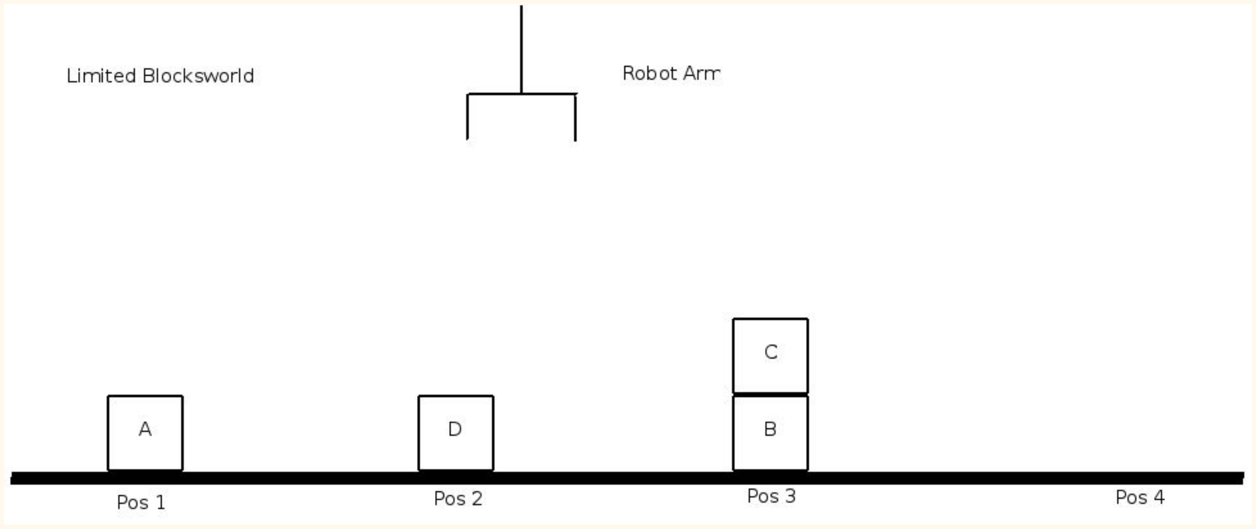
CSL452
MR. CK Narayan

Lab 4

Block World Planner

By

1. Kartavya Ramnani (2013CSB1014)
2. Shinde Lav Chandrakant (2013CSB1032)



INTRODUCTION

In this lab, we implemented the following planning techniques for the Blocks world :

1. Forward (progression) planner using Breadth First Search
2. Forward (progression) planner using A* search
3. Goal Stack Planner

The Problem:

There are N blocks, table and a robotic arm. Blocks are identified by integers 1 to N. Each block can sit on top of another block or on the table. There can be a stack of blocks of arbitrary height. However only one block can be directly on another block. No two blocks can be sitting directly on the same block. The bottom most block of a stack must be on the table. The table can hold any number of blocks. If there is no block on top of a block, then the block is clear. The robotic arm can hold only one block. If the robotic arm does not hold any block, it is empty.

To Run the Code, please type the following command :

```
python main.py filename.txt
```

The Solution :

1. Forward Planner using breadth first search.

Planning is a particular type of problem solving in which actions and goals are declaratively specified in logic and generally concerns performing actions in the real world.

To run the code, “main.py” has to be run with the file’s name to given as input in the terminal. The code will run BFS planner if the mode is “f”. The code where it is implemented is : “bfs.py”.

How does the code tackle the situation ? It is state-space planning. For that, state-space must be populated. The states are handled in the form of propositions which are written exhaustively. The states are then converted into Block List where each element provides information about each block. The states for the BFS are in form of propositions. With the initial state in the queue, BFS then marks it visited and then gets all the applicable actions which can be performed on the state. These actions are then used to generate further states which form the next level of BFS. Therefore, each possible action generates another state which has the Parent state as one of its attributes. This search goes on until the goal state is achieved. The performance has been excellent and the time taken by bfs.py is as follows :

1.txt : 0.0245 seconds

3.txt : 0.4503 seconds

5.txt : >30 minutes

As, one can see, 5.txt has 12 blocks making the search space extremely huge. This calls for a better method, a heuristic which must be used for making the search specific, fast and easy.

2. Forward Planner using A* search.

The implementation is pretty much the same as BFS, just the state to be expanded is decided using a heuristic. Heuristic is : f(n) = h(n) + g(n) , where h(n) is the heuristic applied for estimating the cost and g(n) is the cost from the start node to node n.

g(n) used here is unit cost per action from the start node. It is the heuristic which creates the difference. The heuristic calculates the number of blocks that are currently not in the correct 'position'.It calculates the difference between the current state and the goal state, but looks at the details of each block. If Block A in the goal state is supposed to be on top of Block B and under Block C and in the current state it is neither on top of B or under C, then we add 2 to the heuristic. Similarly, the score is calculated using each block in the state. The state with the minimum heuristic score is expanded.

