Penn CIS 5210- AI: A Modern Approach - Chapter 3

Jonathon Delemos - Dr. Chris Callison Burch

June 11, 2025

This course investigates algorithms to implement resource-limited knowledge-based agents which sense and act in the world. Topics include, search, machine learning, probabilistic reasoning, natural language processing, knowledge representation and logic. After a brief introduction to the language, programming assignments will be in Python. — Description of CIS 4210/5210 in course catalog

1 Chapter Three: Solving Problems by Searching

1.1 3.0 introduction

Problem-Solving Agent is when the correct action to take is not offered by the greedy solution. **Searching** is now required.
Problem-Solving Agents use **atomic** representation. This means that A leads to B. **Planning agents** use factored or structured representation. A factored representation splits all the variables up in values. Think of this like an array of information. Two different factored reprensations might share certain elements, but be different vectors. More on that in chapter 2.
Let's briefly recap structured representation.
Structured reprensation is a bit like a relational database for storing data that interacts with each other.
Consequantiliasm is the idea that the agent flows through a series of states. The sequence of states is determined to be desirable based off the **performance measure.**

1.2 3.1 - Problem-Solving Agents

Unknown - In an unknown environment, the agent is forced engage in random behavior. **Goals** are very important to the agent. Clearly defined goals moreso. In the book, the author describes travelling through Romania, specifically from arad to bucharest. Before taking any actions, the agent uses a **search** tree to find a solution. When a solution is found, the agent can ignore it's percepts and engage in the execution. This is called an **open-loop** system. Example might be driving from A -¿ B -¿ C. You don't really need to look at the map again. If there is a chance the road conditions may change, you might want to use a **closed-loop system**. This is more non-deterministic. The strategy could change depending on which precepts arrive.

1.3 3.2 - Search problems and solutions

Search problem can be formally defined as follows:

- States the environment can be in.
- Initial State the agent starts in.
- Goal states. There are can alternative states, sometimes the goal is defined by a property that applies to many states. If I write "goal", that could mean a variety of goal states.
- Actions available to the agents. ACTIONS(Arad) = (toSibiu,ToTimisoara,ToZerind)
- Transition Model. RESULT(Arad,ToZerind) = ToZerind

- Action Cost Function ACTIONCOST(Arad, Zerind, 3) where c(s,a,s') and s' is the cost of doing s-¿a.

Touring Problems are search problems. You can already see how they might need the states previously described. An example might be the **Travelling Salesman** problem. Examples of other solving problems through searching include VLSI layout, robot navigation, automatic assembly sequencing, and protein design.

1.4 3.3 - Search Algorithms

Search Algorithm takes a search problem as an input and returns a solution. Throughout this chapter, we investigate **search trees** over state space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. The **state space** describes set of states in the world, and the actions that allow transitions from one to another. The **search tree** describes paths between these states, reaching towards the goal. **Child nodes** are nodes in a search that were generated from the parent node.

Algorithm 1 Best-First Search - Pseudo

```
function BestFirstSearch(problem, $f$)
    $node \leftarrow$ Node($problem.initial\_state$)
    $frontier \leftarrow$ priority queue ordered by $f(node)$
    insert $node$ into $frontier$
    $explored \leftarrow \emptyset$
    while $frontier$ is not empty do
        $node \leftarrow frontier.pop()$
        if $problem.goal\_test(node.state)$ then
            return $solution(node)$
        $explored \leftarrow explored \cup \{node.state\}$
        for all $action \in problem.actions(node.state)$ do
            $child \leftarrow child\_node(problem, node, action)$
            if $child.state \notin explored$ and $child \notin frontier$ then
                insert $child$ into $frontier$
            else if $child$ is in $frontier$ with higher $f$-value then
                replace that node with $child$ in $frontier$
    return failure
```

That's a rough algorithm for best-first search. But let's figure out how to structure these. I want to try to construct one like I might construct a deterministic finite automata. Listed below is the structure of the search data.

Search Data Structures:

- node.State: the state to which the node corresponds
- node.Parent: the node in the tree that generated this node;
- node.Action: the action that was applied to the parents state to generate this node;
- node.PathCost: the total cost of the path from the initial state to this node. In math, we use g(node) as a synonym for the Path-Cost.

The frontier is typically stored in a queue. Three kinds of queues are typically used in search algorithms, priority, LIFO, FIFO. If a search tree has a loop or cycle, it is considered infinite. We call a search algorithm a **graph search** if it checks for redundant paths. Imagine a small grid where all locations fit in memory. An algorithm that doesn't check for redundant paths is a **tree-like-search**. Finally, we can define a **redundant** path as a worse way to accomplish the same objective. Example: two paths reach the final state and one is faster. The second path is

Algorithm 2 Best-First Search - Python

```
function BestFirstSearch(problem, $h$)
      # Create the initial node from the start state
   $node = \text{Node}(problem.initial.state, none, none, 0)$
   $frontier = []$
      # Here we are assigning the frontier to a heap which behaves like pq
   $heapq.heappush = (frontier, (h(node.state), node))$
   $visited = (set())$

      # Time to start the loop
   while $frontier$:
      $node = heapq.heappop(frontier)$
      if $problem.goal_state(node.state)$ :
         return $solution(node)$
      $reached.add(node.State)$
      for $action$ in problem.actions(node.State)
         $child$ = child.node(problem, node, action)
         if $child$ not in visited:
            $heapq$.heappush(frontier, (h(child.state), child))
         return None
```

redundant. Best-first search is a graph search algorithm: it will remove all references to reached we get a tree-like search that uses less memory but will examine redundant paths to the same state. This is slower.
When creating a measure for problem solving performance, we consider:

- **Completeness** : Is the algorithm guaranteed to find a solution.
- **Cost Optimality** : does it find a solution with the lowest path cost of all solutions?
- **Time Complexity** : How long does it take to find a solution?
- **Space Complexity** : How much memory is needed?

