Introduction to Artificial Intelligence

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

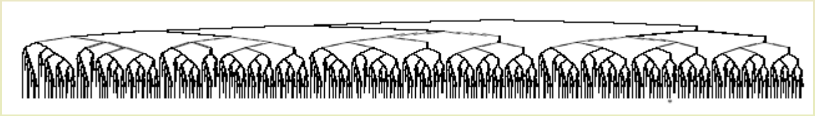
LSSU Math 400

Subsection 1

Introduction

Large Search Spaces

- The search in an extremely large search tree presents a problem for 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.
- For instance, the SLD resolution search tree for the proof of a specific very simple formula with three Horn clauses has shape:



  The tree was cut off at depth 14 and has a solution in the leaf node marked by $*$. It has a small branching factor of at most two.
- For realistic problems, the branching factor and depth of the first solution may become significantly bigger.

Enormity of the Search Space

- Assume the branching factor is 30 and the first solution is at depth 50.
- The search tree has $30^{50} = 7.2 \times 10^{73}$ leaf nodes.
- The number of inference steps is even bigger because, in addition, every inner node of the tree corresponds to an inference step:
$$\sum_{d=0}^{50} 30^d = \frac{30^{51} - 1}{30 - 1} = 7.4 \times 10^{73}.$$
- Evidently, nearly all of the nodes of this search tree are on the last level. This is generally the case.
- Assume we had 10,000 computers which can each perform 10^9 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
$$\frac{7.4 \times 10^{73} \text{ inferences}}{10000 \cdot 10^9 \text{ inferences/sec}} = 7.4 \times 10^{60} \text{ sec} \approx 2.3 \times 10^{53} \text{ years},$$
about 1043 times as much time as the age of our universe.

Facing the Enormity of a Task in Practice: Utility

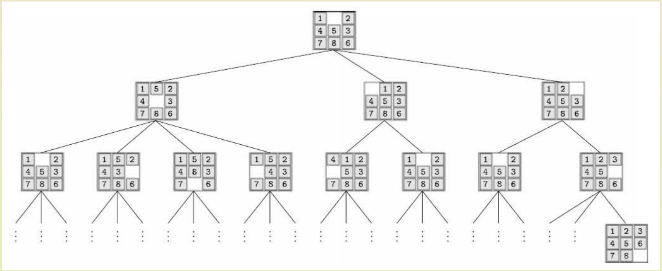
- There is no realistic chance of searching such spaces completely.
- The assumptions on the size of the search space are realistic: In chess, there are over 30 possible moves for a typical situation, and a game lasting 50 half-turns is relatively short.
- How can it be then, that there are good chess players and also good chess computers? How can it be that mathematicians find proofs for theorems in which the search space is even much bigger?
- Humans use intelligent strategies which dramatically reduce the search space.
- The experienced human will, by 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.

Facing the Enormity of a Task in Practice: Intuition

- 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 intuition plays a big role.
- This kind of search using ad-hoc criteria, is called **heuristic search**.
- Computers can improve their heuristic search strategies by learning, like humans.
- Search that does not use such methods, but, instead, relies on blindly trying out all possibilities, is termed **uninformed search**.

The 8-Puzzle

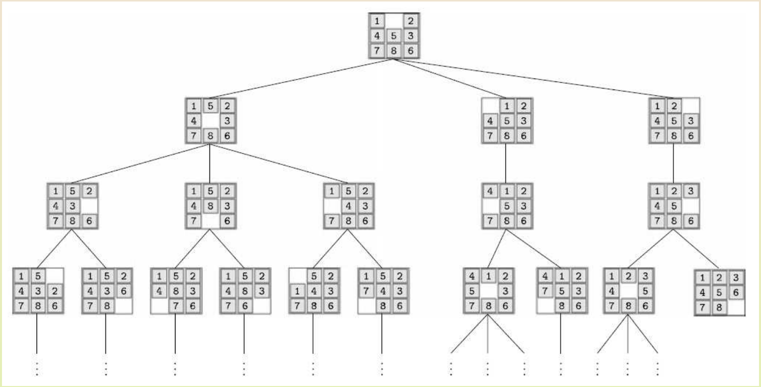
- The 8-puzzle: Squares with the numbers 1 to 8 are distributed in a 3×3 matrix. The goal is to reach a certain ordering of the squares. At each step a square can be moved left, right, up, or down into the empty space. (Or the empty space moves in the opposite direction.)
- The search tree for a specific starting state is



