Algorithmen für Einführung in die künstliche Intelligenz - WiSe 2023/2024

Nils Dambowy

January 26, 2024

Introduction

This document goes over the different algorithms used in the lecture "Einführung in die künstliche Intelligenz"(Introduction to Artificial Intelligence) taught by Dr. Kristian Kersting in the wintersemester 2023/2024 at Technische Universität Darmstadt.

Tree Search Strategies

In order to understand Tree Search strategies we first have to introduce a few definitions.

Definition 1 (State space)**.** A **state space** is a set of all possible states. States describe a possible situation in our environment. It is implicitly defined by the initial state and the successor function.

Definition 2 (Transition/Action)**.** A **transition** describes possible a action to take betweeb one state or another. In this lecture we only count direct transitions between two states(single actions).

Definition 3 (Costs)**.** Often transitions aren't alike and differ. We express this by adding a '**cost**'to each action. Often the goal in search algorithms is to minimize the cost to reach the goal.

Definition 4 (Path)**.** A **path** is a sequence of states connected by a sequence of actions.

Definition 5 ((Optimal) solution)**.** A **solution** is a path that lead from the initial state to a goal state. An optimal solution is a solution with minimal path cost.

We can now think of the states as nodes and of the actions as edges connecting different states. The main idea of tree search algorithms is to treat this state-space graph as a tree. However, this raises the question in what we should expand the nodes or what actions should we perform next? In this lecture two categories of *search strategies* are presented - uninformed and informed.

Definition 6 (Search strategy)**.** A **search strategy** is defined by picking the order of node expansion.

Different search strategies are described/evaluated along the following dimensions:

1. **Completeness**: Does it always find a solution if one exists?
2. **Time Complexity**: Number of node expansions.
3. **Space Complexity**: Maximum number of nodes in memory.
4. **Optimality**: Does it always find the optimal(least cost) solution?

Uninformed Tree Search strategies

A uninformed search strategy has no other information other than the problem defintition.

Uniform-Cost Search

Definition 7 (Uniform-Cost Search)**.** Each node is associated with a fixed cost (nodes can have different costs) and they accumulate over the path within the search. Uniform-Cost Search uses the lowesrt cumulative cost to find a path. **Breadth-First search** is a special case of uniform-cost search where all costs are equal. It starts at the tree root and explores the tree level by level.

1. **Completeness**: Yes, if each step has a positive cost, otherwise infinite loops are possible. Hence BFS, is also complete.
2. **Time Complexity**: $O(b^d)$ for BFS and $O(b^{1+\lfloor \frac{OptCost}{Eps} \rfloor})$ for UCS, where OptCost = cost of optimal solution, every actions costs at least eps.
3. **Space Complexity**: /
4. **Optimality**: Yes, since nodes expand in increasing order of path costs. In turn BFS is also optimal.

Examples 8. The given search problem (see figure **??**) involves states A, B, C, D, and Z. Costs for transitions between nodes are specified on the edges. Your task is to find a path from the initial state A to the goal state Z using various search algorithms. The successor function generates the successors of a node in alphabetical order. If two nodes in the open list (fringe) of the search have the same sorting, then the alphabet determines their order (A first, Z last).

For Uniform-Cost search indicate the path the search would find and at each step provide the fringe, the list of nodes yet to be visited, in

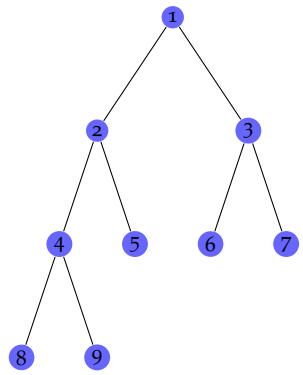


Figure 1: Order of node expansion using BFS.

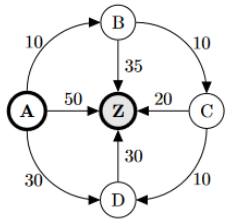


Figure 2: Graph taken from the wintersemester 2020/2021 demo exam.

the relevant sorting. Note that nodes may appear multiple times in the fringe if multiple paths lead to them.

