Report on Blocks World with Triangles

Khan Henderson,[1] Jacob Hilst,[2] Kevin Nelson[3]

St. Olaf College[1,2,3]
hender11@stolaf.edu,[1] hilst1@stolaf.edu,[2] nelson67@stolaf.edu[3]

Abstract

Our group took on the Block world's problem with the addition of triangle blocks to complicate our planning algorithm. We attempted an A* heuristic approach based on the current and goal positions of a given block.

Introduction

Blocks World is a planning domain used in Artificial Intelligence to showcase how a given algorithm performs at planning. Typically, blocks world involves an initial state of several stacks of blocks and a goal state. A mechanical arm can move blocks one at a time, placing a block onto another stack or onto the table, starting a new stack. Our work uses a variation of blocks world. We keep the typical setup but with the addition of triangles, where blocks cannot be placed on top of triangles.

Classical search approaches to this problem have included BFS, A* and hill-climbing. However these approaches did not discuss our 'triangle' addition, thus we treat these algorithms as a baseline for which to project our algorithm in the 'no triangle' case. However, our algorithm must still solve the problem with a valid solution.

Related Works

The blocks world puzzle has been the subject of various works and is seen as a domain that serves particularly well in assisting AI courses. The most basic domains are simply assuming blocks of equal size. The typical objective of the blocks world problem is describing how to move the blocks from an initial state to a goal state in the optimal number of moves.

The typical blocks world problem assumes a series of equal sized blocks are to be moved one at a time either on an infinite capacity table or onto another block. We note that the typical blocks world problem has been interesting as it's been proven by **Chenoweth** to be NP Hard. His proof shows logically that the question 'Given a blocks world problem, **B**, is there a route of **M** moves?' is strongly NP complete, which means that as the input grows the time complexity grows in polynomial time (1991, Chenoweth). Additionally this proof can be altered to show that the addition of 'triangle' blocks does not alter the proof substantially for NP completeness o f the blocks world problem.

Bonet and Geffner Demonstrates and tests a variety of class algorithms used to solve the blocks world problem. Primarily 3 Algorithms are explicitly expressed. STRIPS, Hill Climbing Planner [HSP], and Best First Search Planner [BFS].

HSP and BFS particularly interested our group. HSP selects the best children for exploration from an initial state and its available actions, with plateauing accounted for through restarting the problem when a given heuristic value has not been updated sufficiently. Additionally - to avoid cycling - the states are usually stored in a hash table (2008, Bonet and Geffner). Additionally one may switch to a random state after plateauing sufficiently instead of restarting the problem completely.

HSP is not an optimal planner, and furthermore Bonet and Geffner demonstrated that BFS is superior to HSP. BFS hold and Open and Closed list of nodes similar to A*, where the weight of each node is obtained by adding the accumulated cost of the path thus far with the heuristic estimated cost to the goal, multiplied by a constant W>=1. When W=1, as it is in our algorithm, the algorithm is A* (2008, Bonet and Geffner). Most importantly it is shown that the solution found this way will never guarantee the optimal cost in more than W times. This is useful in the event a more complicated/costly action is used instead of simply W=1. For example, a function that identifies blocks which are 'glued together' might require double the cost to treat each block as 'glued'.

An important finding of (Bonet) is that the plan length for BFS is less than HSP, whereas the time is greater than HSP, roughly over 10 block world instances. This is what informed us to attempt a Best First Search Solution.

Method

To implement the priority queue, that stores the state, heuristic, blockInAir and the traceable actions, we implement a tuple that stores (State, Heuristic(State), BlockInAir, Actions=''). Each neighbor creates a NewState which is inserted through a binary search into the priority queue based on its Heuristic. In addition each valid neighbor is generated through an action, for example 'pickup', which builds on the current State 'currState'. Thus our insertion resembles (NewState, Heur(NewState), BlockInAir=block.Id, currState[3]+': pickup block.Id'). We then delineate at the end of the algorithm from the goal state the Action's string such that we can quickly solve the block's world problem in a demonstration, while still having access to the entire generated problem space.

Heuristic Method

There are several ways to obtain a heuristic for Blocks World. The two options we tested are:

Option 1: For each block, if the block is on an incorrect block, increase the heuristic by 2. This is because that block must be picked up (one step) and placed down (one step).. This is an admissible heuristic, since assuming the minimal cost to move an incorrectly positioned block, it never overestimates.

