 6.6 The algorithm for breadth-first search

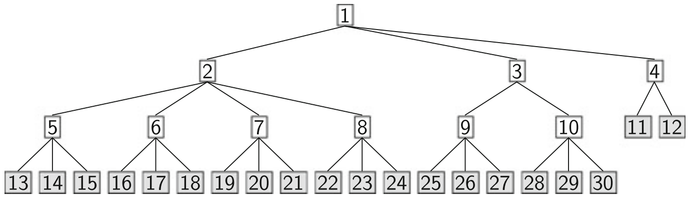


Fig. 6.7 Breadth-first search during the expansion of the third-level nodes. The nodes are numbered according to the order they were generated. The successors of nodes 11 and 12 have not yet been generated

“GoalReached” calculates whether the argument is a goal node, and “Successors” calculates the list of all successor nodes of its argument. Figure 6.7 shows a snapshot of breadth-first search.

Analysis Since breadth-first search completely searches through every depth and reaches every depth in finite time, it is complete if the branching factor b is finite. The optimal (that is, the shortest) solution is found if the costs of all actions are the same (see Exercise 6.7 on page 123). Computation time and memory space grow exponentially with the depth of the tree. For a tree with constant branching factor b and depth d, the total compute time is thus given by

$$c \cdot \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d).$$

Although only the last level is saved in memory, the memory space requirement is also $O(b^d)$.

With the speed of today's computers, which can generate billions of nodes within minutes, main memory quickly fills up and the search ends. The problem of the shortest solution not always being found can be solved by the so-called *Uniform Cost Search*, in which the node with the lowest cost from the list of nodes (which is sorted ascendingly by cost) is always expanded, and the new nodes sorted in. Thus we find the optimal solution. The memory problem is not yet solved, however. A solution for this problem is provided by depth-first search.

6.2.2 Depth-First Search

In depth-first search only a few nodes are stored in memory at one time. After the expansion of a node only its successors are saved, and the first successor node is immediately expanded. Thus the search quickly becomes very deep. Only when a node has no successors and the search fails at that depth is the next open node expanded via *backtracking* to the last branch, and so on. We can best perceive this in the elegant recursive algorithm in Fig. 6.8 and in the search tree in Fig. 6.9 on page 100.

Analysis Depth-first search requires much less memory than breadth-first search because at most b nodes are saved at each depth. Thus we need at most $b \cdot d$ memory cells.

However, depth-first search is not complete for infinitely deep trees because depth-first search runs into an infinite loop when there is no solution in the far left branch. Therefore the question of finding the optimal solution is obsolete. Because of the infinite loop, no bound on the computation time can be given. In the case of a finitely deep search tree with depth d, a total of about b^d nodes are generated. Thus the computation time grows, just as in breadth-first search, exponentially with depth.

We can make the search tree finite by setting a depth limit. Now if no solution is found in the pruned search tree, there can nonetheless be solutions outside the limit.

```
DEPTHFIRSTSEARCH(Node, Goal)
If GoalReached(Node, Goal) Return("Solution found")
NewNodes = Successors(Node)
While NewNodes ≠ ∅
    Result = DEPTH-FIRST-SEARCH(First(NewNodes), Goal)
    If Result = "Solution found" Return("Solution found")
    NewNodes = Rest(NewNodes)
Return("No solution")
```

Fig. 6.8 The algorithm for depth-first search. The function "First" returns the first element of a list, and "Rest" the rest of the list

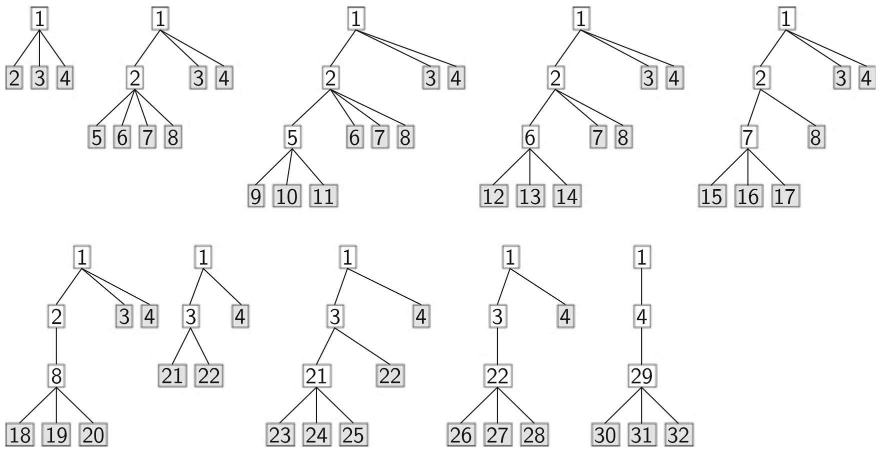


Fig. 6.9 Execution of depth-first search. All nodes at depth three are unsuccessful and cause backtracking. The nodes are numbered in the order they were generated

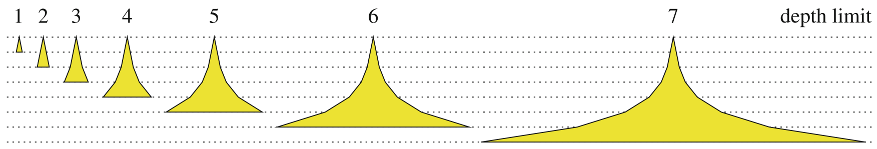


Fig. 6.10 Schematic representation of the development of the search tree in iterative deepening with limits from 1 to 7. The breadth of the tree corresponds to a branching factor of 2

Thus the search becomes incomplete. There are obvious ideas, however, for getting the search to completeness.

6.2.3 Iterative Deepening

We begin the depth-first search with a depth limit of 1. If no solution is found, we raise the limit by 1 and start searching from the beginning, and so on, as shown in Fig. 6.10. This iterative raising of the depth limit is called *iterative deepening*.

We must augment the depth-first search program given in Fig. 6.8 on page 99 with the two additional parameters "Depth" and "Limit". "Depth" is raised by one at the recursive call, and the head line of the while loop is replaced by "**While** NewNodes $\neq \emptyset$ **And** Depth < Limit". The modified algorithm is represented in Fig. 6.11 on page 101.

Analysis The memory requirement is the same as in depth-first search. One could argue that repeatedly re-starting depth-first search at depth zero causes a lot of redundant work. For large branching factors this is not the case. We now show that

```
ITERATIVEDEEPENING(Node, Goal)
DepthLimit = 0
Repeat
    Result = DEPTHFIRSTSEARCH-B(Node, Goal, 0, DepthLimit)
    DepthLimit = DepthLimit + 1
Until Result = “Solution found”
```

```
DEPTHFIRSTSEARCH-B(Node, Goal, Depth, Limit)
If GoalReached(Node, Goal) Return(“Solution found”)
NewNodes = Successors(Node)
While NewNodes $\neq \emptyset$ And Depth $<$ Limit
    Result =
        DEPTHFIRSTSEARCH-B(First(NewNodes), Goal, Depth + 1, Limit)
    If Result = “Solution found” Return(“Solution found”)
    NewNodes = Rest(NewNodes)
Return(“No solution”)
```

Fig. 6.11 The algorithm for iterative deepening, which calls the slightly modified depth-first search with a depth limit (TIEFENSUCHE-B)

the sum of the number of nodes of all depths up to the one before last $d_{max} - 1$ in all trees searched is much smaller than the number of nodes in the last tree searched.

