3 SOLVING PROBLEMS BY SEARCHING

- A goal-based agent aims at solving problems by performing actions that lead to desirable states
- Let us first consider the uninformed situation in which the agent is not given any information about the problem other than its definition
- In blind search only the structure of the problem to be solved is formulated
- The agent aims at reaching a goal state
- The world is static, discrete, and deterministic

- The possible states of the world define the state space $\Sigma$ of the search problem

In the beginning the world is in the initial state $s_1 \in \Sigma$
- The agent aims at reaching one of the goal states $G \subseteq \Sigma$
- Quite often one uses a successor function $S: \Sigma \rightarrow P(\Sigma)$ to define the agent's possible actions
- Each state $s \in \Sigma$ has a set of legal successor states $S(s)$ that can be reached in one step
- Paths have a non-negative cost, most often the sum of costs of steps in the path
The definitions above naturally determine a directed and weighted graph.

The simplest problem in searching is to find out whether any goal state can be reached from the initial state $s_1$.

The actual solution to the problem, however, is a path from the initial state to a goal state.

Usually one takes also the costs of paths into account and tries to find a cost-optimal path to a goal state.

Many tasks of a goal-based agent are easy to formulate directly in this representation.

For example, the states of the world of 8-puzzle (a sliding-block puzzle) are all $9!/2 = 181,440$ reachable configurations of the tiles.

Initial state is one of the possible states.

The goal state is the one given on the right above.

Possible values of the successor function are moving the blank to left, right, up, or down.

Each move costs one unit and the path cost is the total number of moves.
Donald Knuth’s (1964) illustration of how infinite state spaces can arise.

**Conjecture:** Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

\[
\sqrt{\sqrt{\sqrt{(4!)!}}} = 5
\]

- **States** \( \Sigma \): Positive numbers
- **Initial state** \( s_1 \): 4
- **Actions:** Apply factorial, square root, or floor operation
- **Transition model:** As given by the mathematical definitions of the operations
- **Goal test:** State is the desired positive integer

**Search Tree**

- When the search for a goal state begins from the initial state and proceeds by steps determined by the successor function, we can view the progress of search in a tree structure.
- When the root of the search tree, which corresponds to the initial state, is expanded, we apply the successor function to it and generate new search nodes to the tree — as children of the root — corresponding to the successor states.
- The search continues by expanding other nodes in the tree respectively.
- The search strategy (search algorithm) determines in which order the nodes of the tree are expanded.
The node is a data structure with five components:

- The state to which the node corresponds,
- Link to the parent node,
- The action that was applied to the parent to generate this node, and
- The cost of the path from the initial state to the node

Global parameters of a search tree include:

- $b$ (average or maximum) branching factor,
- $d$ the depth of the shallowest goal, and
- $m$ the maximum length of any path in the state space
3.4 Uninformed Search Strategies

- The search algorithms are implemented as special cases of normal tree traversal.
- The time complexity of search is usually measured by the number of nodes generated to the tree.
- Space complexity, on the other hand, measures the number of nodes that are maintained in the memory at the same time.
- A search algorithm is complete if it is guaranteed to find a solution (reach a goal state starting from the initial state) when there is one.
- The solution returned by a complete algorithm is not necessarily optimal: several goal states with different costs may be reachable from the initial state.

3.4.1 Breadth-first search

- When the nodes of the search tree are traversed in level-order, the tree gets searched in breadth-first order.
- All nodes at a given depth are expanded before any nodes at the next level are expanded.
- Suppose that the solution is at depth \( d \)
- In the worst case we expand all but the last node at level \( d \).
- Every node that is generated must remain in memory, because it may belong to the solution path.
- Let \( b \) be the branching factor of the search.
- Thus the worst-case time and space complexities are

\[
b + b^2 + b^3 + \ldots + b^d = O(b^d)\
\]
In general, the weakness of breadth-first search is its exponential (in depth) time and space usage. In particular, the need to maintain all explored nodes causes problems. For example, when the tree has branching factor $10$, nodes are generated 1 million per second and each node requires 1,000 bytes of storage, then:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 µsec</td>
<td>107 kB ($10^{3}$)</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 µsec</td>
<td>10.6 MB ($10^{6}$)</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 sec</td>
<td>1 GB ($10^{9}$)</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 h</td>
<td>10 teraB ($10^{12}$)</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petaB ($10^{15}$)</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petaB</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 exaB ($10^{18}$)</td>
</tr>
</tbody>
</table>
• Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node.

• On the other hand, to this special case of uniform-cost search, we could also apply the greedy algorithm, which always expands the node with the lowest path cost.

• If the cost of every step is strictly positive the solution returned by the greedy algorithm is guaranteed to be optimal.

• The space complexity of the greedy algorithm is still high.

