Heuristic Search

Please do not circulate

1 Problem Definition

We will now look at what we call as *search* problem (as well as algorithms to solve it), which are the most preliminary when it comes to rational agents. Table 1 is a comparison of the various classes of problems (including search) that a rational agent might have to encounter. The following can be

Problem Class	Given	Remarks
Search	$\langle S, A, C, T\rangle$ T deterministic s_1 and G	Offline no learning open loop decision
Markov Decision Processes (MDP)	$\langle S, A, R, T\rangle$ T stochastic	Offline no learning closed loop decision
Reinforcement Learning (RL)	$\langle S, A, -, -\rangle$ T stochastic R, T not known	Online learning closed loop decision

Table 1: A comparison of the various class of problems that a rational agent might have to encounter in practice. Here s_1 is a *start* state and $G \subseteq S$ is a set of *goal* states.

inferred from Table 1:

1. The problem classes are contained as follows: RL $\supseteq$ MDP $\supseteq$ Search.
2. All the 3 quantities namely S, R, T are made available to the agent. All that the agent now needs is a *search-program*, which will use the given information and compute the optimal agent function, and further this can be achieve in an *offline* manner even without interacting with the environment.
3. Note that T is deterministic which implies open loop control is possible, i.e., given a start state and a sequence of actions, the current state is completely determined, i.e., the agent knows the current state s_t even without any percept from the environment.
4. Due the fact that T is deterministic it also follows that there is no need to collect samples or data, and hence there is no learning involved.

Formally, the search problem is defined by $\langle S, A, C, T\rangle$, a start state s_1 and a set of goal states given by $G \subseteq S$. Note that C here stands of cost and is not a reward map (the reason as to why cost is natural in this setting will be clear as we proceed with the rest of the discussion). The aim of the agent (from now on we will just use agent instead of rational agent) is to reach the goal from the start state while incurring minimum cost. This can be mathematically formulated as below:

$$\begin{aligned} \text{Objective} &: \operatorname{argmin}_{a_1,a_2,\ldots,a_N} \sum_{t=1}^{N} C(s_t, a_t) \\ \text{Constraints} &: s_{t+1} = T(s_t, a_t) \\ &: C(s,a) > 0, \forall s \notin G \\ &: C(s,a) = 0, \forall s \in G \\ &: \text{given initial state } s_1 \text{and goal set } G, s_N \in G \end{aligned} \tag{1}$$

Objective of the search problem is to *minimize* the cost $\sum_{t=1}^{N} C(s_t, a_t)$ over the set of all possible action sequences $a_1, \ldots, a_N$ that take the agent from the start state to the goal state. In what follows let us break this down.

Need for a search program: What does the agent need? It needs to know what actions to make so that goal is reached from the start state. So given a start state s_1, firstly the agent needs to choose an action $a_1 = ?$. Say it chooses some action $a_1 = a^i$ (where $a^i \in A = a^1, a^2, \ldots, a^{|A|}$). Now since T is deterministic and given to the agent, the next state can also be computed, i.e., $s_2 = T(s_1, a_1 = a^i)$. The agent then has to decide on $a_2 =?$, and then it can compute s_3, and so on. However, the agent does not know what is the correct sequence of actions. A way to compute this correct sequence of actions is to exhaustively search for all combination of actions. Intuitively speaking the ? marks in $a_1 =?, a_2 =?, \ldots, a_N =?$ have to be filled with various possible actions, and the agent has to check which combination *clicks*, in a way similar to cracking a combination lock. The search program is just a formal way to performing this exhaustive search.

2 Rational Agents: Design Principles

Before we proceed to specifying the search algorithms, we now mention four important design principles related to rational agents as below:

1. One step expansion
2. Greedy/Optimism
3. Problem structure exploitation
4. Value System

Almost all algorithms for rational agents are built using these four design principles and search algorithms are no exception. We will now look at the search program which is search algorithm + data structure.

3 Search Program: Algorithm and Data Structure

The search program can be understood by simultaneously looking at Figure 1 and Algorithm 1. The search program maintains the following data structures

1. Search Tree: This contains the explored sets. Each node in the tree stores information such as its state, the action from the parent, the total cost in reaching the corresponding state from the start state, the depth of the node in the search tree, and a *value* (see design principle).
2. The leaf nodes of the search tree are called *fringe*. The fringe is also stored in a priority queue (Q in Algorithm 1) and the priority is given based on the value of that node (more on values later).
3. When the goal is not yet found (**??** 7-Algorithm 1), we have to expand the node (this is the one-step expansion principle). This is done by trying out all the actions a^1 to $a^{|A|}$, and then computing $\{T(s, a^1, \ldots, T(s, a^{|A|}))\}$. Now the newly found nodes are the leaf nodes and hence need to be added to the fringe (**??** 9-Algorithm 1).
4. Note that a node is removed from the Queue first and then it is expanded. Thus the fringe set keeps changing throughout the algorithm. Also, if one node is removed from the priority queue, then it is possible that a maximum of $|A|$ new nodes are added (assuming that $T(s, a^1), \ldots, T(s, a^{|A|})$ are distinct.

