swissQuant
Mastering Complexity

Mathematical Challenge December 2017

Search algorithms for artificial intelligence systems

References

- [1] Russell, Stuart, Peter Norvig, "Artificial Intelligence. A modern approach." *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995): 27.

Description

Artificial intelligence is nowadays a buzzword and various distortions of its original meaning are common. Indeed, even the original and technical meaning is not univocal. AI may refer to the development of artificial systems showing either human like or rational behavior. Furthermore, it can be distinguished between systems that try to achieve this via either human-like or not human-like internal structures.

Search methods are algorithms that efficiently explore the space of possible system states with the aim of either finding an acceptable state (defined by a set of constraints) or an optimal state (defined by a cost function and possibly some constraints). Search methods are therefore a crucial ingredient of AI systems trying to achieve rational behavior via a not necessarily human-like internal structure. In particular, search methods are at the core of problem solving agents with rational behavior.

In general a search problem is defined w.r.t.

- state space: describing the state of the system
- action space: listing decisions that can be made to influence the system
- successor function: specifying how the state changes given an action
- cost function: quantifying the attractiveness of selected paths
- target state(s): explicitly listed or implicitly identified by some checks.

Search methods try to identify a solution, i.e. a sequence of actions, defining a path of states connecting the initial state to the target state(s), or an optimal solution, i.e. a solution with lowest costs.

While the standard example where search methods are applied is route planning, there are a multitude of other applications like automatic assembly sequences, industrial product design and others.

We will here follow the reference [1] and assume:

- The states of the system are discrete and observable
- The successor function is known and deterministic
- The target space is known and fixed

Under these assumptions the solution of the search problem is a fixed sequence of actions, i.e. given such environment and problem, the search algorithm provides a sequence of actions to be implemented that ensure the achievement of the given goal (note that the search method is applied during the planning phase which is completed before moving to the execution phase).

To simplify our description we further assume that the cost of moving from one state to another is constant. In this case, costs match path lengths. Some algorithms relying on path length can be adapted to handle not constant costs. Note, however, that in the general case very long paths with limited costs may exist and properties of an algorithm may be relevantly different after adaptation to general costs.

The performance of the different algorithms is characterized by the following properties:

- Completeness: is the algorithm able to find a solution if this exists?
- Optimality: if many solutions exist, does the algorithm provide the optimal one?
- Time-complexity: how long does the algorithm take to find a solution?
- Space complexity: how much memory does the algorithm require?

The general scheme of search methods consists in

- keeping a list (queue) of potential paths initialized with the length one path consisting of just the initial state
- select a path from the queue and replace it with all the paths corresponding to the selected path followed by the outcome of a possible action
- perform this until the target is reached

Breadth-first search (BFS) is a complete algorithm which is generally applicable and ensures optimality (when adjusted for not constant costs, is known as uniform cost search). It consists in selecting for expansion the path in the queue that has the shortest length. BFS, while ensuring optimality, is not time and especially not memory efficient indeed the costs are exponential $O(b^d)$ (assuming constant branching factor b, optimal solution depth of d and equal step costs) .

Two possible ways to reduce the memory requirements are: *iterative deepening search* (IDS) and *bidirectional search* (BS). In IDS a maximum depth d_n is increased at each iteration n. For each iteration a full path search is performed by selecting from the queue the path with longest length but lower than d_n. BS on the other side applies two BFS searches simultaneously, one from the initial state forwards and one from the target state backwards. IDS memory cost is $O(bd)$ and the one of BS $O(b^{d/2})$. However IDS performance strongly depends on the adequacy of the sequence d_n and its time complexity is still $O(b^d)$, whereas BS has time complexity $O(b^{d/2})$ but is not well suited if the target is implicitly defined.

To decrease time complexity and increase the search algorithm efficiency in general, we have to exploit some additional information about the problem besides the simple problem formulation. I.e. we have to move from uninformed search strategies like BFS, IDS and BS to so called *information search strategies*.

A particular class of such methods consists in *heuristic function based search strategies*. These leverage problem knowledge casted in the form of a *heuristic function* $h(i)$ which estimates the minimal distance from the state i to the target. Replacing in BFS the length of each path $l(path)$ with $f(path) = l(path) + h(path(end))$ (where $path(end)$ is the current final state of the path), we obtain the A^* *search algorithm*. Assuming that the heuristic has a relative error of $\epsilon = (h - l)/l$ the time complexity becomes $O(b^{\epsilon d})$.

The A^* search algorithm has optimality guarantees provided that the heuristic function h is *consistent*, i.e. for every nodes n and n' and path $n \to n'$

$$h(n) \leq l(n \to n') + h(n')$$

Consistent heuristics can be obtained a.o. in the following two ways:

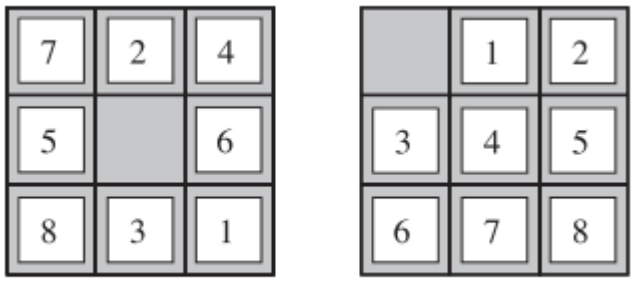
a. $h(n) = l_{relaxed}(n \rightarrow target)$
b. $h(n) = l(n \rightarrow subtarget)$

where $l_{relaxed}$ is the actual distance from the target for a "relaxed" problem (where the allowed set of actions is a superset of the original one) and a $subtarget$ is a state just partially satisfying the target conditions.

The A^* search algorithm can be modified exactly as the BFS was modified into the IDS to improve memory efficiency, obtaining the iterative-deepening A^*. However, there exist better alternatives like the *simplified memory-bounded* A^* (SMA). SMA proceeds as A^* until the memory bound is reached, at that point it drops the path with the highest f-value from the queue. Note however that SMA may be forced to regenerate dropped paths when the current best path length becomes larger than the f-value of the previously dropped ones. Therefore, time complexity for hard problems may be relevantly higher than the one of A^*. Finally, if the solution is not within a length smaller than the memory bound, it will not be reached.

Questions:

- [Q1] Show that for consistent heuristics, A^* search guarantees optimality
- [Q2] Consider the 8-puzzle, which consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state:



(source [1])

Determine two heuristics of the type

a. $h_a(n) = l_{relaxed}(n \rightarrow target)$
b. $h_b(n) = l(n \rightarrow subtarget)$

Compare the time and memory complexity of the following approaches

a. BFS
b. IDS
c. A^* based on h_a
d. A^* based on h_b
e. SMA based on h_a
f. SMA based on h_b

How do the complexity scale considering a 15-puzzle (4x4 board)

We look forward to your opinions and insights.

Best Regards,

swissQuant Group Leadership Team