AI - Lab 4

Shivam Mittal - 2015csb1032

10th April 2017

1 Defining the state

Array for then N*N+3N+1 proposition, containing 0 for false, and 1 if the proposition is true. (Here N is the number of blocks given in the question)

First n*n (0 to n*n-1) propositions for (on blocka blockb).
state[(blocka-1)*N+(blockb-1)] = true (means on blocka block b is true)

Then n (n*n to n*n+n-1) proposition for (ontable block).
state[N*N+block-1] = true (means ontable block is true)

Then n (n*n+n to n*n+2n-1) proposition for (clear block).
state[N*N+N+block-1] = true (means clear block is true)

Then n (n*n+2n to n*n+3n-1) proposition for (hold block).
state[N*N+(2*N)+block-1] = true (means hold block is true)

n*n+3n for empty
state[N*N+(3*N)] = true (means empty is true)

2 Forward planner using BFS search

2.1 Introduction

A deterministic plan is a sequence of actions to achieve a goal from a given starting state. A deterministic planner is a problem solver that can produce a plan. The input to a planner is an initial world description, a specification of the actions available to the agent, and a goal description. The specification of the actions includes their preconditions and their effects.
One of the simplest planning strategies is to treat the planning problem as a path planning problem in the state-space graph. In a state-space graph, nodes are states, and arcs correspond to actions from one state to another. The arcs coming out of a state s correspond to all of the legal actions that can be carried out in that state. That is, for each state s, there is an arc for each action a whose

precondition holds in state s, and where the resulting state does not violate a maintenance goal. A plan is a path from the initial state to a state that satisfies the achievement goal.
A forward planner searches the state-space graph from the initial state looking for a state that satisfies a goal description. The branching factor of the state-space graph depends on the number of actions. The next state after performing an action is calculated by adding the positive effect literals and removing the negative effect literals.
We can use any graph searching technique such as uniformed (DFS/BFS), informed (A*), and local search to search through the graph an find a plan.

Here, we use breadth first search to search through our graph. It starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors. The termination condition is a goal state check, which checks if the current state logically satisfy the goal.

2.2 Statistics

For 1.txt or 2.txt

1. No. of nodes expanded = 124
2. Time taken = 0m0.055s
3. Solution length = 10

For 3.txt or 4.txt

1. No. of nodes expanded = 635
2. Time taken = 0m0.116s
3. Solution length = 14

2.3 Explanation

Here, the pathcost of each action is uniform, and since we traverse all the nodes at one level before going to the next level, the solution we find is optimal. Hence, the length of the solution found here is the shortest one. But, the time complexity is an in BFS since because it is an undirected search and we traverse unwanted nodes. This is the reason, that the problem in the 5.txt or 6.txt file, containing 12 blocks problem could not find a plan in reasonable time (atleast in 10 minutes).

3 Forward planner using A* search

3.1 Introduction

A* search is an informed search algorithm used for path-finding and graph traversal.

It combines the advantages of both Dijkstra's algorithm (in that it can find a shortest path) and Greedy Best-First-Search (in that it can use a heuristic to guide search). It combines the information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Best-First-Search uses (favoring vertices that are closer to the goal).
Let g(n) represent the exact cost of the path from the starting point to any vertex n, and h(n) represent the heuristic estimated cost from vertex n to the goal. Dijkstra's algorithms builds a priority queue of nodes ordered on their g(n) values.
Best-First-Search employs a priority queue of nodes ordered on h(n) values. A* balances the two as it moves from the starting point to the goal. It builds a priority queue of nodes ordered on f(n) = g(n) + h(n) which is the total estimated path cost through the node n.

3.2 The heuristic

The heuristic used in the A* search basically counts the propositions yet to be satisfied and multiplies them by a suitable factor (which are true in the goal state and not true in the current state). It also counts the proposition which needs to be falsified (which are true in the current state and not true in the goal state).
If the proposition is (on blocka blockb) or (ontable block), then the factor multiplied with is taken to be 2. While, if the proposition is (clear block) or (hold block), then the factor is chosen to be 1.
These factor are taken, considering the intuitive fact that to make (on blocka blockb) and (ontable block) to be true generally takes greater number of actions than (clear block) and (hold block).

This heuristic is inadmissible because it is not the case that the cost to reach the goal is always underestimated. Even if we take the factor as one, then also it might be the case that by performing one action, more than one propositions might become true. Hence, this heuristic is inadmissible but the computation time of this heuristic is comparatively low and also the computation time and search nodes expanded becomes significantly less by this heuristic although leading a suboptimal solution.

3.3 Statistics

For 1.txt or 2.txt

1. No. of nodes expanded = 20
2. Time taken = 0m0.033s
3. Solution length = 10

For 3.txt or 4.txt

1. No. of nodes expanded = 48
2. Time taken = 0m0.038s
3. Solution length = 16

For 5.txt or 6.txt

1. No. of nodes expanded = 927
2. Time taken = 0m2.404s
3. Solution length = 28

3.4 Explanation

Here, the solution length we have obtained is not the shortest i.e. the solution is suboptimal because the heuristic we have made in inadmissible. But the advantage o this approach is that the time and the nodes expanded to find the solution is significantly reduced.

4 Goal stack planner

4.1 Introduction

Goal stack planner basically combines the advantages of forward search and backward search. The small branching factor advantage of bacckward search and the soundness of forward search.
The basic strategy is to push all the goals into the stack. Then, to satisfy a proposition, a relevant action is chosen. But, the catch is that the preconditions of the relevant actions are pushed onto the stack above the action. This results in a scenario where any action is added to the action plan only when the preconditions of that action are satisfied, this results in sound states always.
This algorithm may result in an infinite loop because there is no mechanism to backtrack, and hence the relevant actions are chosen smartly based on the proposition to be specified and the current state. The basic idea is to choose that relevant action which results in the goal in the minimum effort (whose most of the preconditions are satisfied).

4.2 Statistics

For 1.txt or 2.txt

1. Time taken = 0m0.032s
2. Solution length = 14

For 3.txt or 4.txt

1. Time taken = 0m0.032s
2. Solution length = 26

For 5.txt or 6.txt

1. Time taken = 0m0.049s
2. Solution length = 36

4.3 Explanation

Here, the solution length we have obtained is not the shortest i.e. the solution is suboptimal. It is even greater than the solution obtained in A* search. This is because the algorithm picks any relevant action that could satisfy the goal, there is no guarantee of optimality.
But, goal stack planning yields the action plan in the smallest amount of time as it only adds the action that can be satisfied. The time shown in the statistics could be reduced further, it is a little more because I added proper bookkeeping and checks to ensure that duplicate work is not being done. Turns out that, the computation time for this bookkeeping is greater than the advantage it provides. But still bookkeeping should be added in the traditional goal stack planning to avoid redundant efforts.

5 Conclusion

From the above observation of the statistics, we can conclude that forward planning using BFS is complete and optimal, but has high time complexity due to reasons explained above.
A* reduces the number of nodes expanded significantly but gives suboptimal solution due to the inadmissible heuristic defined.
Goal stack planning is the fastest of the three algorithms as it adds only the relevant actions whose preconditions are satisfied. Further the time and the solution length depends largely on the order of pushing the predicates and the relevant action choosing strategy.