July 1, 2020

State Space search

Using the book Artificial-Intelligence a Modern Approach S Russel and P. Norvig

Completeness / Optimality

    If there is a solution it will be found

    Getting the solution with the most utility

Soundness

    Every solution found is a correct solution

Complexity

    Resources required time complexity e.g., o(n), Space complexity o(log(n))

CH3 – Uninformed search

BFS

Breadth-first search finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost rather than the lowest-depth node. It is easy to see that breadth-first search is just uniform cost search with equal weights.

DFS

Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a nongoal node with no expansion) does the search go back and expand nodes at shallower levels. This strategy can be implemented using a stack.

Taking into account the issues related to complexity and completeness we can consider several variants

Depth limited Search
A BFS with depth limit Depth-limited search imposes a cutoff on the maximum depth of a path. Goal not found? Stop or use a restart strategy. How to choose the depth.

Iterative deepening
Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then expend the depth to 1 then depth 2, and so on.

So far we are only going forward from a start state to a goal/terminal/solution state

Backwards Start from a goal and see if there is a path to the initial/start state

Bidirectional search The idea behind bidirectional search is to search both forward from the initial state j and backward from the goal, and stop when the two searches meet in the middle

Chapter 3 includes an important discussion: The impact of reaching the same state twice (repeating states)

Uniform Cost Search

Each edge has a weight (cost function) the search is a BFS search where node expansion is done in a way the chooses the lowest-cost node on the fringe (as measured by the path cost g(n)), rather than the lowest-depth node. It is easy to see that breadth-first search is just uniform cost search with equal costs **The Dijkstra shortest path algorithm** is a uniform cost search

CH4 – Informed search

In informed search we use available information (if available) about the cost of a path (so far) and the "distance" of current state from the goal.

Cost so far might be more readily available (if the cost function is given then the cost so far is known).

The distance from the goal is generally not know. Have to estimate, estimation is via a heuristic function.

Best first search – using evaluation function that "estimates" the distance from a goal.

The construction of the evaluation function is generally based on Heuristic.

Greedy, Hill climbing, local optimum

Best first search – using evaluation function that "estimates" the distance from a goal.

The construction of the evaluation function is generally based on Heuristic.

Greedy algorithms select the best option between available choices

Greedy search algorithms select the best next step as the step that minimizes the estimated distance of the current state from the goal. The Kruskal **minimum spanning tree** is a greedy algorithm

Hill climbing is a greedy algorithm that uses emulates "hill climbing" by choosing the steepest descent.

Generally, Best First algorithms do not guarantee that we will find a solution or achieve the global optimum. May "get stuck, hopefully in a local optimum.

Local optimum algorithms.

We use a heuristic evaluation function h(n) the estimated distance from vertex n to a (the) solution

Greedy algorithms are not optimal and not complete

The A* algorithm

Takes into account the distance from the solution and the cost so far; two function h(n), c(n)

c(n) is the cost to reach n from start given . We have

Proof of the optimality of A*
Proof of the completeness of A*
Computational Complexity of A* is the least of all the others (best first)
Still a huge space is a challenge.

Examples and avoiding repeated states
End of Chapter 4. – End of search material that can be considered for the final

Chapter 5 talks about searching in the game space.

Break
Some examples of searches and games repeated states

Discussion about the final and the grading policy

Example of a greedy search

Shortest path – Uniform cost manageable complexity optimal
The Minimum Spanning Tree – greedy manageable complexity optimal O(n*log(n))
The Traveling Sales-person problem
NP complete

The knapsack problem - A thief is entering a depot with a sack they want to fill the sack with the most valuable items, many different items with many different sizes and many different values what should they take.
Np complete – there is no known polynomial solution that guarantees global optimality. Only exponential solutions that guarantee global optimality are known.

Algorithms to live by.

CH 3

One simple search strategy is a breadth-first search. In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on. In general, all the nodes at depth d in the search tree are expanded before the nodes at depth d + Breadth-first search can be implemented by calling the GENERAL-SEARCH algorithm with a queuing function that puts the newly generated states at the end of the queue, after all the previously generated states:

Breadth-first search finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost rather than the lowest-depth node. It is easy to see that breadth-first search is just uniform cost search with equal weights.

Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a nongoal node with no expansion) does the search go back and expand nodes at shallower levels. This strategy can be implemented using a stack.

Depth-limited search avoids the pitfalls of depth-first search by imposing a cutoff on the max-imum depth of a path.

Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth then depth 2, and so on.

Bidirectional search The idea behind bidirectional search is to search both forward from the initial state j and backward from the goal, and stop when the two searches meet in the middle

AVOIDING REPEATED STATES – Problem wasting time by expanding states that have already been encountered and expanded before on some other path.

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node's parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node's ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of potentially. It is better to think of this as where s is the number of states in the entire state space.
- DT – Cashing in the state space search domain

CH 4

Best first search – using evaluation function that “estimates” the distance from a goal.

The construction of the evaluation function is generally based on Heuristic.

Greedy, Hill climbing, local optimum

Minimizing the total path cost: A* search

Greedy search minimizes the estimated cost to the goal, *h(n),* and thereby cuts the search cost considerably. Unfortunately, it is neither optimal nor complete. Uniform-cost search, on the other hand, minimizes the cost of the path so far, *g(ri);* it is optimal and complete, but can be very inefficient. It would be nice if we could combine these two strategies to get the advantages of both. Fortunately, we can do exactly that, combining the two evaluation functions simply by summing them.

Proof of the optimality of A*

Proof of the completeness of A*

Computational Complexity of A*
Still a huge space is a challenge

We will consider the general case of a game with two players, whom we will call MAX and M1N,
for reasons that will soon become obvious. MAX moves first, and then they take turns moving
until the game is over. At the end of the game, points are awarded to the winning player (or
sometimes penalties are given to the loser). A game can be formally defined as a kind of search
problem with the following components:
• **The initial state,** which includes the board position and an indication of whose move it is.
• **A set** of **operators,** which define the legal moves that a player can make.
• A **terminal test,** which determines when the game is over. States where the game has
ended are called **terminal states.**
PAYOFF FUNCTION • A **utility function** (also called a **payoff function),** which gives a numeric value for the
outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent
by the values +1, —1, or 0. Some games have a wider variety of possible outcomes; for
example, the payoffs in backgammon range from +192 to —192.
If this were a normal search problem, then all MAX would have to do is search for a sequence
of moves that leads to a terminal state that is a winner (according to the utility function), and
then go ahead and make the first move in the sequence. Unfortunately, MIN has something to say
STRATEGY about it. MAX therefore must find a **strategy** that will lead to a winning terminal state regardless
of what MIN does, where the strategy includes the correct move for MAX for each possible move
by MIN. We will begin by showing how to find the optimal (or rational) strategy, even though
normally we will not have enough time to compute it.
The **mininiax** algorithm is designed to determine the optimal strategy for MAX, and thus
to decide what the best first move is. The algorithm consists of five steps:
• Generate the whole game tree, all the way down to the terminal states.
• Apply the utility function to each terminal state to get its value.
• Use the utility of the terminal states to determine the utility of the nodes one level higher
up in the search tree. Consider the leftmost three leaf nodes in Figure 5.2. In the V node
above it, MIN has the option to move, and the best MIN can do is choose AI i, which leads
to the minimal outcome, 3. Thus, even though the utility function is not immediately
applicable to this V node, we can assign it the utility value 3, under the assumption that
MIN will do the right thing. By similar reasoning, the other two V nodes are assigned the
utility value 2.
• Continue backing up the values from the leaf nodes toward the root, one layer at a time.
• Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the
move that leads to the highest value. In the topmost A node of Figure 5.2, MAX has a choice
of three moves that will lead to states with utility 3, 2, and 2, respectively. Thus, MAX's
MINIMAXDECISION best opening move is *A] *. This is called the **mininiax decision,** because it maximizes the
utility under the assumption that the opponent will play perfectly to minimize it.
PLY
MINIMAX