The branching factor alternates between two, three, and four. The **average branching factor**, i.e., constant branching factor of a tree with equal depth and equal number of leaves, is $\sqrt{8} \approx 2.83$.

The 8-Puzzle: Informing the Search

- Each state is repeated multiple times, since, in uninformed search, every action is reversed in the next step.
- If we disallow cycles of length 2, then we get:



The average branching factor is reduced to about 1.8.

Search Trees and Search Problems

- If from a state s, an action a_1 leads to a new state s', we write $s' = a_1(s)$. Another action a_2 may lead to state s'': $s'' = a_2(s)$.
- Recursive application of all possible actions to all states, beginning with the starting state, yields the **search tree**.

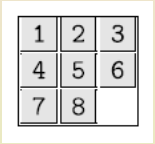
Definition (Search Problem)

A **search problem** is defined by the following values:

- **State**: Description of a state of the world in which the search occurs.
- **Starting state**: The initial state in which search starts.
- **Goal state**: A state where search terminates.
- **Actions**: All of the agent's allowed actions.
- **Solution**: The path from the starting state to the goal state.
- **Cost function**: Assigns a cost value to every action.
- **State space**: Set of all states.

8-Puzzle Revisited

- For the 8-puzzle, we get:
  - State: 3×3 matrix S with the values $1, 2, \ldots, 8$ (once each) and one empty square.
  - Starting state: A fixed, but arbitrary, state.
  - Goal state: A fixed, but arbitrary state, e.g.,



  - 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 either the set of all states, if solutions exist, or degenerate in domains where solutions do not exist.

Branching Factor

Definition (Branching Factor, Completeness)

- 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.
- A search algorithm is called **complete** if it finds a solution for every solvable problem. I.e., if a complete search algorithm terminates without finding a solution, then the problem is unsolvable.

- A tree with constant branching factor b and depth d has total number of nodes $n = \sum_{i=0}^{d} b^i = \frac{b^{d+1}-1}{b-1}$.
- So, for a given depth d and node count n, the effective branching factor b is the solution of $n = \frac{b^{d+1}-1}{b-1}$.

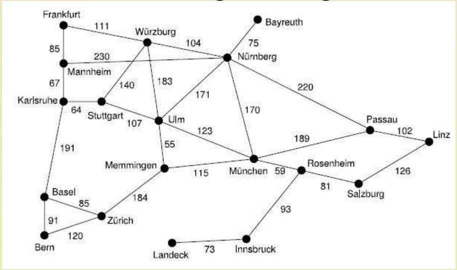
Large Branching Factors

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

Theorem

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

- Example: Given a map with cities as nodes and highway connections between the cities as distance-weighted edges,



we are looking for an optimal route from a city A to a city B.

From the Map Problem to a Search Problem: Optimality

- State: A city as the current location of the traveler.
- Starting state: An arbitrary city A.
- Goal state: An arbitrary city B.
- 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.
- State space: All cities, i.e., all nodes of the graph.
- To find the route with minimal length, the costs must be taken into account:

Definition (Optimal Search Algorithms)

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

Determinism and Observability

- The 8-puzzle problem is **deterministic**, which means that every action leads from a state to a unique successor state.
- It is also **observable**: the agent always knows which state it is in.
- In route planning in real applications these are not always given:
  - The action "Drive from Munich to Ulm" may, e.g., because of an accident, lead to the successor state "Munich".
  - It may also occur that location awareness is lost.
- For simplicity, we look at deterministic and observable problems only.
- For these problems, action planning is simpler because it is possible to find action sequences for the solution by modeling:
  - 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. Online algorithms are needed, which make decisions based on sensor signals in every situation. Reinforcement learning works toward optimization of decisions.