Let $N_b(d)$ be the number of nodes of a search tree with branching factor b and depth d and d_{max} be the last depth searched. The last tree searched contains

$$N_b(d_{max}) = \sum_{i=0}^{d_{max}} b^i = \frac{b^{d_{max}+1} - 1}{b - 1}$$

nodes. All trees searched beforehand together have

$$\begin{aligned}
\sum_{d=1}^{d_{max}-1} N_b(d) &= \sum_{d=1}^{d_{max}-1} \frac{b^{d+1} - 1}{b - 1} = \frac{1}{b-1}\left(\left(\sum_{d=1}^{d_{max}-1} b^{d+1}\right) - d_{max} + 1\right) \\
&= \frac{1}{b-1}\left(\left(\sum_{d=2}^{d_{max}} b^{d}\right) - d_{max} + 1\right) \\
&= \frac{1}{b-1}\left(\frac{b^{d_{max}+1} - 1}{b - 1} - 1 - b - d_{max} + 1\right) \\
&\approx \frac{1}{b-1}\left(\frac{b^{d_{max}+1} - 1}{b - 1}\right) = \frac{1}{b-1} N_b(d_{max})
\end{aligned}$$

nodes. For $b > 2$ this is less than the number $N_b(d_{max})$ of nodes in the last tree. For $b = 20$ the first $d_{max} - 1$ trees together contain only about $\frac{1}{b-1} = 1/19$ of the number of nodes in the last tree. The computation time for all iterations besides the last can be ignored.

Just like breadth-first search, this method is complete, and given a constant cost for all actions, it finds the shortest solution.

6.2.4 Comparison

The described search algorithms have been put side-by-side in Table 6.1.

We can clearly see that iterative deepening is the winner of this test because it gets the best grade in all categories. In fact, of all four algorithms presented it is the only practically usable one.

We do indeed have a winner for this test, although for realistic applications it is usually not successful. Even for the 15-puzzle, the 8-puzzle's big brother (see Exercise 6.4 on page 122), there are about 2×10^{13} different states. For non-trivial inference systems the state space is many orders of magnitude bigger. As shown in Sect. 6.1, all the computing power in the world will not help much more. Instead what is needed is an intelligent search that only explores a tiny fraction of the search space and finds a solution there.

6.2.5 Cycle Check

As shown in Sect. 6.1, nodes may be repeatedly visited during a search. In the 8-puzzle, for example, every move can be immediately undone, which leads to unnecessary cycles of length two. Such cycles can be prevented by recording within each node all of its predecessors and, when expanding a node, comparing the newly created successor nodes with the predecessor nodes. All of the duplicates found can be removed from the list of successor nodes. This simple check costs only a small constant factor of additional memory space and increases the constant computation time c by an additional constant δ for the check itself for a total of $c + \delta$. This overhead for the cycle check is (hopefully) offset by a reduction in the cost of the

Table 6.1 Comparison of the uninformed search algorithms. (*) means that the statement is only true given a constant action cost. d_s is the maximal depth for a finite search tree

	Breadth-first search	Uniform cost search	Depth-first search	Iterative deepening
Completeness	Yes	Yes	No	Yes
Optimal solution	Yes (*)	Yes	No	Yes (*)
Computation time	b^d	b^d	∞ or b^d	b^d
Memory use	$\boldsymbol{b^d}$	$\boldsymbol{b^d}$	$\boldsymbol{bd}$	$\boldsymbol{bd}$

search. The reduction depends, of course, on the particular application and therefore cannot be given in general terms.

For the 8-puzzle we obtain the result as follows. If, for example, during breadth-first search with effective branching factor b on a finite tree of depth d, the computation time without the cycle check is $c \cdot b^d$, the required time with the cycle check becomes

$$(c + \delta) \cdot (b - 1)^d.$$

The check thus practically always results in a clear gain because reducing the branching factor by one has an exponentially growing effect as the depth increases, whereas the additional computation time δ only somewhat increases the constant factor.

Now the question arises as to how a check on cycles of arbitrary length would affect the search performance. The list of all predecessors must now be stored for each node, which can be done very efficiently (see Exercise 6.8 on page 123). During the search, each newly created node must now be compared with all its predecessors. The computation time of depth-first search or breadth-first search is given by

$$c_1 \cdot \sum_{i=0}^{d} b^i + c_2 \cdot \sum_{i=0}^{d} i \cdot b^i.$$

Here, the first term is the already-known cost of generating the nodes, and the second term is the cost of the cycle check. We can show that for large values of b and d,

$$\sum_{i=0}^{d} i \cdot b^i \approx d \cdot b^d.$$

The complexity of the search with the full cycle check therefore only increases by a factor of d faster than for the search without a cycle check. In search trees that are not very deep, this extra complexity is not important. For search tasks with very deep, weakly branching trees, it may be advantageous to use a hash table [CLR90] to store the list of predecessors. Lookups in the table can be done in constant time such that the computation time of the search algorithm only grows by a small constant factor.

In summary, we can conclude that the cycle check implies hardly any additional overhead and is therefore worthwhile for applications with repeatedly occurring nodes.

6.3 Heuristic Search

Heuristics are problem-solving strategies which in many cases find a solution faster than uninformed search. However, this is not guaranteed. Heuristic search could require a lot more time and can even result in the solution not being found.

We humans successfully use heuristic processes for all kinds of things. When buying vegetables at the supermarket, for example, we judge the various options for a pound of strawberries using only a few simple criteria like price, appearance, source of production, and trust in the seller, and then we decide on the best option by gut feeling. It might theoretically be better to subject the strawberries to a basic chemical analysis before deciding whether to buy them. For example, the strawberries might be poisoned. If that were the case the analysis would have been worth the trouble. However, we do not carry out this kind of analysis because there is a very high probability that our heuristic selection will succeed and will quickly get us to our goal of eating tasty strawberries.

Heuristic decisions are closely linked with the need to make *real-time decisions with limited resources*. In practice a good solution found quickly is preferred over a solution that is optimal, but very expensive to derive.

A heuristic evaluation function $f(s)$ for states is used to mathematically model a heuristic. The goal is to find, with little effort, a solution to the stated search problem with minimal total cost. Please note that there is a subtle difference between the effort to find a solution and the total cost of this solution. For example it may take Google Maps half a second's worth of effort to find a route from the City Hall in San Francisco to Tuolumne Meadows in Yosemite National Park, but the ride from San Francisco to Tuolumne Meadows by car may take four hours and some money for gasoline etc. (total cost).

Next we will modify the breadth-first search algorithm by adding the evaluation function to it. The currently open nodes are no longer expanded left to right by row, but rather according to their heuristic rating. From the set of open nodes, the node with the minimal rating is always expanded first. This is achieved by immediately evaluating nodes as they are expanded and sorting them into the list of open nodes. The list may then contain nodes from different depths in the tree.

Because heuristic evaluation of states is very important for the search, we will differentiate from now on between states and their associated nodes. The node contains the state and further information relevant to the search, such as its depth in the search tree and the heuristic rating of the state. As a result, the function "Successors", which generates the successors (children) of a node, must also immediately calculate for these successor nodes their heuristic ratings as a component of each node. We define the general search algorithm HEURISTICSEARCH in Fig. 6.12 on page 105.