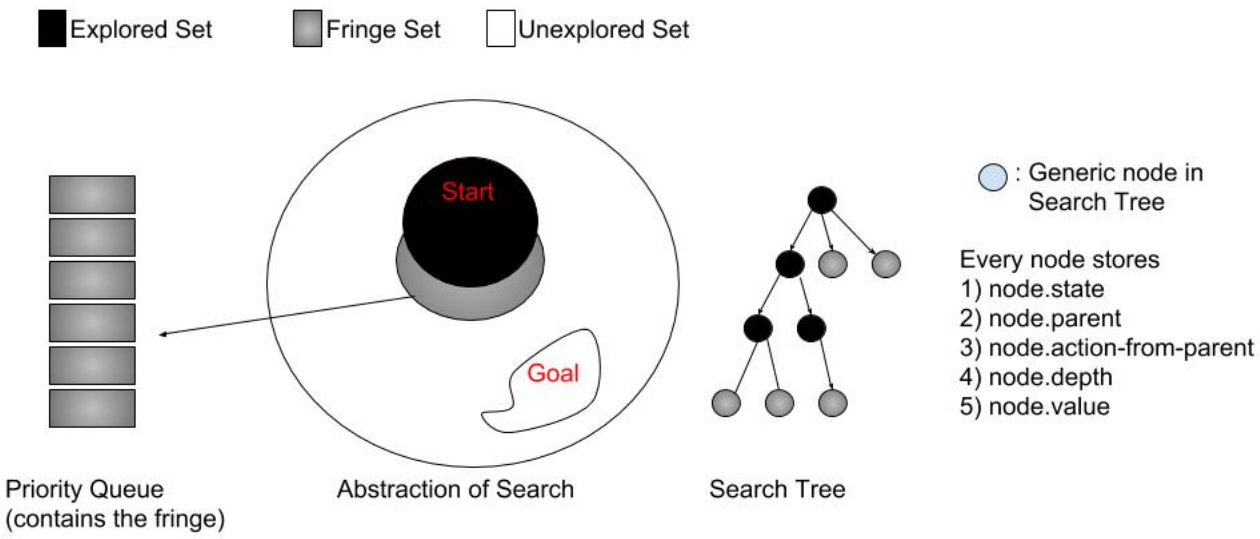

Figure 1: Idea of Search

Algorithm 1 Search Algorithm: Here, the search tree is grown via the $node(\cdot)$ function call. The INSERT and REMOVE operations belong to the priority queue.

Input:$\langle S, A, C, T\rangle, s_1 = s^i, G$
Q.INSERT($node(s^i)$)
while Q not-empty **do**
  node=Q.REMOVE()
  s=node.state
  if $s \in G$ **then**
    Goal found (Return Optimal Path)
  else
    for $s' \in \{T(s, a^1), \ldots T(s, a^{|A|})\}$ (Expansion of s) **do**
      Q.INSERT($node(S')$) (Expansion of s)
    end for
  end if
end while
Output: $a_1, a_2, \ldots, a_N$,

4 Greedy, Optimism, Problem Structure, Completeness and Optimality

Greedy/Optimism: Amongst the various fringe elements which one to expand first? This question is settled by the greedy/optimism principle. We will now look at the interplay of the principles of *value* and *optimism/greediness*. We mentioned that the priority queue is implemented with the value of the node. When we remove an element from the priority queue (**??** 3-Algorithm 1), the element with *minimum* value is removed. This is known as the optimism principle, i.e., to be greedy by picking that fringe node whose value is the *minimum* for expansion.

We are yet to specify what should the value of a node be. We will now elaborate on this point, since the values decide which nodes are prioritized for expansion. The various *value*-systems will lead to different search algorithms as discussed below:

- **Breadth-First-Search** (BFS): Suppose we maintained a counter, say i and store it as the value of a node. However, each time we insert a node (we assign $node.value = i$), we also increment the counter $i = i + 1$. Using priority of i will mean a FIFO (first-in-first-out) queue.