Subsection 2

Uninformed Search

Breadth-First Search

- In **breadth-first search**, the search tree is explored from top to bottom according to the following algorithm:

BREADTHFIRSTSEARCH(NodeList, Goal)

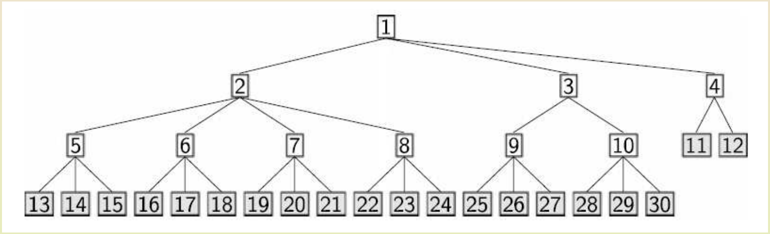
```
NewNodes = $\emptyset$
**For all** Node $\in$ NodeList
    **If** GoalReached(Node, Goal)
        **Return**("Solution Found", Node)
    NewNodes = **Append**(NewNodes, Succesors(Node))
**If** NewNodes $\neq \emptyset$
    **Return**(BREADTHFIRSTSEARCH(NewNodes, Goal))
**Else**
    **Return**("No Solution")
```

Description of Breadth-First Search

- First every node in the node list is tested for whether it is a goal node;
  - 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 process repeats until no more successors are generated.
- This algorithm works for arbitrary applications if the application specific functions “GoalReached” and “Successors” are provided.
  - “GoalReached” calculates whether the argument is a goal node;
  - “Successors” calculates the list of all successor nodes of its argument.

An Example of Breadth-First Search

- In the following picture, the third-level nodes are numbered according to the order in which they were generated:



- The successors of nodes 11 and 12 have not yet been generated.

Analysis of Breadth-First Search: Completeness and Time

- Breadth-first search completely searches through every depth and reaches every depth in finite time. Therefore, if there is a solution, it will find it provided the branching factor b is finite, i.e., it is complete for finite branching factor b.
- The optimal (shortest) solution is found if all costs are the same.
- 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 computation time is given by

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

Analysis of Breadth-First Search: Optimality and Space

- Although only the last level is saved in memory, the memory space requirement is also $\mathrm{O}\left(b^d\right)$. So memory will quickly fill up and the search will end.
- The problem of the shortest solution not always being found can be solved by the so-called **Uniform Cost Search**:
  - The node with the lowest cost from the ascendingly sorted list of nodes is always expanded, and the new nodes sorted in.
- This deals with optimality, but for the memory problem depth-first search must be employed.

Description of Depth-First Search

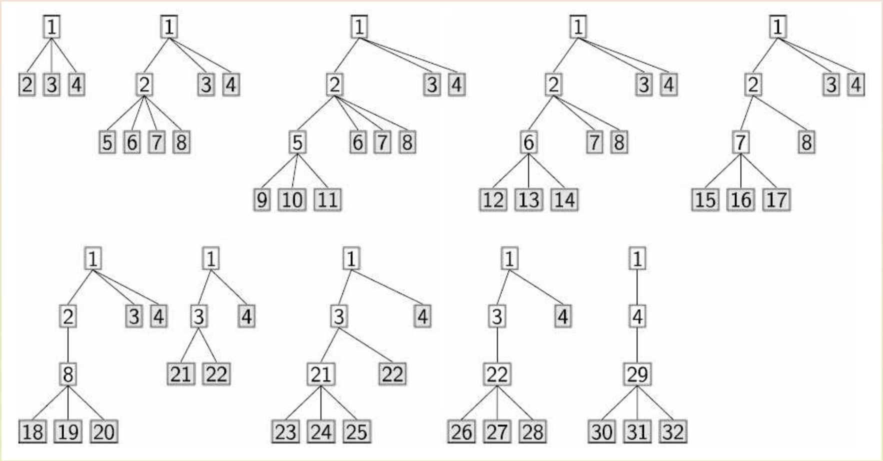
- In depth-first search only a few nodes are stored in memory at one time.
- After the expansion of a node only its successors are saved, and the first successor node is immediately expanded.
- Thus the search quickly becomes very deep.
- Only when a node has no successors and the search fails at that depth is the next open node expanded via backtracking to the previous branch, and so on.