Solution: In our case the path A ⇒ B ⇒ C ⇒ Z will be our solution. Concerning the fringe we have:

1. Step 1: (B,10), (D, 30) and (Z, 50).
2. Step 2: (C,30), (D, 30), (Z, 45) and (Z,50).
3. Step 3: (D,30), (D,30), (Z,40), (Z,45) and (Z, 50).
4. Step 4: (D,30), (Z,40), (Z,45), (Z, 50) and (Z, 60).
5. Step 5: (Z,40), (Z,45), (Z, 50) and (Z, 60).
6. Step 6: (Z,45), (Z, 50) and (Z, 60).

Depth-First Search

Definition 9 (Depth-First Search)**.** Depth-First search starts at the tree root and explores the tree as far as possible along one branch before going back step-wise and exploring alternative branches.

1. **Completeness**: No, fails in infinite-depth search spaces and spaces with loops. Can be modified to be complete by avoiding repeated states and limit depth.
2. **Time Complexity**: Explores each branch until max depth m, i.e. $O(b^m)$. Terrible if m > d (depth of goal node), but may be good in dense settings
3. **Space Complexity**: Only a branch and their unexpanded siblings have to be stored. Therefore linear complexity, i.e. $O(b \cdot m)$
4. **Optimality**: No, londer solutions may be found before shorter solutions. Solution could be more expensive then the optimal one.

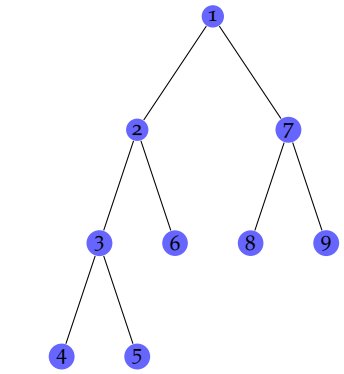


Figure 3: Order of node expansion using DFS.

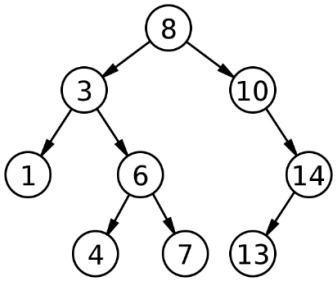


Figure 4: Graph taken from the wintersemester 2022/2023 demo exam.

Examples 10. Given the graph (see figure **??**). In what order would the nodes be exapanded give we started DFS in the tree root.

Solution: In our case the path 8 ⇒ 3 ⇒ 1 ⇒ 6 ⇒ 4 ⇒ 7 ⇒ 10 ⇒ 14 ⇒ 13 will be our solution.

Depth-limited Search

Definition 11 (Depth-limited Search)**.** The depth within the search is limited to l. Nodes with depth d > l are not considered.

1. **Completeness**: No.
2. **Time Complexity**: $O(b^l)$
3. **Space Complexity**:$O(b \times m)$
4. **Optimality**: No, see DFS.

Iterative Deepening Search

Definition 12 (Iterative Deepening Search)**.** Similiar to Depth-limited Search, but increase l after each failed search, i.e. l = 1, 2, 3, ...

1. **Completeness**: Yes.
2. **Time Complexity**: first levels have to be search d times $\Rightarrow d \cdot b + (d-1)b^2 + \ldots + 1 \cdot b^d = \sum_{i=1}^{d} (d-i) \cdot b^i$
3. **Space Complexity**:Linear complexity $O(b \cdot m)$
4. **Optimality**: Yes, the shortest path is found.

Examples 13. We now want to find a path from Node 8 to Node 7 (see figure **??**) using Iterative Deepening Depth-First Search (IDDFS). Specify the path the search would find, and at each step of the search, indicate which nodes were expanded in what order. **Solution:**

1. Step 1: 8
2. Step 2: 8, 3, 10
3. Step 3: 8, 3, 1, 6, 10, 14
4. Step 4: 8, 3, 1, 6, 4, 7

Based on the implementation Iterative Deepening Search either breaks here or finishes the step.

Bidirectional Search

Definition 14 (Bidirectional Search)**.** Performs two search strategies simultaneously, starting with the root and goal state. Stop if node occurs in both searches. Bidirectional search reduces the complexity $b^{\frac{d}{2}} + b^{\frac{d}{2}} << b^d$. However it is only applicable if actions can be reversed and if we choose to do DFS bidirectional search, the search pathes may not meet.