The node list is initialized with the starting nodes. Then, in the loop, the first node from the list is removed and tested for whether it is a solution node. If not, it will be expanded with the function "Successors" and its successors added to the list with the function "SortIn". "SortIn(X,Y)" inserts the elements from the unsorted list X into the ascendingly sorted list Y. The heuristic rating is used as the sorting key. Thus it is guaranteed that the best node (that is, the one with the lowest heuristic value) is always at the beginning of the list.[3]

[3]When sorting in a new node from the node list, it may be advantageous to check whether the node is already available and, if so, to delete the duplicate.

```
HEURISTICSEARCH(Start, Goal)

NodeList = [Start]
While True
    If NodeList = ∅ Return("No solution")
    Node = First(NodeList)
    NodeList = Rest(NodeList)
    If GoalReached(Node, Goal) Return("Solution found", Node)
    NodeList = SortIn(Successors(Node),NodeList)
```

Fig. 6.12 The algorithm for heuristic search

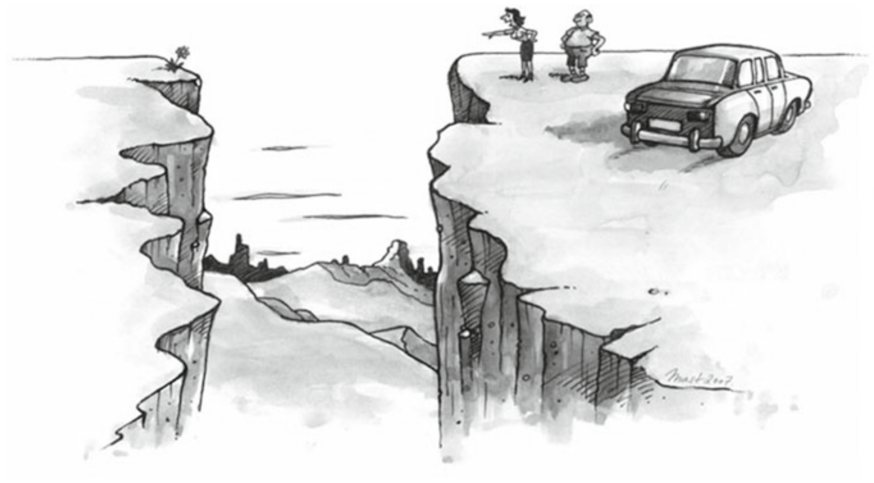


Fig. 6.13 He: "Dear, think of the fuel cost! I'll pluck one for you somewhere else." She: "No, I want that one over there!"

Depth-first and breadth-first search also happen to be special cases of the function HEURISTICSEARCH. We can easily generate them by plugging in the appropriate evaluation function (Exercise 6.11 on page 123).

The best heuristic would be a function that calculates the actual costs from each node to the goal. To do that, however, would require a traversal of the entire search space, which is exactly what the heuristic is supposed to prevent. Therefore we need a heuristic that is fast and simple to compute. How do we find such a heuristic?

An interesting idea for finding a heuristic is simplification of the problem. The original task is simplified enough that it can be solved with little computational cost. The costs from a state to the goal in the simplified problem then serve as an estimate for the actual problem (see Fig. 6.13). This *cost estimate function* we denote h.

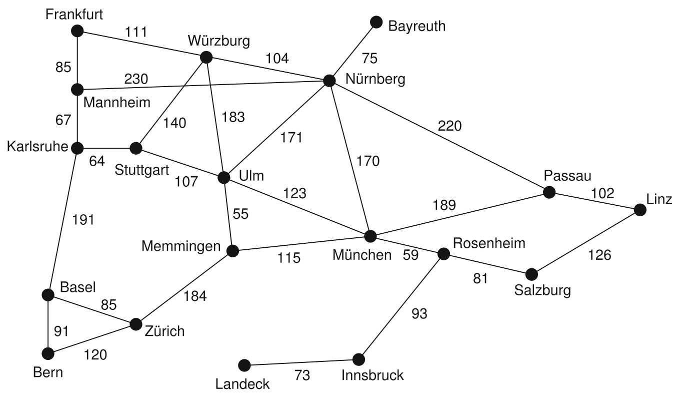
6.3.1 Greedy Search

It seems sensible to choose the state with the lowest estimated h value (that is, the one with the lowest estimated cost) from the list of currently available states. The cost estimate then can be used directly as the evaluation function. For the evaluation in the function HeuristicSearch we set $f(s) = h(s)$. This can be seen clearly in the trip planning example (Example 6.2 on page 96). We set up the task of finding the straight line path from city to city (that is, the flying distance) as a simplification of the problem. Instead of searching the optimal route, we first determine from every node a route with minimal flying distance to the goal. We choose Ulm as the destination. Thus the cost estimate function becomes

$$h(s) = \text{flying distance from city } s \text{ to Ulm.}$$

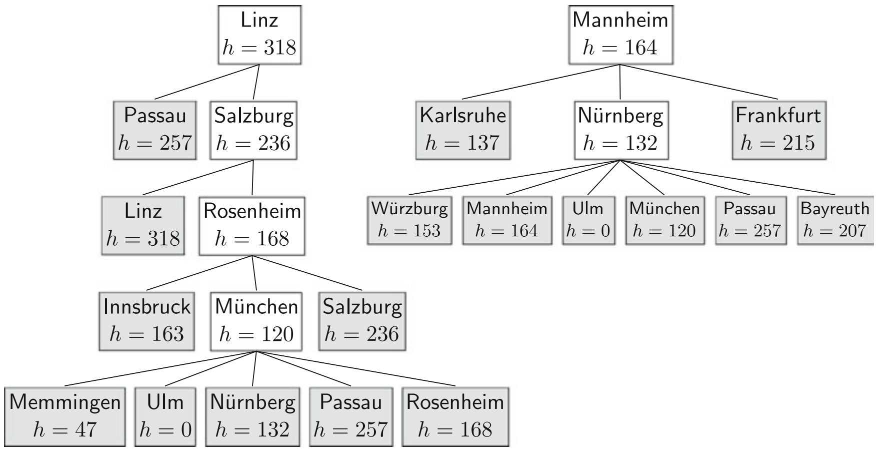
The flying distances from all cities to Ulm are given in Fig. 6.14 next to the graph.

The search tree for starting in Linz is represented in Fig. 6.15 on page 107 left. We can see that the tree is very slender. The search thus finishes quickly. Unfortunately, this search does not always find the optimal solution. For example, this algorithm fails to find the optimal solution when starting in Mannheim (Fig. 6.15 on page 107 right). The Mannheim–Nürnberg–Ulm path has a length of 401 km. The route Mannheim–Karlsruhe–Stuttgart–Ulm would be significantly shorter at 238 km. As we observe the graph, the cause of this problem becomes clear. Nürnberg is in fact somewhat closer than Karlsruhe to Ulm, but the distance from Mannheim to Nürnberg is significantly greater than that from Mannheim to Karlsruhe. The heuristic only looks ahead "greedily" to the goal instead of also taking into account the stretch that has already been laid down to the current node. This is why we give it the name *greedy search*.



Basel	204
Bayreuth	207
Bern	247
Frankfurt	215
Innsbruck	163
Karlsruhe	137
Landeck	143
Linz	318
München	120
Mannheim	164
Memmingen	47
Nürnberg	132
Passau	257
Rosenheim	168
Stuttgart	75
Salzburg	236
Würzburg	153
Zürich	157

Fig. 6.14 City graph with flying distances from all cities to Ulm



Node	München	Innsbruck	Salzburg	Passau	Linz
heuristic rating	120	163	236	257	318

Fig. 6.15 Greedy search: from Linz to Ulm (*left*) and from Mannheim to Ulm (*right*). The node list data structure for the left search tree, sorted by the node rating before the expansion of the node München is given

6.3.2 A★-Search

We now want to take into account the costs that have accrued during the search up to the current node s. First we define the *cost function*

$$g(s) = \text{Sum of accrued costs from the start to the current node,}$$

then add to that the estimated cost to the goal and obtain as the *heuristic evaluation function*

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

Now we add yet another small, but important requirement.

Definition 6.4 A heuristic cost estimate function $h(s)$ that never overestimates the actual cost from state s to the goal is called *admissible*.

The function HEURISTICSEARCH together with an evaluation function $f(s) = g(s) + h(s)$ and an admissible heuristic function h is called A★-*algorithm*. This famous algorithm is complete and optimal. A★ thus always finds the shortest solution for every solvable search problem. We will explain and prove this in the following discussion.