Depth-First Search

- The recursive algorithm follows:
- "First" returns the first element and "Rest" the rest of the list.

```
DEPTHFIRSTSEARCH(Node, Goal)
    If GoalReached(Node, Goal) Return("Solution found")
    NewNodes = Successors(Node)
    While NewNodes $\neq \emptyset$
        Result = DEPTHFIRSTSEARCH(First(NewNodes), Goal)
        If Result = "Solution found" Return("Solution found")
        NewNodes = Rest(NewNodes)
    Return("No solution")
```

Example of a Depth-First Search Tree



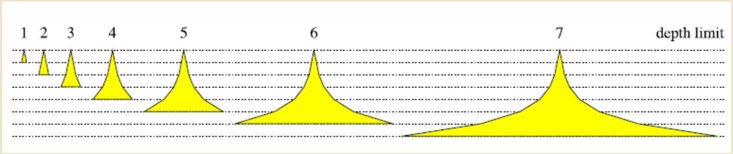
All nodes at depth three are unsuccessful and cause backtracking.
Nodes are numbered in the order they were generated.

Analysis of Depth-First Search

- Depth-first search requires much less memory than breadth-first search because at most b nodes are saved at each depth. Thus we need $b \cdot d$ memory cells.
- 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.
- Accordingly, the question of finding the optimal solution is not applicable.
- 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. If no solution is found in the pruned search tree, there can nonetheless be solutions outside the limit. Thus the search becomes incomplete.
- There are, however, modifications for ensuring completeness.

Description of 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:



- This iterative raising of the depth limit is called **iterative deepening**.
- We must augment the depth-first search program 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

$$\textbf{While}\ \text{NewNodes} \neq \emptyset\ \textbf{And}\ \text{Depth} < \text{Limit}$$