Option 2: For each stack, starting at the bottom of the stack, iterate through the stack until a block is found that is in the incorrect position. Thus, the minimum number of steps needed is moving all the blocks above. Thus, count how many blocks above to get heuristic for that stack. Sum heuristics for all stacks to obtain heuristics for a given state. This is an admissible heuristic because it adds the minimum possible steps for each stack, thus not overestimating.

Algorithm 1: Our Block's World Algorithm

```
Input: initialState, goalState
Output: The path from initialState to goalState
1:      Let Q = a priority queue
2:      Q.insert(initialState)
3:      while currState = Q.pop() is not solution do
4:          if BlockInAir!=None then
5:              for neighbor in validNeighbors do
6:                  if neighbor not in Q do
7:                      Q.insert((neighbor))
8:          else
9:              for neighbor in validNeighbors do
10:                 if neighbor not in Q do
11:                     Q.insert((neighbor))
12:         end if
13:     end while
14:     return solution
```

Obtaining all Valid Neighbors

To get all valid neighbors of a given state, we consider the 5 fundamental operation of Blocks World's arm. They are:

- **Pickup**: picking up a block that is on the table
- **Putdown**: putting down a block onto the table, creating a new stack
- **Stack**: putting a block on top of a stack
- **Unstack**: picking up a block that is on a stack
- **Move**: moving the arm horizontally from one position over the table to another

The possible operations that can be performed on a state varies based on whether the arm currently holds a block or not, explaining the conditional in line 4 of Algorithm 1. Furthermore, these operations all check to see if the blocks instructed to operate on are valid for the given operation.

Each Step in More Depth

Step 1: Find state X. A state in the unvisited states queue which has the lowest H (heuristic) value

1. Sort unvisited states queue by each states H value
2. Pop the first element of the queue

First the algorithm sortes the states in the unvisited states queue by their heuristic value. This is done by using the python3 sort function then sorting the tuples which hold both the state with its heuristic value by the heuristic value.

Step 2: Discover all States B, C D .. etc which are one move away from state X.

1. Find all blocks A, B, C … etc where block.isclear = True. AKA find every block which has no blocks above it.
2. Find every way to stack the clear blocks on top of each other and onto the table.

After step one

Step 3: Filter out visited states

To make sure that states are aren't explored twice, we filter out states that are in the visited states queue.

Step 4: Calculate the H values of each newly discovered unvisited state

1. Turn every state and the goal state into 2D matrix
2. For every block in the state which is at another index in the goal state. Add one to the heuristic value.

In step 4 the algorithm turns the state into a 2D matrix and the goal state into a 2D matrix. For every block in the state

2D matrix which is not in the same index as the same block in the goal state add plus one the heuristic value

Step 5: Add all newly discovered unvisited states to the unvisited states queue.

In order that the discovered unvisited states are explored, they need to be added to the unvisited states queue. If this doesn't happen then we can't go back to step 1 and explore the unvisited states after step 6.

Step 6: Go back to step 1 unless one of the states has an heuristic value of 0.

If there is a state which has a heuristic value of 0 it means that the algorithm has reached the goal state. Otherwise go back to step1, explore a unvisited state and try to find a path from there.

Results

The algorithm takes a long time to get to the goal state if the goal state is a single column where all blocks are stacked on top of each other. In a goal state where blocks A to F were stacked on top of eachother, the algorithm had to explore 36514 unique states to find the answer. In contrast, when the blocks were spread out amongst 2, 3 of 4 columns the algorithm only had to explore 1,000 to 12,000 unique states to find the most optimal path. The algorithm also takes up an enormous amount of memory. To get the path, each explored state keeps track of the path the algorithm took to get to that state. So if 30,000 states and each state on average took 5 moves to get to that state the algorithm would be holding 150,000 states memory. The longer the path is the more memory intensive the algorithm will be.

Conclusion

Our block world algorithm takes much too long to reach the goal state in ideal situations. In the future we would focus on a more memory efficient queue and improve our heuristic by adding values to our Heuristic matrix based on whether the block is clear or not, whether the block should be clear or not, and whether the required blocks are all present at the start of the algorithm, or if we have to many blocks.

References

Bonet, B., Geffner, H., 2008. *Planning as Heuristic Search*. Depto. De Computación, Universidad Simón Bolívar. [https://repositori.upf.edu/bitstream/handle/10230/36325/Geffner_ai_plan.pdf?sequence=1&isAllowed=y]

Chenoweth, S., 1991. *On the NP-Hardness of Blocks World*. AAAI-91 Proceedings. [https://www.aaai.org/Papers/AAAI/1991/AAAI91-097.pdf]