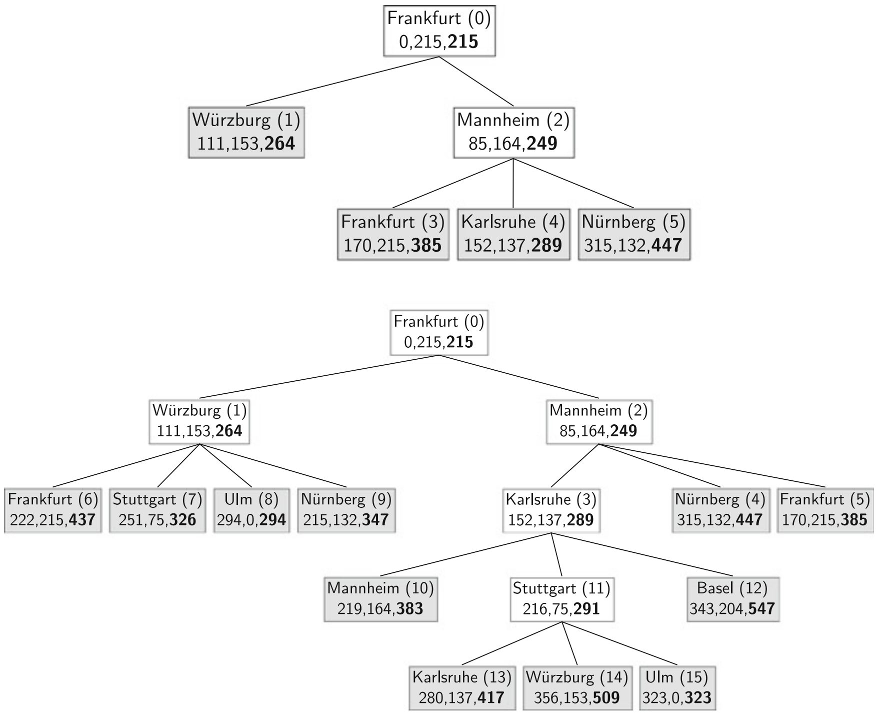


Fig. 6.16 Two snapshots of the A* search tree for the optimal route from Frankfurt to Ulm. In the boxes below the name of the city s we show $g(s)$, $h(s)$, $f(s)$. Numbers in parentheses after the city names show the order in which the nodes have been generated by the "Successor" function

First we apply the A*-algorithm to the example. We are looking for the shortest path from Frankfurt to Ulm.

In the top part of Fig. 6.16 we see that the successors of Mannheim are generated before the successors of Würzburg. The optimal solution Frankfurt–Würzburg–Ulm is generated shortly thereafter in the eighth step, but it is not yet recognized as such. Thus the algorithm does not terminate yet because the node Karlsruhe (3) has a better (lower) f value and thus is ahead of the node Ulm (8) in line. Only when all f values are greater than or equal to that of the solution node Ulm (8) have we ensured that we have an optimal solution. Otherwise there could potentially be another solution with lower costs. We will now show that this is true generally.

Theorem 6.2 *The A* algorithm is optimal. That is, it always finds the solution with the lowest total cost if the heuristic h is admissible.*

Proof In the HEURISTICSEARCH algorithm, every newly generated node s is sorted in by the function "SortIn" according to its heuristic rating $f(s)$. The node with the

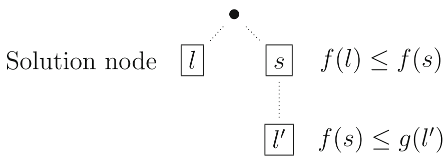


Fig. 6.17 The first solution node l found by A* never has a higher cost than another arbitrary node l'

smallest rating value thus is at the beginning of the list. If the node l at the beginning of the list is a solution node, then no other node has a better heuristic rating. For all other nodes s it is true then that $f(l) \leq f(s)$. Because the heuristic is admissible, no better solution l' can be found, even after expansion of all other nodes (see Fig. 6.17). Written formally:

$$g(l) = g(l) + h(l) = f(l) \leq f(s) = g(s) + h(s) \leq g(l').$$

The first equality holds because l is a solution node with $h(l) = 0$. The second is the definition of f. The third (in)equality holds because the list of open nodes is sorted in ascending order. The fourth equality is again the definition of f. Finally, the last (in)equality is the admissibility of the heuristic, which never overestimates the cost from node s to an arbitrary solution. Thus it has been shown that $g(l) \leq g(l')$, that is, that the discovered solution l is optimal. □

6.3.3 Route Planning with the A* Search Algorithm

Many current car navigation systems use the A* algorithm. The simplest, but very good heuristic for computing A* is the straight-line distance from the current node to the destination. The use of 5 to 60 so-called landmarks is somewhat better. For these randomly chosen points the shortest paths to and from all nodes on the map are calculated in a precomputation step. Let l be such a landmark, s the current node, and z the destination node. Also let $c^\star(x, y)$ be the cost of the shortest path from x to y. Then we obtain for the shortest path from s to l the triangle inequality (see Exercise 6.11 on page 123)

$$c^\star(s,l) \leq c^\star(s,z) + c^\star(z,l).$$

Solving for $c^\star(s, z)$ results in the admissible heuristic

$$h(s) = c^\star(s,l) - c^\star(z,l) \leq c^\star(s,z).$$

In [Bat16], it was shown that this heuristic is better than the straight-line distance for route planning. On one hand, it can be calculated faster than the straight-line distance. Due to precomputation, distances to the landmarks can be quickly retrieved from an array, whereas the Euclidean distances must be computed individually. It turns out that the landmark heuristic shrinks the search space even more. This can be seen in the left image of Fig. 6.18, which illustrates the search tree of $A^\star$ search for planning a route from Ravensburg to Biberach (two towns in southern Germany).[4] The edges without any heuristic (i.e. with $h(s) = 0$) are plotted in red colour, dark green lines show the search tree using the straight-line distance heuristic, and the edges of the landmark heuristic with twenty landmarks are plotted in blue.

The right image shows the same route using bidirectional search, where a route from Ravensburg to Biberach and one in the opposite direction are planned effectively in parallel. If the routes meet, given certain conditions of the heuristic, an optimal route has been found [Bat16]. A quantitative analysis of the search tree sizes and the computation time on a PC can be found in Table 6.2.

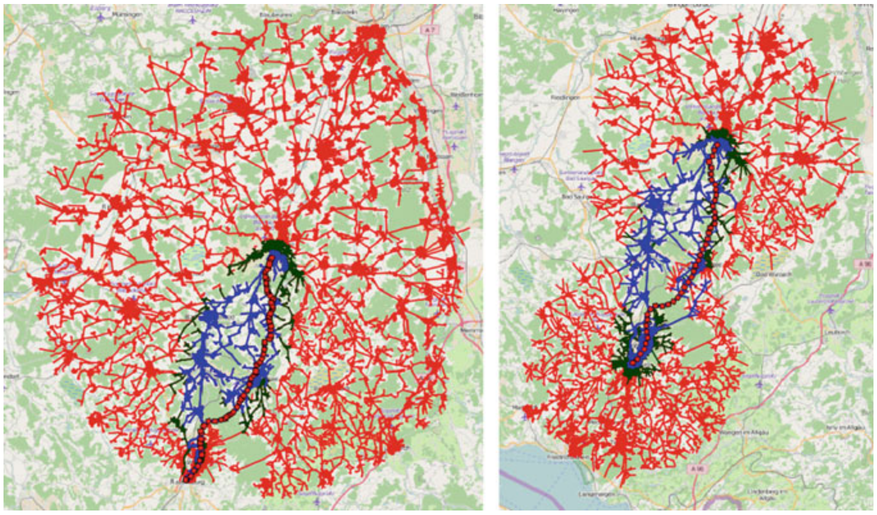


Fig. 6.18 $A^\star$ search tree without heuristic (red), with straight-line distance (dark green) and with landmarks (blue). The left image shows unidirectional search and the right shows bidirectional search. Note that the green edges are covered by blue and the red edges are covered by green and blue

Table 6.2 Comparison of search tree size and computation time for route planning with and without each of the two heuristics. The landmark heuristic is the clear winner

	Unidirectional		Bidirectional	
	Tree Size [nodes]	Comp. time [msec.]	Tree Size [nodes]	Comp. time [msec.]
No heuristic	62000	192	41850	122
Straight-line distance	9380	86	12193	84
Landmark heuristic	5260	16	7290	16

[4]Both graphs in Fig. 6.18 were generated by A. Batzill using the system described in [Bat16].