Iterative Deepening

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")
```

Analysis of Iterative Deepening: Last Tree

- The memory requirement is the same as in depth-first search.
- Repeatedly re-starting depth-first search at depth zero does not entail a lot of redundant work for large branching factors.
- 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 number of nodes of the last tree searched is

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

Analysis of Iterative Deepening: All Trees

- The number of nodes of all trees searched beforehand together is

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

  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 $\frac{1}{b-1} = \frac{1}{19}$ of the number of nodes in the last tree. The computation time for all iterations besides the last can be ignored.
- This method is complete, and given a constant cost for all actions, it finds the shortest solution.

Comparison of Uninformed Search Algorithms

	Breadth-First	Uniform Cost	Depth-First	Iterative Deep.
Complete	Yes	Yes	No	Yes
Optimal	Yes(∗)	Yes	No	Yes(∗)
Time	b^d	b^d	∞ or b^{d_s}	b^d
Space	b^d	b^d	bd	bd

(∗) means that the statement is only true given a constant action cost. d_s is the maximal depth for a finite search tree

- Iterative deepening is the winner of the comparison because it gets the best grade in all categories.
- For realistic applications it is usually not successful because of huge search space.
- What is needed is an intelligent search that only explores a tiny fraction of the search space and finds a solution there.

Subsection 3

Heuristic Search

Heiristics

- **Heuristics** are problem-solving strategies which in many cases find a solution faster than uninformed search.
- This is not guaranteed: they may require a lot more time and could even result in the solution not being found.
- **Heuristic decisions** arise from the need for quick real-time decisions with limited resources: A good solution found quickly is preferred over an expensive optimal solution.
- 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.
- There is a subtle difference between the effort to find a solution and the total cost of this solution.
  - Example: Google Maps may spend half a second's worth of effort to find a route from SSM to Detroit, but the trip may cost five hours and a bit of money.

Introduction to Heuristic Search

- Next we modify the breadth-first search algorithm by adding the evaluation function.
- The currently open nodes are no longer expanded left to right, 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 must differentiate 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 of a node, must also immediately calculate for these successor nodes their heuristic ratings as a component of each node.

Heuristic Search

HeuristicSearch(Start, Goal)

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

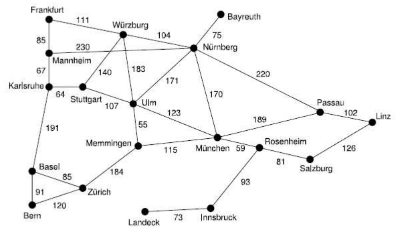
- The node list is initialized with the starting nodes.
- In the loop, the first node from the list is removed and tested for whether it is a solution node.
  - If not, it will be expanded with the function "Successors" and its successors added to the list with the function "SortIn".
  - "SortIn(X, Y)"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

Heuristic Ratings

- Depth-first and breadth-first search are special cases of the function HEURISTICSEARCH.
  - This happens, since they can be easily generated by plugging in the appropriate evaluation function.
- 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.
    - This **cost estimate function** is denoted by h.

Greedy Search

- It seems sensible to choose the state with the lowest estimated cost h from the list of currently available states.
- The cost estimate then can be used as the evaluation function, i.e., in the function HEURISTICSEARCH we set $f(s) = h(s)$.
