Programming Assignment 4

Aditya Gupta
2015CSB1003

April 9, 2017

1 Planners for Block World

1.1 Introduction

The goal of this lab was to implement a planning agent for solving Block World Problem wherein given N blocks and actions to "pick a block from table", "unstack a block from another block", "release a holding block" and "stack a holding block onto another block" we were required to reach a goal state configuration from an initial state using three different planners, "forward search planner" with BFS and A*; "goal stack planning". The initial and final state is composed of propositions such as `(on 1 2)` and `(ontable 3)`. Each action is a transformation of these propositions which has some `preconditions`, which when true allows us to perform that action and the `effects` of that actions are unified with the current state and the negative literals removed. States are represented in PDDL.

2 Forward Search using BFS

2.1 Introduction

In this search the states are maintained in a queue (initially containing only the initial state) and then every time a element is taken from the queue and checked for satisfying the goals. If it does not satisfy the goals then all actions which are applicable on the current state are taken (after instantiation) and applied on the current state all resulting states pushed onto the queue, in this way we only see states which require equal number of actions from the starting state and we get the optimal solution (minimum number of steps).

2.2 Observations

File	Time	Solution Length	Optimal Solution Length
`1.txt`	8ms	10	10
`4.txt`	17ms	14	14

- BFS is Complete & Optimal.

3 Forward Search using A*

3.1 Introduction

In this search the planner uses a priority queue instead of a normal queue where the the node with least f value is taken where f is given by:

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

where g is the level or depth of the current state, i.e. the total number of nodes takn until now and h is the heuristic function. The heuristic function enables the search to be more directed towards the goal state and is faster and better than BFS. An optimal solution is returned only if the heuristic is admissible.

3.2 Heuristic Function

3.2.1 H0: An Optimal One: Heights of Blocks

In this heuristic the heights of the blocks in the current state is calculated. This is done via the following way:

- All the blocks on the table are given height 0, this can be done via finding propositions of type `ontable b`.
- For all possible blocks the block on it's top is found out and stored, this can be done via finding propositions of type `on a b`
- Finally the heights of all the blocks are given by starting with the ones on the ground via a BFS.

Now we relax the problem in this way: We can move any block from any height to its goal height by first holding it and then putting it, in most cases we either need to clear blocks from top of this or clear blocks on top of the goal block assuming it is in its correct position. So instead of taking these numerous steps we account for only 2 steps which will infact be the minimum. Also if we are holding a block currently we are halfway done so we only add 1 step. The heuristic value is thus the sum of 2 times blocks not currently at their goal height and 1 if we are holding a block.

3.2.2 H1: An Unoptimal One: Relating to Satisfied Propositions

In this heuristic the total number of satisfied propositions of the goal state are taken into account. Intutively this heuristic seems valid (but not admissible) because any state which is requires less steps to reach goal state would have most propositions of the goal satisfied so if we take each time the state which satisfies most propositions of the goal state we are approaching goal state quite fastly, this may not give the optimal solution as seen in Observations.

3.3 Observations

File	H0		H1		Optimal Solution Length
	Time	Solution Length	Time	Solution Length	
`2.txt`	3ms	10	3ms	10	10
`6.txt`	239.917s	26	20ms	32	26

- H0 is optimal but heuristic calculation takes time.
- H1 is unoptimal but quite fast.

4 Goal Stack Planning

4.1 Introduction

In this search the planner tries to satisfy each goal one at a time assuming them as independently. It maintains a stack of propositions, goals and actions. First it puts all goals it wants to satisfy on the stack and then each proposition of it, then it tries to satisfy each proposition by choosing a relevant action disregarding whether the other propositions that it earlier tried to satisfy become false. After satisfying each of them one at a time it tries to see if each of them are true at the same time, if not it continues to try to satisfy each of its goal once again and so on. We see that it is wise to change the order of the goals to try different scenarios, probably skipping the first and putting it at the last. Also if it satisfies the conjunct goal it then performs the corresponding relevant action earlier chosen.

A relevant action is one which results in the desired proposition being true and does not negate any goal. However we may get many relevant actions and the problem boils down to finding the most relevant action out of these.

4.2 Selection of Relevant Action

4.2.1 S0: First relevant action

In this approach the first relevant action found is taken as the relevant action and then the search is proceeded to satisfy its preconditions and so on. An example is a run for `3.txt`:

```
Initial State: (ontable 5) (clear 2) (on 1 3) (on 2 1) (
    on 3 4) (on 4 5) (empty)
Goal state: (ontable 5) (ontable 4) (clear 3) (clear 4) (
    on 1 5) (on 2 1) (on 3 2) (empty)

Current Goal: (on 1 5)
Relevant Action: stack 1 5
Current Goal: (hold 1)
Relevant Action: unstack 1 2
Current Goal: (empty) [True]
Current Goal: (clear 1)
Relevant Action: release 1
Current Goal: (hold 1)
Relevant Action: unstack 1 2
Current Goal: (empty) [True]
Current Goal: (clear 1)
Relevant Action: release 1
Current Goal: (hold 1)
Relevant Action: unstack 1 2
Current Goal: (empty) [True]
Current Goal: (clear 1)
Relevant Action: release 1
Current Goal: (hold 1)
Relevant Action: unstack 1 2
Current Goal: (empty) [True]
Current Goal: (clear 1)
Relevant Action: release 1
Current Goal: (hold 1)
Relevant Action: unstack 1 2
Current Goal: (empty) [True]
...
```