The results from the 2nd technique has been fantastic !!

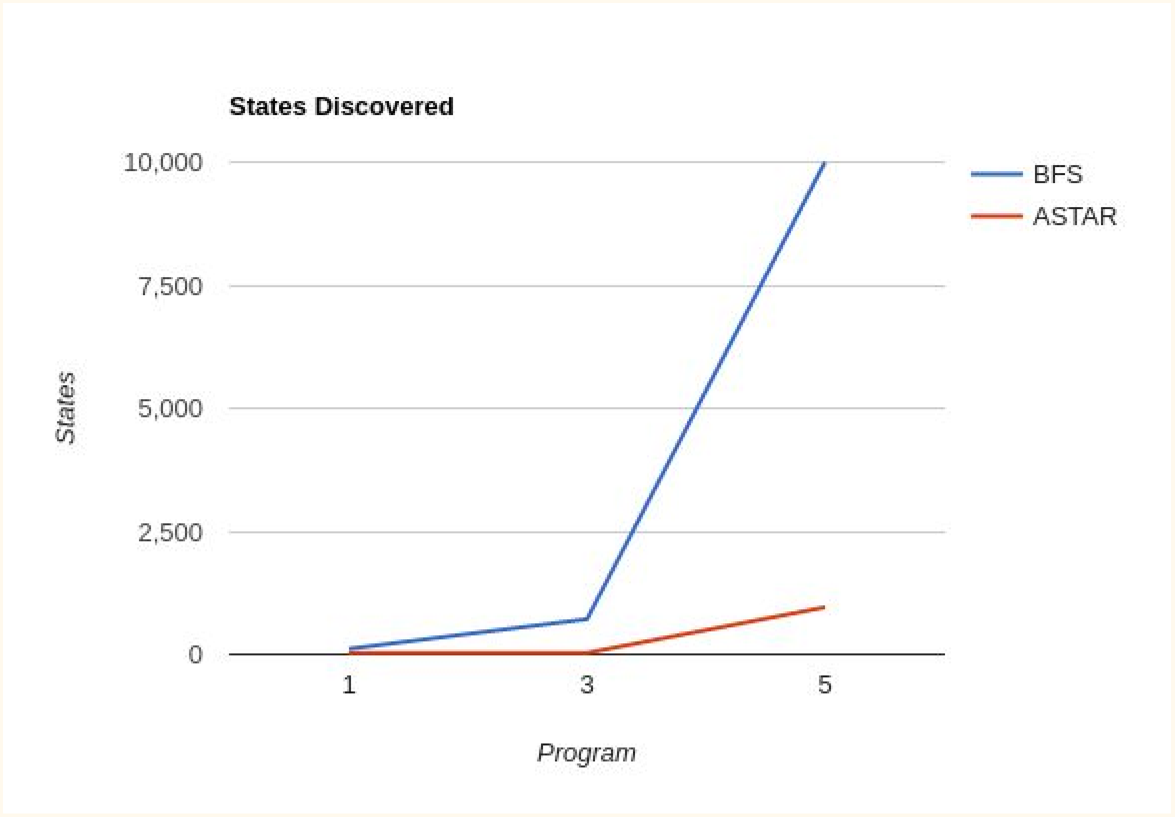
1.txt : 0.0058 seconds

3.txt : 0.0070 seconds

5.txt : 4.0433 seconds

Clearly, there has been an amazing improvement from the BFS technique.

Other Comparison include : Number of states discovered :



File Name	No. Of states disc (BFS)	No. of states disc (A*)
1.txt	114	27
3.txt	716	30
5.txt	Infinity (the code did not run)	957

3. Goal Stack Planning

The basic idea of a goal stack planning is to handle the interactive compound goals by using a stack which contains goals, actions, predicates and a database to maintain the current scenario of the problem after an action has been applied and added to the plan.

We first populate the goal state and then the individual predicates in it which need to be satisfied. Then according to the predicated we try to make them true by applying suitable actions or by simply comparing and checking whether it is already true in the current scenario.

The action is loaded on the stack and then its preconditions are loaded. Once all the preconditions are satisfied then only the action is added to the plan. The current situation initially is stored as the initial state and then changes only when the action is applied.

In this code, the goal stack planning algorithm has been implemented accordingly considering the arm with its hold and release actions/states as well.

The running times and the steps (actions) required for the code are as follows:

File Name	Number of Actions	Time Taken (in seconds)
3.txt	14	0.00077
5.txt	40	0.00243