• When all step costs are equal, the greedy algorithm is identical to breadth-first search.

3.4.3 Depth-first search

• When the nodes of a search tree are expanded in preorder, the tree gets searched in depth-first order.

• The deepest node in the current fringe of the search tree becomes expanded.

• When one branch from the root to a leaf has been explored, the search backs up to the next deepest node that still has unexplored successors.

• Depth-first search has very modest memory requirements.

  • It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.

  • Depth-first search requires storage of only $b^m$ nodes.
Using the same assumptions as in the previous example, we find that depth-first search would require 156 kB (instead of 10 exaB) at depth 16 (7 trillion times less).

- If the search tree is infinite, depth-first search is not complete.
- The only goal node may always be in the branch of the tree that is examined the last.
- In the worst case also depth-first search takes an exponential time: \( O(b^d) \).
- At its worst \( m = d \), the time taken by depth-first search may be much more than that of breadth-first search.
- Moreover, we cannot guarantee the optimality of the solution that it comes up with.
3.4.4 Depth-limited search

- We can avoid examining unbounded branches by limiting the search to depth \( \ell \).
- The nodes at level \( \ell \) are treated as if they have no successors.
- Depth-first search can be viewed as a special case with \( \ell = m \).
- When \( d \leq \ell \) the search algorithm is complete, but in general one cannot guarantee finding a solution.
- Obviously, the algorithm does not guarantee finding an optimal solution.
- The time complexity is now \( O(b^\ell) \) and the space complexity is \( O(b\ell) \).

3.4.5 Iterative deepening search

- We can combine the good properties of limited-depth search and general depth-first search by letting the value of the parameter \( \ell \) grow gradually.
- E.g. \( \ell = 0, 1, 2, \ldots \) until a goal node is found.
- In fact, thus we gain a combination of the benefits of breadth-first and depth-first search.
- The space complexity is controlled by the fact that the search algorithm is depth-first search.
- On the other hand, gradual growth of the parameter \( \ell \) guarantees that the method is complete.
- It is also optimal when the cost of a path is non-decreasing function of the depth of the node.
3.4.6 Bidirectional search

- If node predecessors are easy to compute — e.g., $\text{Pred}(n) = S(n)$ — then search can proceed simultaneously forward from the initial state and backward from the goal.
- The process stops when the two searches meet.
- The motivation for this idea is that $2b^{d/2} \ll b^d$.

- If the searches proceeding to both directions are implemented as breadth-first searches, the method is complete and leads to an optimal result.
- If there is a single goal state, the backward search is straightforward, but having several goal states may require creating new temporary goal states.

An important complication of the search process is the possibility to expand states that have already been expanded before.

- Thus, a finite state space may yield an infinite search tree.
- A solvable problem may become practically unsolvable.
- Detecting already expanded states usually requires storing all states that have been encountered.
- One needs to do that even in depth-first search.
- On the other hand, pruning may lead to missing an optimal path.
Searching with Partial Information

- Above we (unrealistically) assumed that the environment is fully observable and deterministic.
- Moreover, we assumed that the agent knows what the effects of each action are.
- Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in.
- Its percepts provide no new information after each action.
- In a more realistic situation the agent’s knowledge of states and actions is incomplete.
- If the agent has no sensors at all, then as far as it knows it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.

- An agent without sensors, thus, must reason about sets of states that it might get to, rather than single states.
- At each instant the agent has a belief of in which state it might be.
- Hence, the action happens in the power set $P(\Sigma)$ of the state space $\Sigma$, which contains $2^{2^{2}}$ belief states.
- A solution is now a path to that leads to a belief state, all of whose members are goal states.
- If the environment is partially observable or if the effects of actions are uncertain, then each new action yields new information.
- Every possible contingency that might arise during execution need considering.
• The cause of uncertainty may be another agent, an adversary
• When the states of the environment and actions are uncertain, the agent has to explore its environment to gather information
  • In a partially observable world one cannot determine a fixed action sequence in advance, but needs to condition actions on future percepts
  • As the agent can gather new knowledge through its actions, it is often not useful to plan for each possible situation
  • Rather, it is better to interleave search and execution

3.5 Informed (Heuristic) Search Strategies

• We now consider informed search that uses problem-specific knowledge beyond the definition of the problem itself
• This information helps to find solutions more efficiently than an uninformed strategy
• The information concerns the regularities of the state space
• An evaluation function $f(n)$ determines how promising a node $n$ in the search tree appears to be for the task of reaching the goal
• Best-first search chooses to expand the node that appears by evaluation function to be most promising among the candidates
• Traditionally, one aims at minimizing the value of function $f$
Heuristic Search Strategies

- A key component of an evaluation function is a heuristic function $h(n)$, which estimates the cost of the cheapest path from node $n$ to a goal node.
- In connection of a search problem “heuristics” refers to a certain (but loose) upper or lower bound for the cost of the best solution.
- Goal states are nevertheless identified: in a corresponding node $n$ it is required that $h(n) = 0$.
- E.g., a certain lower bound — bringing no information — would be to set $h(n) = 0$.
- Heuristic functions are the most common form in which additional knowledge is imported to the search algorithm.