Observing unidirectional search, we see that both heuristics clearly reduce the search space. The computation times are truly interesting. In the case of the landmark heuristic, we see the computation time and the size of the search space reduced by a factor of about 12. The cost of computing the heuristic is thus insignificant. The straight-line distance, however, results in a search space reduction of a factor of 6.6, but only an improvement of a factor of 2.2 in run time due to the overhead of computing the euclidean distance.

In the case of bidirectional search, in contrast to unidirectional search, we see a significant reduction of the search space even without heuristic. On the other hand, the search space is larger than the unidirectional case for both heuristics. However, because the nodes are partitioned into two sorted lists in bidirectional search (see HeuristicSearch function in Fig. 6.12 on page 105), the lists are handled faster and the resulting computation times are roughly the same [Bat16].

When planning a route, usually the driver cares more about driving time than the distance driven. We should thus adjust the heuristic accordingly and replace straight-line distance $d(s, z)$ with time $t(s,z) = d(s,z)/v_{max}$. Here we have to divide by the maximum average velocity, which degrades the heuristic because it causes the heuristically estimated times to be much too small. The landmark heuristic, in contrast, builds on precomputed optimal routes and therefore does not degrade. Thus, as shown in [Bat16], the search for a time-optimized route using landmark heuristic is significantly faster than with the modified straight-line distance.

The *contraction hierarchies* algorithm performs even better than A* with landmark heuristic. It is based on the idea of combining, in a precompution step, several edges into so-called *shortcuts*, which are then used to reduce the search space [GSSD08, Bat16].

6.3.4 IDA*-Search

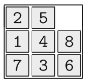
The A* search inherits a quirk from breadth-first search. It has to save many nodes in memory, which can lead to very high memory use. Furthermore, the list of open nodes must be sorted. Thus insertion of nodes into the list and removal of nodes from the list can no longer run in constant time, which increases the algorithm's complexity slightly. Based on the heapsort algorithm, we can structure the node list as a heap with logarithmic time complexity for insertion and removal of nodes (see [CLR90]).

Both problems can be solved—similarly to breadth-first search—by iterative deepening. We work with depth-first search and successively raise the limit. However, rather than working with a depth limit, here we use a limit for the heuristic evaluation $f(s)$. This process is called the IDA*-algorithm.

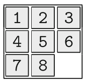
6.3.5 Empirical Comparison of the Search Algorithms

In A*, or (alternatively) IDA*, we have a search algorithm with many good properties. It is complete and optimal. It can thus be used without risk. The most important thing, however, is that it works with heuristics, and therefore can significantly reduce the computation time needed to find a solution. We would like to explore this empirically in the 8-puzzle example.

For the 8-puzzle there are two simple admissible heuristics. The heuristic h_1 simply counts the number of squares that are not in the right place. Clearly this heuristic is admissible. Heuristic h_2 measures the *Manhattan distance*. For every square the horizontal and vertical distances to that square's location in the goal state are added together. This value is then summed over all squares. For example, the Manhattan distance of the two states



and



is calculated as

$$h_2(s) = 1 + 1 + 1 + 1 + 2 + 0 + 3 + 1 = 10.$$

The admissibility of the Manhattan distance is also obvious (see Exercise 6.13 on page 123).

The described algorithms were implemented in Mathematica. For a comparison with uninformed search, the A* algorithm with the two heuristics h_1 and h_2 and iterative deepening was applied to 132 randomly generated 8-puzzle problems. The average values for the number of steps and computation time are given in Table 6.3. We see that the heuristics significantly reduce the search cost compared to uninformed search.

If we compare iterative deepening to A* with h_1 at depth 12, for example, it becomes evident that h_1 reduces the number of steps by a factor of about 3,000, but

Table 6.3 Comparison of the computation cost of uninformed search and heuristic search for solvable 8-puzzle problems with various depths. Measurements are in steps and seconds. All values are averages over multiple runs (see last column)

Depth	Iterative deepening		A* algorithm				Num. runs
			Heuristic h_1		Heuristic h_2		
	Steps	Time [sec]	Steps	Time [sec]	Steps	Time [sec]	
2	20	0.003	3.0	0.0010	3.0	0.0010	10
4	81	0.013	5.2	0.0015	5.0	0.0022	24
6	806	0.13	10.2	0.0034	8.3	0.0039	19
8	6455	1.0	17.3	0.0060	12.2	0.0063	14
10	50512	7.9	48.1	0.018	22.1	0.011	15
12	486751	75.7	162.2	0.074	56.0	0.031	12
			IDA*				
14	–	–	10079.2	2.6	855.6	0.25	16
16	–	–	69386.6	19.0	3806.5	1.3	13
18	–	–	708780.0	161.6	53941.5	14.1	4

the computation time by only a factor of 1,023. This is due to the higher cost per step for the computation of the heuristic.

Closer examination reveals a jump in the number of steps between depth 12 and depth 14 in the column for h_1. This jump cannot be explained solely by the repeated work done by IDA*. It comes about because the implementation of the A* algorithm deletes duplicates of identical nodes and thereby shrinks the search space. This is not possible with IDA* because it saves almost no nodes. Despite this, A* can no longer compete with IDA* beyond depth 14 because the cost of sorting in new nodes pushes up the time per step so much.

A computation of the effective branching factor according to (6.1) on page 96 yields values of about 2.8 for uninformed search. This number is consistent with the value from Sect. 6.1. Heuristic h_1 reduces the branching factor to values of about 1.5 and h_2 to about 1.3. We can see in the table that a small reduction of the branching factor from 1.5 to 1.3 gives us a big advantage in computation time.

Heuristic search thus has an important practical significance because it can solve problems which are far out of reach for uninformed search.

6.3.6 Summary

Of the various search algorithms for uninformed search, iterative deepening is the only practical one because it is complete and can get by with very little memory. However, for difficult combinatorial search problems, even iterative deepening usually fails due to the size of the search space. Heuristic search helps here through its reduction of the effective branching factor. The IDA*-algorithm, like iterative deepening, is complete and requires very little memory.

Heuristics naturally only give a significant advantage if the heuristic is "good". When solving difficult search problems, the developer's actual task consists of designing heuristics which greatly reduce the effective branching factor. In Sect. 6.5 we will deal with this problem and also show how machine learning techniques can be used to automatically generate heuristics.

In closing, it remains to note that heuristics have no performance advantage for unsolvable problems because the unsolvability of a problem can only be established when the complete search tree has been searched through. For decidable problems such as the 8-puzzle this means that the whole search tree must be traversed up to a maximal depth whether a heuristic is being used or not. The heuristic is always a disadvantage in this case, attributable to the computational cost of evaluating the heuristic. This disadvantage can usually be estimated by a constant factor independent of the size of the problem. For undecidable problems such as the proof of PL1 formulas, the search tree can be infinitely deep. This means that, in the unsolvable case, the search potentially never ends. In summary we can say the following: for solvable problems, heuristics often reduce computation time dramatically, but for unsolvable problems the cost can even be higher with heuristics.

6.4 Games with Opponents

Games for two players, such as chess, checkers, Othello, and Go are deterministic because every action (a move) results in the same child state given the same parent state. In contrast, backgammon is non-deterministic because its child state depends on the result of a dice roll. These games are all observable because every player always knows the complete game state. Many card games, such as poker, for example, are only partially observable because the player does not know the other players' cards, or only has partial knowledge about them.