- Example: In planning a trip, we set finding the straight line path from city to city 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.

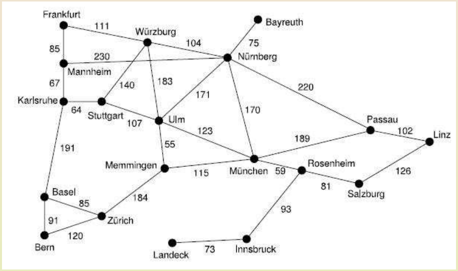


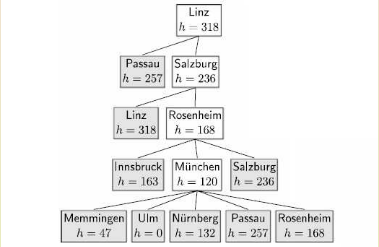
Basel	204
Bayreuth	207
Bern	247
Frankfurt	215
Innsbruck	163
Karlsruhe	137
Landeck	143
Linz	318
München	120
Mannheim	164
Memmingen	47
Nürnberg	132
Passau	257
Rosenheim	168
Stuttgart	75
Salzburg	236
Würzburg	153
Zürich	157

If Ulm is chosen as the destination, the cost estimate function becomes $h(s)$ = flying distance from s to Ulm.

Example of a Fast Execution for Trip Planning

- The search tree for starting in Linz is represented below:



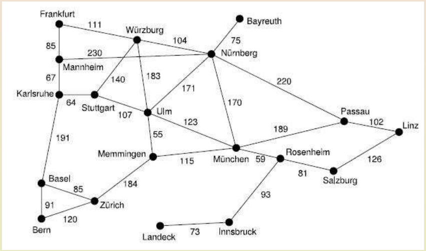


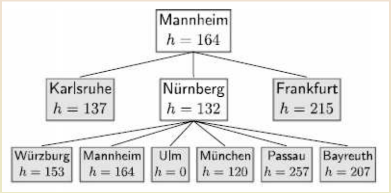
Node	München	Innsbruck	Salzburg	Passau	Linz
heuristic rating	120	163	236	257	318

- The tree is very slender and the search, thus, finishes quickly.
- This search does not always find the optimal solution.

Example of a Failure to Find Optimal Solution

- Example: This algorithm fails to find the optimal solution when starting in Mannheim.





- The Mannheim-Nürnberg-Ulm path has a length of 401 km.
- The route Mannheim-Karlsruhe-Stuttgart-Ulm is shorter at 238 km.
- Nürnberg is closer than Karlsruhe to Ulm, but the distance from Mannheim to Nürnberg is significantly greater than that from Mannheim to Karlsruhe.
- But the heuristic only looks ahead “greedily” to the goal.

Admissible Heuristics and A*-Algorithms

- We want to take into account the costs $g(s)$ that have accrued during the search up to the current node s.
- We define the cost function $g(s) =$ Sum of accrued costs from the start to the current node.
- Then add the estimated cost $h(s)$ to the goal: $f(s) = g(s) + h(s)$.

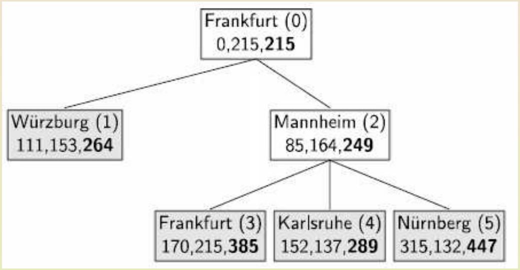
Definition of Admissible Heuristics

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 $f(s) = g(s) + h(s)$, where h is an admissible heuristic function, is called A*-**algorithm**.
- This famous algorithm is complete and optimal.

An Example of A*

- We are looking for the shortest path from Frankfurt to Ulm.



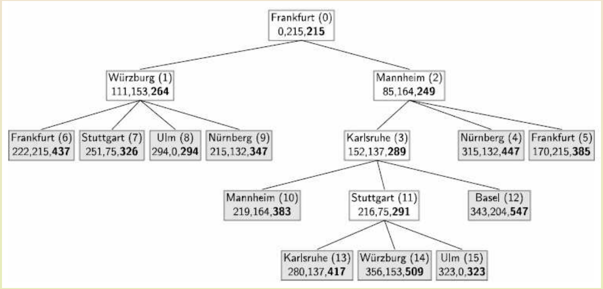
In the boxes, under the name of the city s, $g(s)$, $h(s)$, and $f(s)$ are shown.

Numbers in parentheses after the city names show the order in which the nodes have been generated by the "Successor" function.

- The successors of Mannheim are generated before the successors of Würzburg.

An Example of A* (Cont'd)

- The optimal solution Frankfurt-Würzburg-Ulm is generated shortly thereafter in the eighth step, but it is not yet recognized as such.
- 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.



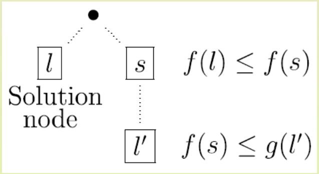
- Only when all f values are greater than or equal to that of the solution node Ulm (8) have we ensured that we have an optimal solution.
- Otherwise there could potentially be another solution with lower costs.

Optimality of A*-Algorithm

Theorem (Optimality of A*)

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

- In the HEURISTICSEARCH algorithm, every newly generated node s is sorted in by "SortIn" according to its heuristic rating $f(s)$. The node with the 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 $f(\ell) < f(s)$. No better solution ℓ' can be found, even after expansion of all other nodes:
  $g(\ell) \overset{h(\ell)=0}{=} g(\ell) + h(\ell) \overset{\text{defn}}{=} f(\ell) \overset{\text{sorted}}{<} f(s) \overset{\text{defn}}{=} g(s) + h(s) \overset{\text{admss}}{\leq} g(\ell')$.



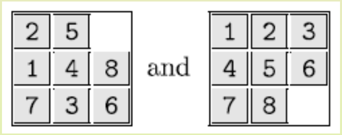
- Thus, the discovered solution ℓ is optimal.

IDA*-Search

- The A*-search inherits from breadth-first search the drawback that 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 complexity.
  - Based on the heapsort algorithm, we can structure the node list as a heap with logarithmic time complexity for insertion and removal.
