AI

Aamod Varma

CS - 3600, Spring 2025

Contents

Chapter 1

Search

We will go over **uninformed** (given no information except problem definition) and **informed** search (given some guidance on where solutions are) algorithms.

1.1 Problem solving agents

We need to formulate the problem first by deciding what actions and states to consider, given a goal. Process of looking for a sequence of actions to reach goal is called **search**.
An **incremental formulation** starts with the empty state and adds each state incrementally. A **complete-state formulation** starts with the set of things that can be on and then tries to optimize them.

1.2 Searching

We start at the root node. We then expand the current state, so the branches of the current node we are at. Then we choose which of these three children to consider. Each child is called a **leaf node** and set of all leaf nodes to be expanded is called a **frontier**.
Some graphs have loopy paths such that one ends up traversing a loop in the graph indefinitely.
The general approach to a graph search algorithm is,

```
func GRAPH-SEARCH(problem) returns solution,failure
    init frontier with initial state
    init explored set as empty
    loop do
        if frontier empty return with failure
        choose leaf node, remove from frontier
        if node is goal return solution
        add node to explored set
        expand node
            if children not in explored add to frontier
```

1.2.1 Measuring performance

- **Completeness**: Is the also guaranteed to find solution if it exists
- **Optimality**: Does it find the optimal solution
- **Time complexity**: How long to find solution
- **Space Complexity**: How much memory is required

b, **branching factor** is the max num of successors of any node and d, **depth** is the depth of the shallowest goal and m, is the **max length** of any path in the state space.

1.3 Uninformed Search

1.3.1 Breadth-first search

Root node is expanded first, all successors are next and their successors. So all nodes expanded at a given depth before we go on to the next level.
Need a **FIFO** (first in first out) queue.
Metrics are,

- Complete: Yes. If the goal is in some finite depth, BFS will eventually reach it.
- Optimal: No. Shallowest goal is not necessarily the most optimal
- Time Complexity: At depth d we will have b^d nodes so we have,

$$b + \cdots + b^d = O(b^d)$$

- Space complexity: We store every expanded in explored. So we have $O(b^{d-1})$ nodes in the explored and $O(b^d)$ in the frontier. Latter dominates so that is the space complexity.

Main problems are memory requirements.

Uniform-cost search

Is optimal with any step-cost function. Instead of expanding the shallowest node, UCS expands the node with the lowest path cost $g(n)$. Stores frontier as a priority queue (heap).
The goal test on a node is when we expand the node and not when we add it to the frontier. Because when we add we may be on a sub optimal path and a better path may exist in the current frontier.

- Complete: Yes if provided cost is non-zero, if not it can get stuck in an infinite loop.
- Optimal: Yes. Optimal in general.
- Time Complexity: Worse than breadth search. If all cost is same it is equivalent to breadth in terms of function however even if it finds the goal node as it expands all of its children its b^{d+1}

1.3.2 Depth-first search

Expands deepest node in frontier. Basically deepest level until nodes have no children. Then goes one up to next deepest node with unexplored successors and so on. Uses a **LIFO** queue or a **STACK**.
Generally implemented using a recursive function.

- Complete: In finite state space if Graph-search is used its complete (as we have explored set to prevent loops). If Tree-search is used its not complete. However in finite state space both fail.
- Optimal: Both versions are non-optimal
- Time Complexity: Depends on state space. We have $O(b^m)$ where m is the max depth. Can be greater than the state space itself (if finite).

Depth-first only advantage is its space complexity which is $O(bm)$ as you don't need to store all children. Can delete if a node has been expanded, can be removed after all its children are explored.
Backtracking search is a variant uses less memory. Only one successor is generated at a time than all successors. Each partially expanded node knows what to expand next.

1.3.3 Depth-limited search

We limit the depth to l. Assume nodes at level l have no children. Is obviously incomplete as if goal exists at level $> l$ it won't be found.

1.3.4 Iterative deepening depth-first search

Iteratively increase the limit - first 0, then 1,2, ... until goal is found. Combines benefit for DFS and BFS searches. Makes it complete. Space complexity is $\mathbf{O(bd)}$. Time complexity is $\mathbf{O(b^d)}$
Is the preferred uninformed search algo.

1.3.5 Bidirectional search

We run two simultaneous searches - from forward from initial state and backward from goal. Motivation is $b^{d/2} + b^{d/2}$ is better than b^d.
Search ends when frontiers of the two searches intersect (The first sol might not be the most optimal). Additional search required.