The problems discussed so far in this chapter were deterministic and observable. In the following we will look at games which, too, are deterministic and observable. Furthermore, we will limit ourselves to zero-sum games. These are games in which every gain one player makes means a loss of the same value for the opponent. The sum of the gain and loss is always equal to zero. This is true of the games chess, checkers, Othello, and Go, mentioned above.

6.4.1 Minimax Search

The goal of each player is to make optimal moves that result in victory. In principle it is possible to construct a search tree and completely search through it (like with the 8-puzzle) for a series of moves that will result in victory. However, there are several peculiarities to watch out for:

1. The effective branching factor in chess is around 30 to 35. In a typical game with 50 moves per player, the search tree has more than $30^{100} \approx 10^{148}$ leaf nodes. Thus there is no chance to fully explore the search tree. Additionally, chess is often played with a time limit. Because of this *real-time requirement*, the search must be limited to an appropriate depth in the tree, for example eight half-moves. Since among the leaf nodes of this depth-limited tree there are normally no solution nodes (that is, nodes which terminate the game) a heuristic *evaluation function B* for board positions is used. The level of play of the program strongly depends on the quality of this evaluation function. Therefore we will further treat this subject in Sect. 6.5.
2. In the following we will call the player whose game we wish to optimize Max, and his opponent Min. The opponent's (Min's) moves are not known in advance, and thus neither is the actual search tree. This problem can be elegantly solved by assuming that the opponent always makes the best move he can. The higher the evaluation $B(s)$ for position s, the better position s is for the player Max and the worse it is for his opponent Min. Max tries to maximize the evaluation of his moves, whereas Min makes moves that result in as low an evaluation as possible.

A search tree with four half-moves and evaluations of all leaves is given in Fig. 6.19 on page 115. The evaluation of an inner node is derived recursively as the maximum or minimum of its child nodes, depending on the node's level.

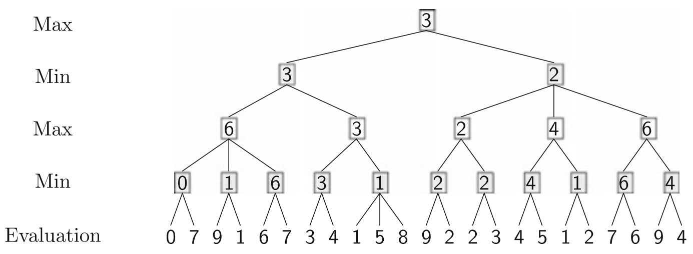


Fig. 6.19 A minimax game tree with look-ahead of four half-moves

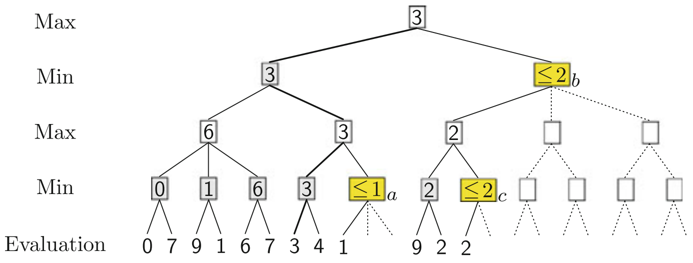


Fig. 6.20 An alpha-beta game tree with look-ahead of four half-moves. The dotted portions of the tree are not traversed because they have no effect on the end result

6.4.2 Alpha-Beta-Pruning

By switching between maximization and minimization, we can save ourselves a lot of work in some circumstances. Alpha-beta pruning works with depth-first search up to a preset depth limit. In this way the search tree is searched through from left to right. Like in minimax search, in the minimum nodes the minimum is generated from the minimum value of the successor nodes and in the maximum nodes likewise the maximum. In Fig. 6.20 this process is depicted for the tree from Fig. 6.19. At the node marked a, all other successors can be ignored after the first child is evaluated as the value 1 because the minimum is sure to be ≤ 1. It could even become smaller still, but that is irrelevant since the maximum is already ≥ 3 one level above. Regardless of how the evaluation of the remaining successors turns out, the maximum will keep the value 3. Analogously the tree will be trimmed at node b. Since the first child of b has the value 2, the minimum to be generated for b can only be less than or equal to 2. But the maximum at the root node is already sure to be ≥ 3. This cannot be changed by values ≤ 2. Thus the remaining subtrees of b can be pruned.

The same reasoning applies for the node c. However, the relevant maximum node is not the direct parent, but the root node. This can be generalized.

```
AlphaBetaMax(Node, $\alpha$, $\beta$)
If DepthLimitReached(Node) Return(Rating(Node))
NewNodes = Successors(Node)
While NewNodes $\neq \emptyset$
    $\alpha$ = Maximum($\alpha$, AlphaBetaMin(First(NewNodes), $\alpha$, $\beta$))
    If $\alpha \geq \beta$ Return($\beta$)
    NewNodes = Rest(NewNodes)
Return($\alpha$)
```

```
AlphaBetaMin(Node, $\alpha$, $\beta$)
If DepthLimitReached(Node) Return(Rating(Node))
NewNodes = Successors(Node)
While NewNodes $\neq \emptyset$
    $\beta$ = Minimum($\beta$, AlphaBetaMax(First(NewNodes), $\alpha$, $\beta$))
    If $\beta \leq \alpha$ Return($\alpha$)
    NewNodes = Rest(NewNodes)
Return($\beta$)
```

Fig. 6.21 The algorithm for alpha-beta search with the two functions AlphaBetaMin and AlphaBetaMax

- At every leaf node the evaluation is calculated.
- For every maximum node the current largest child value is saved in α.
- For every minimum node the current smallest child value is saved in β.
- If at a minimum node k the current value $\beta \leq \alpha$, then the search under k can end. Here α is the largest value of a maximum node in the path from the root to k.
- If at a maximum node l the current value $\alpha \geq \beta$, then the search under l can end. Here β is the smallest value of a minimum node in the path from the root to l.

The algorithm given in Fig. 6.21 is an extension of depth-first search with two functions which are called in alternation. It uses the values defined above for α and β.

The initial alpha-beta pruning call is done with the command
AlphaBetaMax(RootNode, $-\infty$, ∞).

Complexity The computation time saved by alpha-beta pruning heavily depends on the order in which child nodes are traversed. In the worst case, alpha-beta pruning does not offer any advantage. For a constant branching factor b the number n_d of leaf nodes to evaluate at depth d is equal to

$$n_d = b^d.$$

In the best case, when the successors of maximum nodes are descendingly sorted and the successors of minimum nodes are ascendingly sorted, the effective branching factor is reduced to $\sqrt{b}$. In chess this means a substantial reduction of the effective branching factor from 35 to about 6. Then only

$$n_d = \sqrt{b}^d = b^{d/2}$$

leaf nodes would be created. This means that the depth limit and thus also the search horizon are doubled with alpha-beta pruning. However, this is only true in the case of optimally sorted successors because the child nodes' ratings are unknown at the time when they are created. If the child nodes are randomly sorted, then the branching factor is reduced to $b^{3/4}$ and the number of leaf nodes to

$$n_d = b^{\frac{3}{4}d}.$$

With the same computing power a chess computer using alpha-beta pruning can, for example, compute eight half-moves ahead instead of six, with an effective branching factor of about 14. A thorough analysis with a derivation of these parameters can be found in [Pea84].

To double the search depth as mentioned above, we would need the child nodes to be optimally ordered, which is not the case in practice. Otherwise the search would be unnecessary. With a simple trick we can get a relatively good node ordering. We connect alpha-beta pruning with iterative deepening over the depth limit. Thus at every new depth limit we can access the ratings of all nodes of previous levels and order the successors at every branch. Thereby we reach an effective branching factor of roughly 7 to 8, which is not far from the theoretical optimum of $\sqrt{35}$ [Nil98].