- 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, we use a limit for the heuristic evaluation $f(s)$.
  - This process is called the IDA*-**algorithm**.

Two Heuristics for the 8-Puzzle

- In A*, or IDA*, we have a search algorithm with many good properties.
  - It is complete and optimal.
  - It works with heuristics, and therefore can significantly reduce the computation time needed to find a solution.
- 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. 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.



The Manhattan distance of the two states is calculated as $h_2(s) = 1+1+1+1+2+0+3+1 = 10$.
The admissibility of the Manhattan distance is also obvious.

Summary: Heuristics and Solvable Problems

- 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.
- For difficult combinatorial search problems, even iterative deepening usually fails due to the size of the search space.
- Heuristic search reduces the effective branching factor.
- The IDA*-algorithm is complete and uses little memory.
- Heuristics give a significant advantage if the heuristic is "good".
- The task is to find heuristics that reduce the effective branching factor.
- Machine learning techniques may automatically generate heuristics.

Summary: Heuristics and Unsolvable Problems

- 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.
- In this case the heuristic is always a disadvantage because of the extra computational cost of evaluating the heuristic.
- 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.

Subsection 4

Games With Opponents

Deterministic and Observable Zero-Sum Games

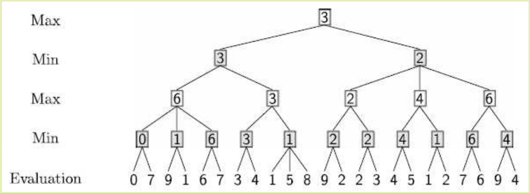
- Games for two players, such as chess, checkers, Othello, and Go are deterministic because every action (a move) results in the same child state given the same parent state.
- In contrast, backgammon is non-deterministic because its child state depends on the result of a dice roll.
- These games are all observable because every player always knows the complete game state.
- Many card games, such as poker, 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.
- We continue with deterministic and observable games that, in addition are **zero-sum** games, i.e., games in which every gain one player makes means a loss of the same value for the opponent.
- This is true of the games chess, checkers, Othello, and Go.

Case For Heuristics

- 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.
- The effective branching factor may be so high that there is no chance to fully explore the search tree.
- In addition, because of time constraints, the search may have to be limited to an appropriate depth in the tree.
- Since among the leaf nodes of this depth-limited tree there may be no solution nodes, a heuristic evaluation function B for positions has to be used.

Minimax Search

- **Max**: a player whose game we wish to optimize;
- **Min**: Max's opponent. Min's moves are not known in advance, and thus neither is the actual search tree.
- We assume 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:



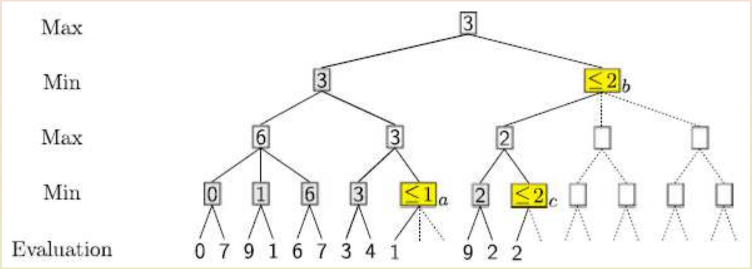
The evaluation of an inner node is derived recursively as the maximum or minimum of its child nodes, depending on the node's level.

Alpha-Beta Pruning: Ideas

- **Alpha-beta pruning**: depth-first search up to a preset depth limit.
- The search tree is searched through from left to right.
- In the minimum nodes the minimum is generated from the minimum value of the successor nodes and, similarly, in the maximum nodes.
- Perform an analysis of possibilities to prune subtrees when further processing can be avoided.

Alpha-Beta Pruning: Example I

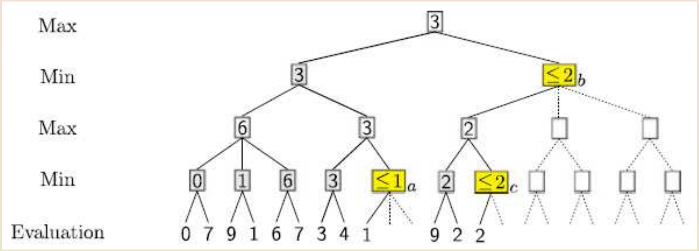
- Look at node a:



- 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.

Alpha-Beta Pruning: Example II

- Look at node b:



- Since the first child of b has the value 2, the minimum to be generated for b can only be ≤ 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.