- Complete: Is complete
- Optimal: Depends on stopping condition.
- Time Complexity: $O(b^{\frac{d}{2}})$

1.3.6 Summary of uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

1.4 Informed (Heuristic) Search

Uses problem-specific knowledge beyond definition of problem - more efficient than uninformed.
General approach - **best-first search**. Node is selected based on an **evaluation function** $f(n)$ (a cost estimate). So node with lowest $f(n)$ is chosen. Choice of f determines the search strategy. $h(n)$ is the heuristic function which is a part of $f(n)$.

$$h(n) = \text{estimated cost of the cheapest path from } n \text{ to a goal.}$$

$h(n)$ is similar to $g(n)$ but $h(n)$ is dependent on the state and not the path. For instance one node in UCS might have different $g(n)$ depending on the path it takes. However it can only have the same $h(n)$.

1.4.1 Greedy best-first

Expands the node closest to the goal. So $f(n) = h(n)$. Eg. for maps our $h(n)$ can just be the straight line distance from node to goal. It is not optimal - rather greedy.
Graph version is complete in finite spaces but not in infinite, tree is not complete (can be stuck in loops). Space and time complexity are $O(b^m)$ where m is max depth. Different heuristics can make it better.

1.4.2 A* search

Our evaluation function is,

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

Where $g(n)$ is path cost from start node to n and $h(n)$ is cost of cheapest path from n to goal.
So we try the node with lowest $g(n) + h(n)$. Works only if $h(n)$ satisfies certain conditions. It is both **Complete and Optimal**

Conditions for optimality

- $h(n)$ needs to be a **admissible heuristic**: One that never overestimates the cost to reach the goal. Here $g(n)$ is the actual cost till current node. So if $h(n)$ overestimates then $f(n)$ overestimates. If this happens then good paths could never be explored (non-optimal).
- $h(n)$ needs to be **consistent** (relevant for graph search only): For every node n and every successor n' of n generated by action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching goal from n'.

$$h(n) \leq c(n, a, n') + h(n')$$

Optimality of A*

The tree search version is optimal if $h(n)$ is admissible and the graph search is optimal if $h(n)$ is consistent.
If $h(n)$ is consistent then the values of $f(n)$ along any path is non decreasing,

$$f(n) \leq f(n')$$

$A*$ is **optimally efficient** for any given consistent heuristic. No other optimal algo is guaranteed to expand fewer nodes than A^* because A^* expands all nodes with $f(n) < C^*$ where C^* is true cost of the optimal solution.
So A^* is **complete, optimal and optimally efficient** among all such algorithms.

Chapter 2

More Searches

We explore purely **local searches** in the state space, evaluating and modifying one or more current states rather than exploring paths from an initial state.

2.1 Local Search

In some problems, path to goal is irrelevant and only the final configuration matters (eg. place 8 queens such that they don't attack each other).
Local search algorithms operate using a single current node and move only to neighbors of that node. Paths are not retained. Two advantages,

- Use little memory
- Can find reasonable solutions in large or infinite (continuous) state spaces.

Local search is useful for solving pure **optimization problems** in which aim is to find the best state according to an **objective function**.
We can consider the **state-space landscape**. It has a location (the state) and elevation (value of objective function or heuristic). If elevation is heuristic want to find the lowest valley - a **global minimum**, if its an objective function we want to find the highest peak - a **global maximum**. A **complete** local search algo always finds a goal, an **optimal** one always finds a global max/min.

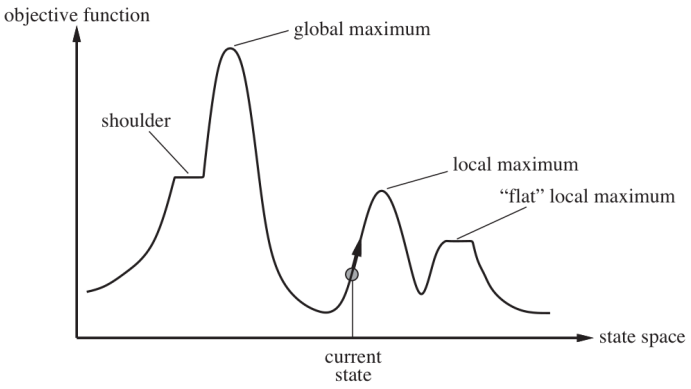

Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

A loop that continually moves in the direction of increasing value - uphill. Terminates when there is no neighbor with higher value. Does not maintain a search tree, data structure only maintains current node and value of objective function. Does not look ahead beyond immediate neighbors.

Example. In the **8-queens problem** we start with all 8 queens on the board. Successor of a state are all possible states by moving a single queen to another square in the same column (each state has $8 \times 7 = 56$ successors). Heuristic h is num of pairs of queens that are attacking each other. Global minimum is when no queen are attacking each other so when $h = 0$ $\diamond$

Sometimes called **greedy local search** as it grabs a good neighbor without thinking anything ahead.