3.5.1 Greedy best-first search

- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- Thus, the evaluation function is $f(n) = h(n)$.
- E.g. in minimizing road distances a heuristic lower bound for distances of cities is their straight-line distance.
  
  - Greedy search ignores the cost of the path that has already been traversed to reach $n$.
  - Therefore, the solution given is not necessarily optimal.
  - If repeating states are not detected, greedy best-first search may oscillate forever between two promising states.
Because greedy best-first search can start down an infinite path and never return to try other possibilities, it is incomplete.

Because of its greediness the search makes choices that can lead to a dead end; then one backs up in the search tree to the deepest unexpanded node.

Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end.

The worst-case time and space complexity is $O(b^m)$.

The quality of the heuristic function determines the practical usability of greedy search.

### 3.5.2 A* search

A* combines the value of the heuristic function $h(n)$ and the cost to reach the node $n$, $g(n)$.

Evaluation function

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

thus estimates the cost of the cheapest solution through $n$.

A* tries the node with the lowest $f(n)$ value first.

This leads to both complete and optimal search algorithm, provided that $h(n)$ satisfies certain conditions.
Optimality of $A^*$

- Provided that $h(n)$ never overestimates the cost to reach the goal, then in tree search $A^*$ gives the optimal solution
  
  - Suppose $G_2$ is a suboptimal goal node generated to the tree
  - Let $C^*$ be the cost of the optimal solution
  - Because $G_2$ is a goal node, it holds that $h(G_2) = 0$, and we know that $f(G_2) = g(G_2) > C^*$
  - On the other hand, if a solution exists, there must exist a node $n$ that is on the optimal solution path in the tree
  - Because $h(n)$ does not overestimate the cost of completing the solution path, $f(n) = g(n) + h(n) < C^*$
  - We have shown that $f(n) < C^* < f(G_2)$, so $G_2$ will not be expanded and $A^*$ must return an optimal solution

- For example straight-line distance is a heuristic that never overestimates road distance between cities
- In graph search finding an optimal solution requires taking care that the optimal solution is not discarded in repeated states
- A particularly important special case are consistent (or monotonic) heuristics for which the triangle inequality holds in form
  
  $$h(n) \leq c(n, a, n') + h(n'),$$

  where $n' \in S(n)$ (the chosen action is $a$) and $c(n, a, n')$ is the step cost

- Straight-line distance is also a monotonic heuristic
A* using a consistent heuristic $h(n)$ is optimal also for graph search.

If $h(n)$ is consistent, the values of $f(n)$ along any path are non-decreasing.

Suppose that $n'$ is a successor of $n$ so that $g(n') = g(n) + c(n, a, n')$ for some action $a$.

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

Hence, the first goal node selected for expansion (in graph search) must be an optimal solution.

In looking for a solution, A* expands all nodes $n$ for which $f(n) < C^*$, and some of those for which $f(n) = C^*$.

However, all nodes $n$ for which $f(n) > C^*$ get pruned.

It is clear that A* search is complete.

A* search is also optimally efficient for any given heuristic function, because any algorithm that does not expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

Despite being complete, optimal, and optimally efficient, A* search also has its weaknesses.

The number of nodes for which $f(n) < C^*$ for most problems is exponential in the length of the solution.
3.5.3 Memory-bounded heuristic search

- Once again the main drawback of search is not computation time, but rather space consumption.
- Therefore, one has had to develop several memory-bounded variants of A*.
- IDA* (Iterative Deepening A*) adapts the idea of iterative deepening.
  - The cutoff used in this context is the $f$-cost ($g + h$) rather than the depth.
  - At each iteration the cutoff value is the smallest $f$-cost of any node that exceeded the cutoff on the previous iteration.
- Subsequent more modern algorithms carry out more complex pruning.

3.6 Heuristic functions

In 8-puzzle we can define the following heuristic functions, which never overestimate:

- $h_1$: the number of misplaced tiles: any tile that is out of place must be moved at least once to obtain the desired configuration.
- $h_2$: The sum of Manhattan distances of tiles from their goal position: the tiles need to be transported to their goal positions to reach the desired configuration.

In the initial configuration all tiles are out of their place: $h_1(s_1) = 8$.
- The value of the second heuristic for the example is:
  $$3 + 1 + 2 + 2 + 3 + 3 + 2 = 18$$