1. **Completeness**: Yes.
2. **Time Complexity**: first levels have to be search d times $\Rightarrow d \cdot b + (d-1)b^2 + \ldots + 1 \cdot b^d = \sum_{i=1}^{d} (d-i) \cdot b^i$
3. **Space Complexity**:Linear complexity $O(b \cdot m)$
4. **Optimality**: Yes, the shortest path is found.

Informed Tree Search Strategies

The problem with uninformed search that the kind of search strategies are ineffiecient, so how do we improve perfomance? What about giving the algorithm 'hints' about desirability of different states? We could, for example, give the straight-line distance for approximation of the remaining travel distance. Those 'hints' are called heuristics.

Definition 15 (Heuristics)**.** A **heuristic** h informally denotes a rule of thumb, i.e. a rule that may be helpful in solving the problem. In tree search, a heuristic denotes a function h that estimates the remaining costs to reach the goal. **Note:** Heuristics can also go wrong!

Greedy Best-first search

Greedy Best-first search uses a heuristics h to evaluate every node in the fringe by assigning them a cost. Afterwards the node with the lowest cost is expanded.

1. **Completeness**: No, we can get stuck in loops. It is complete in a finite state space when we make sure to avoid repeating states.
2. **Time Complexity**: Worst Case $O(b^m)$, same as DFS, but can be improved by using good heuristics.
3. **Space Complexity**: has to keep all nodes in memory, worst case $O(b^m)$.
4. **Optimality**: No, since the solution depends on the heuristics.

A Search*

A* search is built on Greedy Best-first search. It tries to minimize not only the estimated cost h(n) but also the true costs so far g(n). We try to avoid expanding paths that are already expensive and evaluate the complete path cost and not only the remaing costs.

$$f(n) = g(n) + h(n)$$

1. **Completeness**: Yes. Exception: If there infinitely many nodes with f(n) $\leq$ f(G).
2. **Time Complexity**: It can be shown that the number of nodes grows exponentionally unless the error of the heuristics h(n) is bounded by the logarithm of the value of the actual path cost h*(n) i.e.

$$| \text{ h(n) - h*(n)} | \leq O(logh * (n))$$

3. **Space Complexity**: has to keep all nodes in memory.
4. **Optimality**: Depends on the heuristic.

Local search algorithms

Applying the (un)informed search strategies to real world problems quickly shows us their limitations. When dealing with much larger search spaces ($<10^{100}$ states) those types of algorithms have shown to prodoce unsatisfyable results. In addition, if we face problem where path is not the goal (e.g. optimization problems) those types of search strategies aren't of any help.

Definition 16 (Optimization problems)**.** An optimization problem is one where all states/nodes can be a solution (to different degrees) but the target is to find a state that optimizes (min or max) the solution according to an *objective function*. There is no explicit goal state and also no path cost.

Definition 17 (Objective/Evaluation function)**.** An objective function tells us how good a state is, also in comparison to other states. Its value is either minimized or maximized depending on the optimization problem.

Local search algorithms (or Iterative Improvement Methods)

Local search algorithms are a class of algorithms that traverse only a single state rather than saving multiple paths in memory. It modifies its state iterative, trying to improve a criteria. In many optimization problems the sequence of actions and costs are irrelevant.
Advantages:

1. Uses a very little/constant amount of memory
2. Find a reasonably solution in very large state

Disadvantages:

1. No guarantees for completeness or optimality

Hill climbing search / Greedy local search

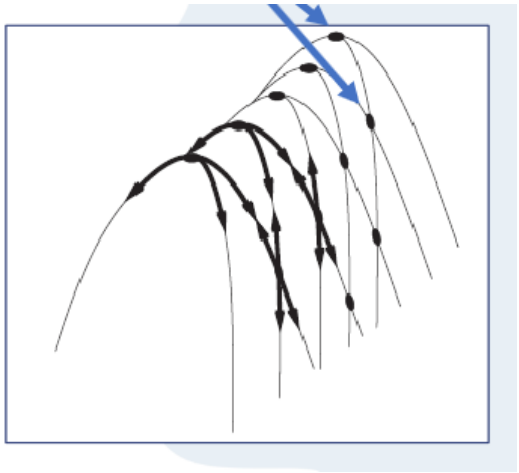
Hill climbing search expands every neighbor and then moves to the one with the highest evaluation. It does this until we reach a maximum(evaluation goes down.)
Problem - Local optima:

1. The algorithm will stop as soon as it reaches a maxima
2. But this maxima does not to have to be global(plateau, ridge or shoulders)

Solution - Local optima:

1. Random Restart Hill Climbing: Different initial positions result in differnt local optima. Make several iterations with different starting positions.
2. Stochastic Hill Climbing: Select the successor node randomly. Better nodes have a higher probability of being selected.