1. Local maxima: A peak that is higher than each of its neighbors but lower than global minimum. Hill-climbing algos will reach here but will be stuck here as all its neighbors are worse than its current position.
2. Ridges: Result in a sequence of local maxima that is difficult to navigate.
3. Plateaux: A flat area of the state-space landscape. No uphill from which progress is possible. Algo might get lost.

In each case algo can't make any more progress. For the 8 queen problem, hill-climbing algos get stuck 86% of the time.
There are other variants such as **stochastic hill climbing** - chooses at random from the uphill moves: probability can vary with the steepness. Converges slowly but sometimes finds better solutions. **First choice hill climbing** uses stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. All these algos are incomplete (as we saw with the 8 queen problem).
Random-restart hill climbing randomly restarts to another initial state space if the default gets stuck. Is able to solve the 8 queens problem.

2.1.1 Simulated annealing

A hill-climbing algo that never makes a downhill move toward states with lower value is guaranteed to be incomplete (as it can get stuck at a local maxima or minima). A pure random walk for instance is complete but inefficient. A solution might be **combining** hill climbing and random walk so its both efficient and complete.
We switch pov from maximizing to minimizing (gradient descent). Analogy is pingpong ball on a bumpy surface. Letting the ball roll, it will stop at a local minima. If stuck we can shake the surface and hopefully bounce the ball out of there. So **simulated annealing** is shaking hard and then gradually reducing the intensity of the shaking.
So instead of picking the best move (like in hill climbing) it picks a **random move**. If the move improves the situation it is accepted, else the algo accepts the move with some probability less than 1 (decreases exponentially with how bad the move is). Probability also decreases as the temperature goes down. (If T goes down slowly, probability of finding global optimum approaches 1).

2.1.2 Local beam search

Initialized with k random states rather than just one in memory. At each step, all successors of all k searches are generated. If any one finds goal, algorithm halts. Else it selects k best successors from the complete list and repeats. **Not** the same as k random restarts in parallel. In local beam search useful information is **passed among the parallel search**. For instance if the kth search is stuck and the $k-1$ search finds two good successors, the kth search with move to one of the two.
Can cause a lack of diversity where they converge to being concentrated in a small region making it a little more than an expensive hill climbing. **Stochastic beam search** is a variant - chooses k successors at random (instead of the best) probability of choosing is a function of its value.

2.1.3 Genetic algorithms

Variant of stochastic beam search in which successor states are generated by combining two parent states than by modifying a single state.
So we start with k random initial states (the population). To produce the next generation of states we have an objective function (in GA terms the **fitness function**). The function returns higher value for better states. For instance for the 8 queen problem we can construct a 8 digit string each number representing the position of the queen in that column. We can mate two strings by using the first half of the first and second half of the second (we decide which strings to mate using their fitness function).
Crossover often takes **long steps** early in and short steps **later on**.

Chapter 3

Adversarial Search

We are in competitive environments where agents goals are in conflict. Most common games are **zero-sum games** of **perfect information**. Which means its **deterministic** in a fully **observable environment**. The utility values at the end are always equal and opposite (eg. in chess one wins and other loses). Games are hard to solve. Chess games can go upto 50 moves by each player, if branching factor is around 35 that means the search tree has 35^{100} or 10^{154} nodes (search graph has 10^{40}). Optimality and efficiency must be balanced for games.

Pruning allows us to ignore portions of the search tree that make no difference to the final choice and **heuristic evaluation functions** allow us to approximate the true utility of a state without doing complete search.

Consider a game with two players, MIN and MAX. MAX moves first and they take turns until the game is over. At the end, points are given to the winner and taken from the loser. For instance tic-tac-toe. Initially MAX has 9 possible moves. Plays alternative between MAX and MIN. Number on each leaf node is the utility value of the terminal state from the pov of MAX; so high value are good for MAX and bad for MIN.

3.1 Optimal decisions

In adversarial search, MIN prevents us from reaching the goal state (pov of MAX). So, MAX needs to find a contingent strategy to figure out a move. So MAX must consider every possible response by MIN to its moves. Given a game

tree, the optimal strategy can be determined from the **minimax value** of each node. The minimax value of a node is the utility of being in the corresponding state assuming that both players play optimally from there to the end.

3.1.1 Minimax algorithm

Computes the minimax decision from the current state. Uses recursive computation of the minimax values of each successor. Time complexity is O^{b^m} as it does depth first search to identify value for all possible paths.