CS4025Z: Artificial Intelligence
Search Algorithms

Zach Wolpe

23 October 2020

Abstract

In this project we explore search algorithms - the basis of many techniques defining artificial intelligence. Here we focus on the branch of search techniques that are computationally tractable & thus do not rely on state value approximations. The applications are vast & pragmatic: vehicle GPS routing; traversing a graph; combinatorial optimization problems like scheduling; sequential decision problems; robotics control algorithms, etc. Finally, we'll explore the benefits of adding heuristics to the problem to intelligently constrain the search space in a computationally tractable manner.

Background

Computationally rational agents

An agent is an entity that perceives & acts. An agent is said to be rational - or sometimes referred to as computationally rational - if it acts to maximise it's (expected) utility. Goals are expressed in terms of the utility of their outcomes.

A reflex agent acts only on what it currently perceives and possibly memory - without consideration of future consequences. A planning agent, however, bases decisions on hypothesized subsequent states - which requires a model (need not be exact) of how the world evolves as actions are taken. An optimal planning algorithm is one that is guaranteed to return the best solution, whilst a complete planning algorithm guarantees any solution - need not be optimal - if one exists.

Search problem formation

A search problem consists of a state space: which defines all the possible states (agent and environment pairings); a successor function: which defines the state transitions; and a start start & goal test. A solution is a sequence of actions (a plan) which transforms the start state to the goal state.

State space representation

The state space is best represented as a graph or tree.

Search algorithms

Search algorithms should be described by whether or not they are optimal & complete, as well as their time & space complexity. Most applications attempt to find an optimal balance between memory & computational efficiency and precision.

Data structures

When implementing these search algorithms, one should carefully consider the data structure being utilized. Many alternative algorithms can be implemented by simply changing the ordering of a priority queue or stack data structure.

Search heuristics

Information can be added to the search problem by utilizing heuristic functions. These functions simply estimate how far the agent is from the goal state. Generally, the better the estimate, the harder it is to compute. A good heuristic is one that balances computation & accuracy. A good heuristic would then allow an agent to traverse a state space more efficiently, effectively incurring some computational cost to more accurately re-weight the potential actions.

Assignment Specification

The assignment was adapted from ***CS188 at UC Berkeley***. Although I have implemented the assignment in full - available on my ***Github*** - this report is only concerned with specific questions.

The assignment is to use various search algorithms to solve problems in the Pacman universe. Pacman is our agent, we help him traverse his way though the maze to satisfy various goals like eating all the dots (food) or finding the terminal state of a maze.

Question $1-to-3$ are concerned with uninformed search techniques, question 4 details the famous $\mathcal{A}*$ algorithm. Question 5 and 6 define a new problem - finding the corners - & design a suitable heuristic to solve the problem more efficiently. Question 7 designs a search heuristic for an agent that wishes to eat all the food on the maze. Finally, question 8 defines a problem in which the exact goal state may not be feasibly computed - as such we implement a sub-optimal candidate solution.

Depth and Breadth First Search

Here we implement Depth First Search (DFS) & Breadth First Search (BFS) - the simplest search algorithms. Both are able to plan to reach the goal state successfully, though notice that DFS was able to reach the goal in 390 node expansions whilst DFS required 620. The heat/trail maps provide the intuition behind this: DFS explores the search tree to it's full depth before exploring the next node in the queue - thus it simulates the full path to the goal. BFS, however, moves horizontally in the search tree - covering local area first - thus taking far longer to explore the region around the goal state in this specific problem.

Figure 2: DFS, left, and BFS, righ, applied to solve the 'bigMaze' problem.

Uniform Cost Search

Suppose we wish to influence Pacman's behaviour to achieve some objective. Search algorithms that focus on accounting for the 'cost of taking an action' can achieve this by favouring states that are deemed lower cost.

For example, we may charge a higher cost for dangerous regions - near ghosts - and less for fruitful regions - near food. The resulting agent would act rationally & favour regions with lots of food. We implement this by stacking the sequences in our search tree as a priority queue, where the cost dictates the priority.

Here the cost is dictated by the placement of the ghosts & the food, resulting in an agent that acts optimally - changing it's behaviour to match the environment.

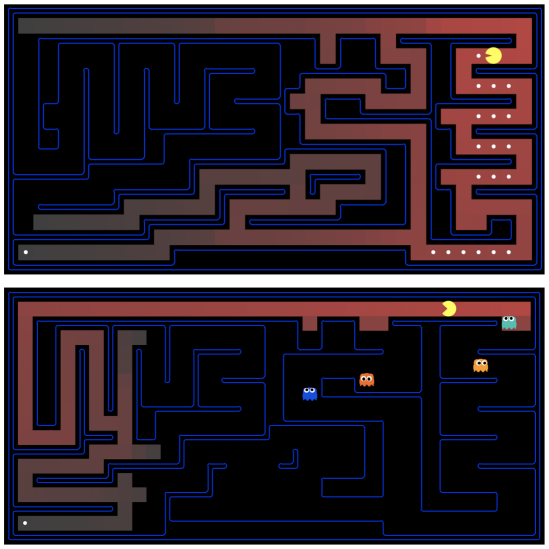


Figure 3: creating rational behaviour by enforcing a cost function.