Problem - Ridge Problem:



1. Every neighbor state appears to be downhill.
2. The search space has an uphill, the neighbors not.

Gradient Descent

When we use Gradien Descent the states in the search space are represented a locatons with a heuristic value as elevation. The goalis now to find the global optimum(local search).

Definition 18 (Gradient)**.** A gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the chane in all weights with regard to the change in error.

Gradient descent can be described as Hill-climbing in a continuos state space. However, instead of climbing up a hill we hike down to the bottom of the valley in order to minimize the cost-function $J(\Theta)$. It works well in smooth spaces, poorly in rough.

Definition 19 (Learning Rate)**.** The learning rate is a hyperparameter, controlling how quickly the model is adapted to the problem.

Smaller learning rate:

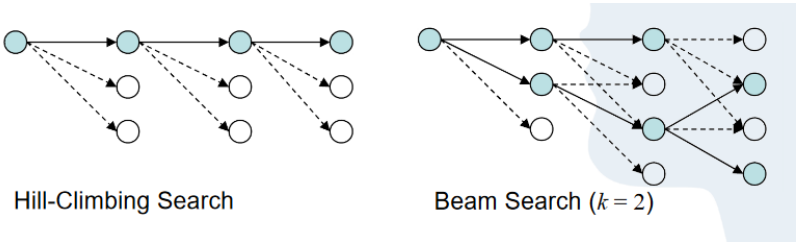
1. Smaller changes, requires more training epochs

Larger learning rate:

1. More rapid changes, needing fewer epochs
2. Can converge to a local optima or not at all

Beam search

The main idea of beam search is to expand the nodes we keep track when using local search (like Hill-climbing). We instead keep track of k states rather than just one (k is called the beam size).



At each iteration, all the successors of all k states are generated and we then select the k best successors from the complete list and repeat.

Simulated Annealing

Here we use conventional hill-climbing style techniques, but occasionally take a step in a direction other than that in which there is improvement(like downhill moves, away from the solution). As time passes, the probability that a downhill step is taken is decreased. We model this by using a 'temperature'.

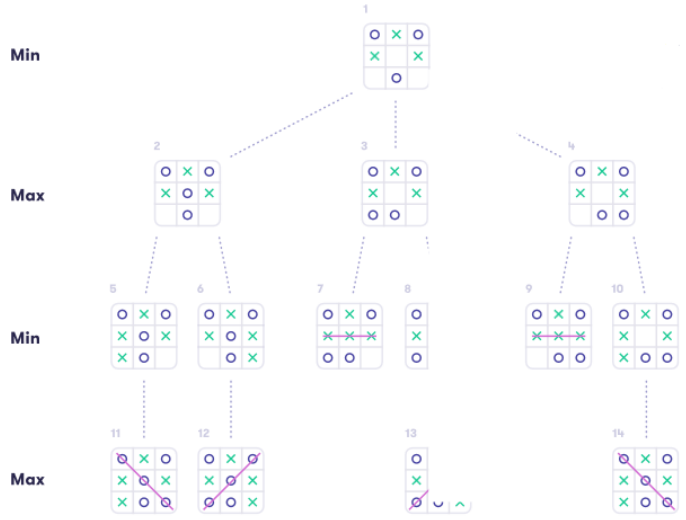
Definition 20 (Temperature)**.** The temperature is a hyperparameter, controlling how frequently we allow "bad moves" to escape local optima. Usually the temperature decays exponentionally over the process. If lowered slowly enough, we converge to a global optimum. This however can unfortunately take a very long time.