Firstly it tries to satisfy the goal `(on 1 5)` which can be done only by `stack 1 5`, then it tries to satisfy `(hold 1)` which is required for stacking and the relevant actions for holding are `unstack 1 2` or `pick 1`, apparently it takes `unstack 1 2` and then for unstacking it needs the block 1 to be clear and one of the actions which does it is `release 1` which the heuristic chooses and apparently it needs to hold the block to release it and the goal again becomes `hold 1` thus making the search go into an infinite loop.

4.2.2 S1: Most relevant action

This approach is based on selection of action which satisfies most propositions of current state, i.e. which action requires least number of further actions since it is the one with most preconditions satisfied by the current goal state. Unfortunately this heuristic makes the search get stuck in a loop quite easily. An example is a run for `3.txt`:

```
Initial State: (ontable 5) (clear 2) (on 1 3) (on 2 1) (
    on 3 4) (on 4 5) (empty)
Goal state: (ontable 5) (ontable 4) (clear 3) (clear 4) (
    on 1 5) (on 2 1) (on 3 2) (empty)

Current Goal: (on 1 5)
Relevant Action: stack 1 5
Current Goal: (hold 1)
Relevant Action: unstack 1 3
Current Goal: (empty) [True]
Current Goal: (clear 1)
Relevant Action: unstack 2 1
Current Goal: (empty) [True]
Current Goal: (clear 2) [True]
Current Goal: (on 2 1) [True]
Applying Action: unstack 2 1
State became: (clear 1) (hold 2) (ontable 5) (on 1 3) (on
     3 4) (on 4 5)

Current Goal: (on 1 3) [True]
Current Goal: (on 1 3) [True]
Current Goal: (empty)
Relevant Action: stack 2 1
Current Goal: (hold 2) [True]
Current Goal: (clear 1) [True]
Applying Action: stack 2 1
State became: (empty) (clear 2) (ontable 5) (on 2 1) (on
    1 3) (on 3 4) (on 4 5)

Current Goal: (clear 1)
Relevant Action: unstack 2 1
...
```

Here also the search gets stuck into a loop where to `unstack 1 3` it needs `clear 1` which is done by `unstack 2 1` but then to then actually perform `unstack 1 3` it needs `empty` and therefore it does `stack 2 1` and then reaches the same state and thus into an infinite loop.

4.2.3 S2: Maintaining count of used actions in particular situation and accounting relevancy

This approach is based on making a **general relevant action selection method for goal stack planning**. In this case we maintain count of every action returned by the "relevant action" selector function in a global structure corresponding to every state and proposition pair, i.e. for a given state and when trying to satisfy a given proposition which action was returned. Now our selection is based on returning the action which has least been used. This provides for all actions to be used equally. Also if there is a tie the action whose most preconditions are satisfied by the current state is taken. As a last resort every action has a chance of 1% being selected (may be replaced by another action if it satisfies any of these three conditions and is looked upon after this action) despite the number of times it has been used and the number of preconditions the current state satisfies. This ensures randomness and completeness. The results on large inputs is however too complex to be solved using this method. Also if by any chance the current state becomes the goal state and there are unnecessary goals and actions on the stack we short circuit the method to end there. The results are not pretty and it does not work all the time and exceeded the alloted time of around 30min for `6.txt` where it reaches a state with all the blocks on the table and then just juggling them around.

4.2.4 S3: Manually Hardcoding the Relevant actions in form Goal-Action Conditions

Another approach would be to actually think about the problem and relate which actions would be useful to satisfy which proposition. The following schema was followed:

- `(empty)`: `release x` given `(hold x)`, To get empty the best solution would be to release the currently holding block rather than stacking it here and there, further complicating the problem.
- `(clear x)`: `unstack y x` given `(on y x)`, To get a block cleared the approproate action would be to unstack blocks on top of it.
- `(hold x)`: `unstack x y` given `(on x y)` otherwise if `(ontable x)` then `pick x`, if the block is on table we can directly hold it otherwise if it is on another block we unstack it from there.
- `(ontable x)`: `release x`, putting a block on table is equivalent to releasing it
- `(on x y)`: `stack x y`, only one way to get a block on another block

4.3 Observations

File	S0		S1		S2		S3		Optimal Solution Length
	Time	Solution Length	Time	Solution Length	Time	Solution Length	Time	Solution Length	
`3.txt`	∞	Incomplete	∞	Incomplete	63ms	734	3ms	18	14
`5.txt`	∞	Incomplete	∞	Incomplete	∞	Incomplete	3ms	40	26

- S0 and S1 are useless and almost always get stuck into loop.
- S2 is better but only works on small data set and also gives quite long solutions given the randomness and applying various action.
- S3 is quite good and fast, still unoptimal, Sussman's anomaly, shouldn't treat each goal independently but this is true for Goal Stack Planning in general.