6.4.3 Non-deterministic Games

Minimax search can be generalized to all games with non-deterministic actions, such as backgammon. Each player rolls before his move, which is influenced by the result of the dice roll. In the game tree there are now therefore three types of levels in the sequence

Max, dice, Min, dice, … ,

where each dice roll node branches six ways. Because we cannot predict the value of the die, we average the values of all rolls and conduct the search as described with the average values from [RN10].

6.5 Heuristic Evaluation Functions

How do we find a good heuristic evaluation function for the task of searching? Here there are fundamentally two approaches. The classical way uses the knowledge of human experts. The knowledge engineer is given the usually difficult task of formalizing the expert's implicit knowledge in the form of a computer program. We now want to show how this process can be simplified in the chess program example.

In the first step, experts are questioned about the most important factors in the selection of a move. Then it is attempted to quantify these factors. We obtain a list of relevant features or attributes. These are then (in the simplest case) combined into a linear evaluation function $B(s)$ for positions, which could look like:

$$\begin{aligned} B(s) = {} & a_1 \cdot \text{material} + a_2 \cdot \text{pawn\_structure} + a_3 \cdot \text{king\_safety} \\ & + a_4 \cdot \text{knight\_in\_center} + a_5 \cdot \text{bishop\_diagonal\_coverage} + \cdots, \end{aligned} \tag{6.3}$$

where "material" is by far the most important feature and is calculated by

$$\text{material} = \text{material(own\_team)} - \text{material(opponent)}$$

with

$$\begin{aligned} \text{material(team)} = {} & \text{num\_pawns(team)} \cdot 100 + \text{num\_knights(team)} \cdot 300 \\ & + \text{num\_bishops(team)} \cdot 300 + \text{num\_rooks(team)} \cdot 500 \\ & + \text{num\_queens(team)} \cdot 900 \end{aligned}$$

Nearly all chess programs make a similar evaluation for material. However, there are big differences for all other features, which we will not go into here [Fra05, Lar00].

In the next step the weights a_i of all features must be determined. These are set intuitively after discussion with experts, then changed after each game based on positive and negative experience. The fact that this optimization process is very expensive and furthermore that the linear combination of features is very limited suggests the use of machine learning.

6.5.1 Learning of Heuristics

We now want to automatically optimize the weights a_i of the evaluation function B(s) from (6.3). In this approach the expert is only asked about the relevant features $f_1(s), \ldots, f_n(s)$ for game state s. Then a machine learning process is used with the goal of finding an evaluation function that is as close to optimal as possible. We start with an initial pre-set evaluation function (determined by the learning process), and then let the chess program play. At the end of the game a rating is derived from the result (victory, defeat, or draw). Based on this rating, the evaluation function is

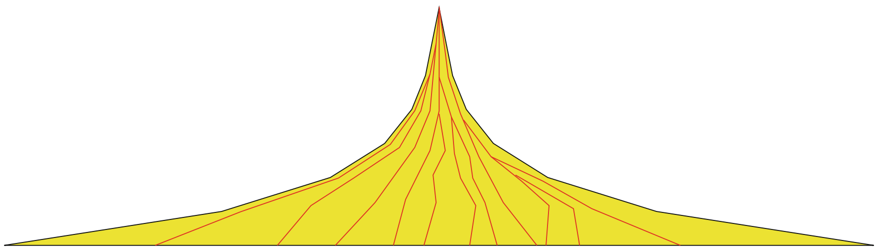


Fig. 6.22 In this sketch of a search tree, several MCTS paths to leaf nodes are shown in red. Notice that only a small part of the tree is searched

changed with the goal of making fewer mistakes next time. In principle, the same thing that is done by the developer is now being taken care of automatically by the learning process.

As easy as this sounds, it is very difficult in practice. A central problem with improving the position rating based on won or lost matches is known today as the *credit assignment* problem. We do in fact have a rating at the end of the game, but no ratings for the individual moves. Thus the agent carries out many actions but does not receive any positive or negative feedback until the very end. How should it then assign this feedback to the many actions taken in the past? And how should it improve its actions in that case? The exciting field of **reinforcement learning** deals with these questions (see Chap. 10).

Monte Carlo tree search (MCTS) [KS06] works quite similarly. To improve the heuristic rating of a game state s, a random number of search tree branches starting from this state are either explored to the end and evaluated, or stopped at a certain depth and then the leaf nodes are evaluated heuristically. The evaluation $B(s)$ of state s is given as the mean of all leaf node scores. The use of MCTS paths requires only a small part of the entire exponentially exploding tree to be searched. This is illustrated in Fig. 6.22. For many computer-simulated games, such as chess, this algorithm can be used to achieve better play for the same computational effort or to reduce computational effort for the same difficulty level [KS06]. This method was used together with machine learning algorithms in 2016 by the program AlphaGo, described in Sect. 10.10, which was the first Go program to defeat world-class human players [SHM+16].

6.6 State of the Art

For evaluation of the quality of the heuristic search processes, I would like to repeat Elaine Rich's definition [Ric83]:

Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.

There is hardly a better suited test for deciding whether a computer program is intelligent as the direct comparison of computer and human in a game like chess, checkers, backgammon or Go.

In 1950, Claude Shannon, Konrad Zuse, and John von Neumann introduced the first chess programs, which, however, could either not be implemented or would take a great deal of time to implement. Just a few years later, in 1955, Arthur Samuel wrote a program that played checkers and could improve its own parameters through a simple learning process. To do this he used the first programmable logic computer, the IBM 701. Compared to the chess computers of today, however, it had access to a large number of archived games, for which every individual move had been rated by experts. Thus the program improved its evaluation function. To achieve further improvements, Samuel had his program play against itself. He solved the credit assignment problem in a simple manner. For each individual position during a game it compares the evaluation by the function $B(s)$ with the one calculated by alpha-beta pruning and changes $B(s)$ accordingly. In 1961 his checkers program beat the fourth-best checkers player in the USA. With this ground-breaking work, Samuel was surely nearly 30 years ahead of his time.

Only at the beginning of the nineties, as reinforcement learning emerged, did Gerald Tersauro build a learning backgammon program named TD-Gammon, which played at the world champion level (see Chap. 10).

6.6.1 Chess

Today many chess programs exist that play above grandmaster level. The breakthrough came in 1997, as IBM's Deep Blue defeated the chess world champion Gary Kasparov with a score of 3.5 games to 2.5. Deep Blue could on average compute 12 half-moves ahead with alpha-beta pruning and heuristic position evaluation.

Around the year 2005 one of the most powerful chess computers was Hydra, a parallel computer owned by a company in the United Arab Emirates. The software was developed by the scientists Christian Donninger (Austria) and Ulf Lorenz (Germany), as well as the German chess grand champion Christopher Lutz. Hydra uses 64 parallel Xeon processors with about 3 GHz computing power and 1 GByte memory each. For the position evaluation function each processor has an FPGA (field programmable gate array) co-processor. Thereby it becomes possible to evaluate 200 million positions per second even with an expensive evaluation function.

With this technology Hydra can on average compute about 18 moves ahead. In special, critical situations the search horizon can even be stretched out to 40 half-moves. Clearly this kind of horizon is beyond what even grand champions can do, for Hydra often makes moves which grand champions cannot comprehend, but which in the end lead to victory. In 2005 Hydra defeated seventh ranked grandmaster Michael Adams with 5.5–0.5 games.

Hydra uses little special textbook knowledge about chess, rather alpha-beta search with relatively general, well-known heuristics and a good hand-coded position evaluation. In particular, Hydra is not capable of learning. Improvements are carried out between games by the developers. As a consequence, Hydra was soon outperformed by machines that used smart learning algorithms rather than expensive hardware.