Adversial Search

Adversial search is search, where one examines the problem that arises when we try to plan ahead of the world and other players are planning against us or have conflicting goals to ours while sharing the same search space. It is used to model games as search problems, here games are presented as game trees. Each player has to consider the actions of the other players and the effect of their actions on their perfomance.
Formalization of the problem:

1. Initial state: Specifies how the game is set up at the start.
2. Player(s): Specifies which players turn it is.
3. Action(s): Returns a set of legal moves in the state *s*.
4. Result(s, a): Transition model, specifies the resulting state s' doing *a* in state *s*.
5. Terminal(s): Tests if the state *s* fulfills the goal/terminal constraints.
6. Utility(s, p): The utility function returns a numeric value for a terminal state *s* from the perspective player *p*.

To solve games, we build so called game trees. Differently to search trees, game trees are arranged in level that correspond to players. The root node is always the active/current player, the lead node are called terminal. Each terminal node has a utility values which corresponds to the outcome of the game.



Minimax Algorithm

We build a game tree - where the nodes represent the states of the game and edges the moves made by the players in the game. The players are...

1. **MIN**: Decrease the chances of **MAX** to win the games. (Opponent)
2. **MAX**: Increases his chances of winning the game.

Both play the game alternating turns and following the above strategy.

- Completeness: Yes, if the tree is finite.
- Time Complexity: $O(b^m)$
- Space Complexity: $O(b \cdot m)$
- Optimality: Yes, assuming an optimal opponent.

However, the main problem of game trees is that they can be massive and simply way to big to traverse in full.
Solution: Instead of traversing the full tree, we limit the depth. We do that by *pruning* the tree. We cut back the tree by ignoring unwanted positions of a search tree which make no difference to its final result and thereby only slow down the algorithm.

Alpha-Beta Pruning

A modified, optimized version of the Minimax algorithm. It uses pruning to reduce the amount of exploration without losing the correctness of Minimax.
Alpha-Beta Pruning is based on two parameters:

1. Alpha: The best(highest-value) choice we have found so far at any point along the path of Maximizer to the root. The inital value of alpha is $-\infty$.
2. Beta: The best(lowest-value) choice we have found so far at any point along the path of Minimizer to the root. The initial value of beta is ∞.

The key difference to minimax is that the **MAX** player will only update the value of alpha and the **MIN** player will only update the value of beta. While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta. We will only pass alpha and beta to the child nodes.

Constraint Satisfaction Problems

What if solutions are described by a number of constraints that the state must satisfy? For problems like this we can use *constraint satisfaction*.

Definition 21 (constraint satisfaction)**.** Constraint satisfaction is a technique where a problem is solved when its solution satisfies certains or rules of the problem.

Components are:

1. A State, defined by variables X_i with d values from domain D_i.
2. A Goal test, defined as a set of constraints c specifying allowable combinations of values for subsets of variables.

Solving Constraint Satisfaction Problems requires

1. A state space.
2. The notion of the solution.

A state in state-space it not a "blackbox" anymore (as in standard search) but defined by assigning values to some or all variables such as

$$X_1 = v_1, X_2 = v_2$$

Can be done in three ways:

1. Consistent/Legal Assignment: An assignment which does not violate an constraint or rules.
2. Complete assignment: An assignment where every variable is assigned with a values and the solution to the CSP remains consistent.
3. Partial assignment: An assignment which assigns values to some of the variables only.

Since we do not care about the path to a solution and solutions are described by constraints, we can do better than search trees but still maintain a data structure. This structure is the *constraint graph*. In a constraint graph every variable is represented by a node and every edge indicates a constraint between them.

Naive Search

One way to solve CSPs is to simply 'search' for a solution. We successively assign values to a variable, check all constraints and if a

constraint is violated we backtrack. We do this until all variables have assigned values. We do this by mapping the CSPs into a search problem. The nodes are the assignments of values to a subset of the variables, the neighbors of a node are nodes in which values are assigned to one additional variable. As a start node we take an empty assignment and as a goal node is a node which assigns a value to each variable and satisfies all constraints.

Complexity of Naive Search:

We have n variables, so all solutions are at depth n in the search tree. All variables have v possible values. At level 1 we have $n \cdot v$ possible assignments, since we can choose one of n variables and one of v values for it. At level 2 we have $(n-1) \cdot v$ possible assignments for each previously unassigned variable, since we can choose one of the remaing $n-1$ variables and on the v values for it. In general: branching factor at depth l: $(n-l+1) \cdot v$. Therefore the search tree has $n! v^n$ leaves.