Finally, we are going to analyze all the

1.5 3.4 - Uninformed Search Strategies

BFS when all actions have the same cost. Uses FIFO queue. Uses **early goal strategy** that tests whether a node is the goal on every iteration.
Djikstra's Algorithm - This is when actions have different costs. It looks a lot like BFS in evaluation, but it's a tad bit more complex. At each turn it computes the cost of travel plus the cost of the next node; it's really quite brilliant. In comparison, DFS doesn't consider cost. It returns the first solution found. For reference, b is the branching factor and m is the memory. To keep DFS from wandering infinitely, we can use a DEPTH-LIMITED search. This is where we only traverse until a certain depth has been met.
Finally, we are going to compare the various algorithms and their relevant attributes.

Table 1: Comparison of Uninformed Search Strategies

Search Algorithm	Complete?	Optimal?	Time Complexity	Space Complexi
Breadth-First Search	Yes (if b is finite)	Yes (if cost = 1 per step)	$O(b^d)$	$O(b^d)$
Uniform-Cost Search	Yes (if cost $\geq \epsilon$)	Yes	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$
Depth-First Search	No	No	$O(b^m)$	$O(m)$
Depth-Limited Search	No (if $d_l < d$)	No	$O(b^{d_l})$	$O(d_l)$
Iterative Deepening DFS	Yes	Yes (if cost = 1 per step)	$O(b^d)$	$O(d)$
Bidirectional Search	Yes (if goal is reachable)	Yes (if cost = 1 per step)	$O(b^{d/2})$	$O(b^{d/2})$

1.6 3.5 - Informed (Heuristic) Search Strategies

Heuristics is a word we use to describe a hint about the location of the goal. For example, distance from city to city might be used.
Greedy best-first appears to work efficiently, but it can get stuck in loops due to it's inability to calculate the cost of how far it's come already.

- Complete - only in finite spaces
- Time Complexity - O (—V—) - V is the number of vertices, best case branch * memory.
- Space Complexity - O (—V—) - V is the number of vertices, best case branch * memory.
- Cost Optimal - sometimes - It can lead into loops.

A* is a very interesting algorithm. It keeps a running count of the distance travelled plus the distance to the target. If an agent gets caught in a loop, then it will eventually escape because the distance travelled inside plus the distance spent getting there will eventually be less than a different option. Once that happens, it will select another option!

- Complete - Yes
- Time Complexity - O (—V—) - V is the number of vertices, best case branch * memory.
- Space Complexity - O (—V—) - V is the number of vertices, best case branch * memory.
- Cost Optimal - Depends on the properties of the heuristics

Admissibility is a key factor, **admissable heuristics** is one that never over-estimates the goal. Therefore it is admissable in our analysis.
-
Triangle Inequality states that *h(n) ¡ c(n, a, n') + h(n')*. In english, this means that the cost of n to goal is less than or equal to the cost to go from n to n' + n' to the goal. This assumes n' is on the way there. Think of it like a gas station on the way to work. The distance to work shouldn't be greater if you stop at the gas station, or else something is wrong. This is an important proof of consistent heuristics. They will provide reliable metrics for success and are therefore safe to implement.
Search Contours - this is a grouping of costs associated for movement in a specific area. Moving outside the area will always result in a cost increase. However, f(n) = g(n) + h(n) might remain consistent as it tries a few routes that reach the same exceeded level. **Surely expanded nodes** these are nodes where f(n) - price to get to n is less than *C, cost of total solution. A* is efficient because it prunes away search nodes that cannot find an optimal solution. For example, if you know one road leads toward and connects with the city, and one road leads directly away/doesn't connect, you can prune the road that leads directly away. This is called **optimal efficiency.**
-
The downside of A* is that it expands a lot of nodes. If we want to optimize time and space but allow suboptimal solutions, we consider this to be a **satisficing** solution. Applying the idea of a **detour index** is the idea that between two locations there is a straight line distance. Let's call it 10 miles. Due to streets available, the actual travelling distance might be 13 miles. This

means we have a detour index of 1.3. Interestingly, most locations have a detour index of 1.2-1.6. This is where **weighted A* search** comes in. The evaluation function is *f(n) = g(n) + W x h(n)* where W is a weighted heuristic value. The weighted heuristic value could prevent us from exploring many states for instance. In general, if the cost of the solution is C*, then the weight A* cost would be somewhere between C* and *W x C**.
Unbounded and Bounded Searches: finding the solution before *X* resources or finding a solution regardless of the cost.
-
Memory Bounded Search - some algorithms allow for better utilization of available space. **Iterative-Deepening A* Search** helps us eliminate the need to cache all reached states.

1.7 Questions?

This chapter took me literally fucking forever.
Are we expected to memorize the code for these algorithms, or just understand what they do and what their limitations are?

1.8 Summary

In this chapter, we discusssed search algorithms that an agent can use to choose actions in a wide variety of environments. We covered uninformed searches: bfs, dfs, uniform cost search, bidirectional; informed searches: greedy best-first search, A* search, bi-directional A* search, IDA*, RBFS, and weighted searches.