In 2009 the system Pocket Fritz 4, running on a PDA, won the Copa Mercosur chess tournament in Buenos Aires with nine wins and one draw against 10 excellent human chess players, three of them grandmasters. Even though not much information about the internal structure of the software is available, this chess machine represents a trend away from raw computing power toward more intelligence. This machine plays at grandmaster level, and is comparable to, if not better than Hydra. According to Pocket Fritz developer Stanislav Tsukrov [Wik13], Pocket Fritz with its chess search engine HIARCS 13 searches less than 20,000 positions per second, which is slower than Hydra by a factor of about 10,000. This leads to the conclusion that HIARCS 13 definitely uses better heuristics to decrease the effective branching factor than Hydra and can thus well be called more intelligent than Hydra. By the way, HIARCS is a short hand for *Higher Intelligence Auto Response Chess System*.

6.6.2 Go

Even though today no human stands a chance against the best chess computers, there are still many challenges for AI. For example Go. In this ancient Japanese game, played on a square board of 361 spaces with 181 white and 180 black stones, the effective branching factor is about 250. After 8 half-moves there are already $1.5 \cdot 10^{19}$ possible positions. Given this complexity, none of the classic, well-known game tree search algorithms have a chance against a good human Go player. Yet in the most recent previous edition of this book, it was stated that:

> The experts agree that "truly intelligent" algorithms are needed here. Combinatoric enumeration of all possibilities is the wrong approach. Rather, procedures are needed that recognize patterns on the board, track gradual developments, and make rapid "intuitive" decisions. Similar to object recognition in complex images, we humans are still far superior to today's computer programs. We process the image as a whole in a highly parallel manner, whereas the computer processes the millions of pixels successively and has great difficulty recognizing the essentials in the abundance of pixels. The program "The Many Faces of Go" recognizes 1100 different patterns and knows 200 different playing strategies. All Go programs, however, still have great difficulty recognizing whether a group of stones is dead or alive, or where in between to classify them.

This statement is now obsolete. In January of 2016, Google [SHM+16] and Facebook [TZ16] published the breakthrough concurrently. That same month, the program AlphaGo, developed and presented in [SHM+16] by Google DeepMind, defeated European Go champion Fan Hui 5:0. Two months later, Korean player Lee Sedol, one of the best in the world, was defeated 4:1. Deep Learning for pattern recognition (see Sect. 9.7), reinforcement learning (see Chap. 10) and Monte Carlo tree search (MCTS, see Sect. 6.5.1) lead to this successful result.

The program plays hundreds of thousands of games against itself and uses the results (win, loss, draw) to learn the best possible heuristic score for a given position. Monte Carlo tree search is used as a replacement for Minimax search, which is not suitable for Go. In Sect. 10.10, after we have gained familiarity with the necessary learning algorithms, we will introduce AlphaGo.

6.7 Exercises

Exercise 6.1

(a) Prove Theorem 6.1 on page 96, in other words, prove that for a tree with large constant branching factor b, almost all nodes are on the last level at depth d.

(b) Show that this is not always true when the effective branching factor is large and not constant.

Exercise 6.2

(a) Calculate the average branching factor for the 8-puzzle without a check for cycles. The average branching factor is the branching factor that a tree with an equal number of nodes on the last level, constant branching factor, and equal depth would have.

(b) Calculate the average branching factor for the 8-puzzle for uninformed search while avoiding cycles of length 2.

Exercise 6.3

(a) What is the difference between the average and the effective branching factor (Definition 6.2 on page 95)?

(b) Why is the effective branching factor better suited to analysis and comparison of the computation time of search algorithms than the average branching factor?

(c) Show that for a heavily branching tree with n nodes and depth d the effective branching factor $\bar{b}$ is approximately equal to the average branching factor and thus equal to $\sqrt[d]{n}$.

Exercise 6.4

(a) Calculate the size of the state space for the 8-puzzle, for the analogous 3-puzzle (2×2-matrix), as well as for the 15-puzzle (4×4-matrix).

(b) Prove that the state graph consisting of the states (nodes) and the actions (edges) for the 3-puzzle falls into two connected sub-graphs, between which there are no connections.

Exercise 6.5 With breadth-first search for the 8-puzzle, find a path (manually) from the starting node $\begin{array}{|c|c|c|}\hline 1 & & 3 \\ \hline 4 & 2 & 6 \\ \hline 7 & 5 & 8 \\ \hline\end{array}$ to the goal node $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & \\ \hline\end{array}$.

Exercise 6.6

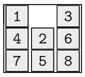
(a) Program breadth-first search, depth-first search, and iterative deepening in the language of your choice and test them on the 8-puzzle example.
(b) Why does it make little sense to use depth-first search on the 8-puzzle?

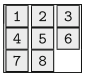
Exercise 6.7

(a) Show that breadth-first search given constant cost for all actions is guaranteed to find the shortest solution.
(b) Show that this is not the case for varying costs.

Exercise 6.8 The predecessors of all nodes must be stored to check for cycles during depth-first search.

(a) For depth first search develop a data structure (not a hash table) that is as efficient as possible for storing all nodes in the search path of a search tree.
(b) For constant branching factor b and depth d, give a formula for the storage space needed by depth-first search with and without storing predecessors.
(c) Show that for large b and d, we have $\sum_{k=0}^{d} k \cdot b^k \approx d \cdot b^d$.

Exercise 6.9 Using A* search for the 8-puzzle, search (manually) for a path from the starting node 

to the goal node 

(a) using the heuristic h_1 (Sect. 6.3.4).
(b) using the heuristic h_2 (Sect. 6.3.4).

Exercise 6.10 Construct the A* search tree for the city graph from Fig. 6.14 on page 106 and use the flying distance to Ulm as the heuristic. Start in Bern with Ulm as the destination. Take care that each city only appears once per path.

Exercise 6.11

(a) Show that the triangle inequality is valid for shortest distances on maps.
(b) Using an example, show that it is not always the case that the triangle inequality holds for direct neighbor nodes x and y, where the distance is $d(x, y)$. That is, it is not the case that $d(x,y) \leq d(x,z) + d(z,y)$.

Exercise 6.12 Program A* search in the programming language of your choice using the heuristics h_1 and h_2 and test these on the 8-puzzle example.

Exercise 6.13 Give a heuristic evaluation function for states with which HEURISTICSEARCH can be implemented as depth-first search, and one for a breadth-first search implementation.

Exercise 6.14 What is the relationship between the picture of the couple at the canyon from Fig. 6.13 on page 105 and admissible heuristics?

Exercise 6.15 Show that the heuristics h_1 and h_2 for the 8-puzzle from Sect. 6.3.4 are admissible.

Exercise 6.16

(a) The search tree for a two-player game is given in Fig. 6.23 with the ratings of all leaf nodes. Use minimax search with α-β pruning from left to right. Cross out all nodes that are not visited and give the optimal resulting rating for each inner node. Mark the chosen path.

(b) Test yourself using P. Winston's applet [Win].

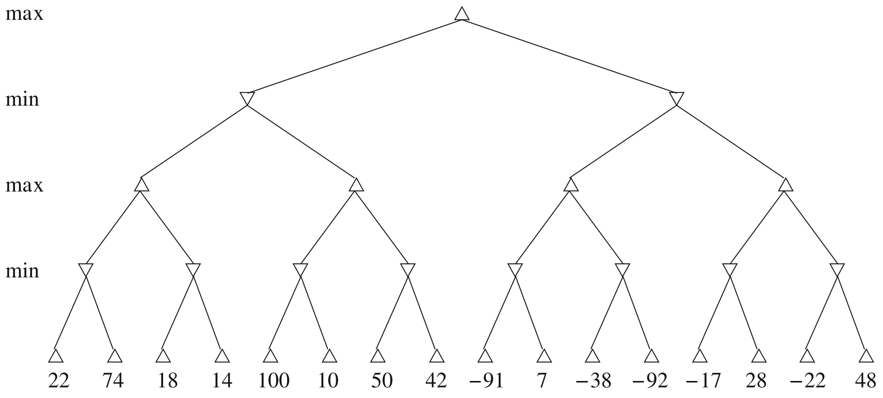


Fig. 6.23 Minimax search tree