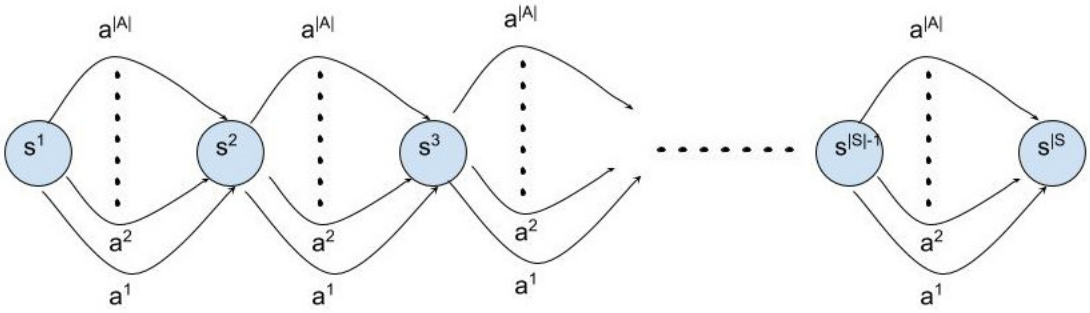

Figure 2: Show that complexity is $O(|A|^{|S|})$

- **Depth-First-Search** (DFS): Here each time we insert a node (we assign $node.value = -i$), we also increment the counter $i = i + 1$. Note that using $-i$ to be the priority will mean a LIFO (last-in-first-out) queue.

- **Total-Cost-Till-Now**: We let the $node.value = g(n)$, where $g(n)$ is the total cost from the start state to the state of the current node.

- **Estimated-Cost-To-Go**: Suppose we are made available of a heuristic function $h(n)$ which is an estimate for the *total cost to go*, i.e., it gives a clue of what it would take to reach the goal from the state of any given node. One can use $node.value = h(n)$.

- **Estimated-Total-Cost**: Here $node.value = Theg(n) + h(n)$, which is an estimate of the total cost to the goal from the start state via the state at the current node. Using this as the value gives rise to the A^* algorithm.

We now have to comment about completeness and optimality. We say that an algorithm is complete when it manages to find *a solution* and optimal when it finds *the least cost solution* to the search problem Equation (1). Let us now see how the 5 different strategies based on the optimism/greedy principle using the 5 different DFS systems fair.

- BFS is complete. It is also optimal when all the step-cost are the same. The worst case for BFS is given by the following example (see Figure 2), when all the actions have same cost, and $s^{|S|}$ is the goal state. The complexity is of the order $O(|A|^{|S|})$.

- DFS is not complete when there are loops or when the state space is infinite. However, DFS does not store a lot of nodes and its memory requirements are lesser in comparison to BFS.

- Using $g(n)$ is optimal and complete.

- Using $h(n)$ is complete but not optimal.

- Using $g(n) + h(n)$ is optimal when $h(n)$ is admissible i.e., $h(n) \leq total - cost - to - go$.

A way to stop loops is by stopping the explored set from getting added to the fringe. This new algorithm Algorithm 2 is a modification of Algorithm 1 and is given below:

Algorithm 2 Search Algorithm with **Explored-List**.

Input:$\langle S, A, C, T\rangle, s_1 = s^i, G$
Q.INSERT($node(s^i)$)
while Q not-empty **do**
  node=Q.REMOVE()
  s=node.state
  if $s \in G$ **then**
    Goal found (Return Optimal Path)
  else
    for $s' \in \{T(s, a^1), \ldots T(s, a^{|A|})\}$ (Expansion of s) **do**
      IF $s' \notin$ Explored set, Q.INSERT($node(s')$) (Expansion of s)
    end for
    Add s to Explore Set
  end if
end while
Output: $a_1, a_2, \ldots, a_N$,

Please refer to the example used in class to understand how Algorithm 1 and Algorithm 2 would work with the 5 different search strategies.

5 Heuristics: Problem Structure Exploitation

The heuristics serve as an additional *information/clue* about the problem. In certain cases such as say computing shortest path in a road/rail network, the Euclidean distance could serve as a consistent and admissible heuristic. In any problem that involves minimization, the solution to the relaxed version of the problem can serve as a heuristic. For example, in the road network the relaxation is to think that there are roads between any two points and the cost of that path is the Euclidean distance between the two points. Thus the Euclidean distance serves as an admissible heuristic.

When the heuristics are a good guess of the actual total cost to go, then they can lead the algorithm in the right path and the solution can be found. The case $h(n) = 0$ is like the heuristic being absent, and the agent does have an idea of what the actual total cost would be. If $h(n)$ is the exact total cost to go, then it is the best possible scenario, because the algorithm will expand only those nodes along the optimal path and will not waste time exploring any other nodes. Thus the goodness of a heuristic can be measured by how strong a clue it gives so that the algorithm does not have to visit unnecessary paths.

Best of DFS and BFS: We know that DFS wins on memory requirements and total-cost-till-now (basically BFS for non-uniform costs) wins on completeness. The A^* tries to achieve the best of both worlds (DFS and BFS), i.e., when the heuristic is good, it behaves like a DFS algorithm, i.e., it tries to go deeper and deeper along the optimal path, and yet at the same time, A^* can also correct if the heuristic was misleading as long as the heuristic was admissible (and consistent in the case of Algorithm 2). Please use the example given in the class to understand how A^* can recover itself even when the heuristics are slightly misleading.

Please do use the problems in the Problem Set 1 to get more insights about the heuristics.