A^* Algorithm

Whilst simple, the $A*$ search algorithm is incredibly effective and utilized in a vast set of domains. Many game engines rely - to some extent - on A^* and Modern GPS systems use A^* for finding the shortest route.

A^* simply prioritises fringes of the search tree by the equally weighted sum of their cost $g((x, y))$ (cost to reach the node from the start state) & heuristic $h((x, y))$ cost (estimated cost to the goal state). That is:

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

Our A^* implementation is able to solve the big maze in far fewer expansions than the others - a great improvement.

Finding all the Corners

In this section we first design a new problem - that is implemented sufficiently generally such that it could be solved by any of the implemented search algorithms - & thereafter design heuristics to better solve the problem with A^*.

Defining the Problem

The agent needs to find a path that traverses through all 4 corners of any given map. The location of the corners are extracted from the *GameState* object passed to the problem on intialization.

Having the location of the corners, one can simple add a list that tracks the corner states visited by an agent, at a given location on the search tree. Initialized as an empty list, when a particular route includes a corner it is added to the agents state. The agent then reaches the goal state when all corners have been visited - that is when the corners list is of length 4.

Heuristic Implemented

To better inform the problem, we wish to prioritize branches that are more likely to be near the goal state. Our goal state in this instance is defined as any state that has traversed all corners on the map. As such we define a simple heuristic that starts at the current location & iteratively computes the minimum distance to the next unexplored corner. That is:

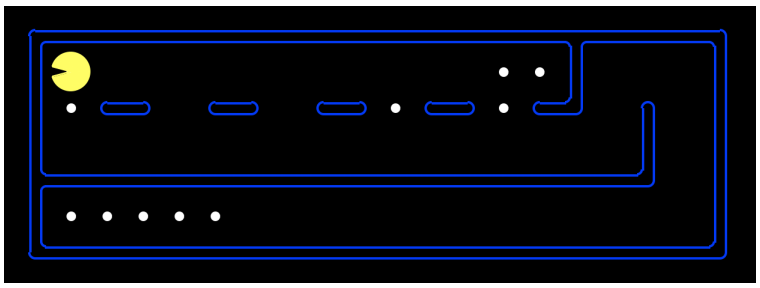
1. Set 'distance' to 0.
2. From the current state compute the distance to the nearest, not yet visited, corner. Add this distance to 'distance'.
3. Set the current state to this corner.
4. Repeat $2 - 3$ until all corners are explored.
5. Return 'distance'.

Manhattan distance is used, as it is known that Pacman can only move in a grid fashion. Any other distance formula can be easily substituted.

Is the Heuristic Non-Trivial, Admissible & Consistent?

It is obviously non-trivial as 0 is only returned at the goal state. The heuristic is admissible as it is clearly the lower bound given Pacman's restriction to moving over Manhattan distances. The heuristic goes directly to the nearest corner and thereafter directly to the nearest corner again until completion. Despite the topology of the grid, this achieves the minimum distance traveled, in the absence of walls. It is a self-evident lowerbound.

Consistency can only guaranteed with a proof. Consistency is more likely when the heuristic is derived by problem relaxation - which is exactly what we have done with in this case (assuming Pacman is free to travel any path). I have not imposed an upper bound on the heuristic value, as this implementation passes all consistency tests and appears consistent as it would yield the minimum cost configuration.



Eating all the Dots

Defining the Problem

In this problem the agent needs to find a path that traverses through all of the food - eats all of the dots.

Heuristic Implemented

The heuristic we implement is very similar to that which we implemented for the 'finding the corners' problem. In some ways these are very similar problems: we have a list locations that need to be visited (either the corners or location of the food), as well as the current location. We thus implement a very similar solution:

1. Set 'distance' to 0.
2. From the current state compute the distance to the nearest, not yet visited, Dot. Add this distance to 'distance'.
3. Set the current state to this corner.
4. Repeat $2 - 3$ until all corners are explored.
5. Scale 'distance' by $\frac{1}{2.5}$.
6. Return 'distance'.

Our heuristic may, however not be consistent. As such a scaling factor of $\frac{1}{2.5}$ - chosen empirically - is multiplied by the final answer, reducing the heuristic values relative to cost.

Is the Heuristic Non-Trivial, Admissible & Consistent?

Again, the heuristic is simply a relaxation of the real problem - assuming the agent can travel in any direction for any duration - thus is admissible & consistent by the same logic as above. The additional scaling factor reduces the likelihood of inconsistency by limiting the influence of the heuristic.

A better bound might be to bound the incorporate the minimum cost of the next move to limit the influence of the heuristic, though I did not implement that here as this solution appears to work well and requires less computation - computing the bound would require another round of evaluating the successor function each time the heuristic is implemented - & the current result appears consistent.

FIN.