Alpha-Beta Pruning: The General Case

- This can be generalized:
  - 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 $\beta \leq \alpha$, where α is the largest value of a maximum node in the path from the root to *k*, then the search under *k* can end.
  - If at a maximum node ℓ the current value $\alpha \geq \beta$, where β is the smallest value of a minimum node in the path from the root to ℓ, then the search under ℓ can end.
- We present the algorithm next; It is an extension of depth-first search with two functions which are called in alternation.

Alpha-Beta Pruning: The Algorithm

$\textsc{AlphaBetaMax}(\text{Node}, \alpha, \beta)$

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

$\textsc{AlphaBetaMin}(\text{Node}, \alpha, \beta)$

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

Alpha-Beta Pruning: 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 $\sqrt{b}$. Then only $n_d = \sqrt{b}^d = b^{d/2}$ leaf nodes would be created.
- Thus, the depth limit and the search horizon are doubled.
- 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 $n_d = b^{\frac{3}{4}d}$.

Further Improvements and Non-Determinism

- 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.
  - If it had been, the search would be unnecessary.
  - With a simple trick we can get a relatively good node ordering.
  - We combine 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.
- 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 three types of levels in the sequence

    $$\text{Max, dice, Min, dice}, \ldots$$

    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 with the average values.

Subsection 5

Heuristic Evaluation Functions

Evaluation Functions: Relying on Human Expertise

- How do we find a good heuristic evaluation function for the task of searching?
- The classical way uses the knowledge of human experts.
- The knowledge engineer formalizes the expert's implicit knowledge in the form of a computer program.
  - In the first step, experts are questioned about the most important factors in the selection of an action.
  - Then an attempt is made 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 states.
  - In the next step the weights 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.

Evaluation Functions: Chess

- A typical linear evaluation function for chess:

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

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

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

- Nearly all chess programs make a similar evaluation for material.
- However, there are big differences for all other features.

Evaluation Functions: Using Machine Learning

- Suppose we want to automatically optimize the weights of the evaluation function $B(s)$ for chess.
- 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 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.
- This automation of the learning cycle is difficult in practice.

Evaluation Functions: Credit Assignment Problem

- A central problem with improving the position rating based on won or lost matches is known today as the **credit assignment problem**.
- Rating occurs at the end of the game, but individual moves are not rated, so actions do not receive any feedback but until the very end.
- How should feedback be given for past actions?
- And how should actions be improved based on feedback?
- The field of reinforcement learning concerns itself with such questions.
- Most of the best chess computers in the world today still work without machine learning techniques for two reasons:
  - Reinforcement learning algorithms developed up to now require a great deal of computation time given large state spaces.
  - The manually created heuristics of high performance chess computers are already heavily optimized. So only very good learning systems can lead to improvements.