Artificial Intelligence: Planning with Block Arrangements

Lars Askegaard, Bennet Botz, Kaz Matsuo

Instructed by Professor Sravya Kondakunta, St. Olaf College

Abstract

Path planning is a category of Artificial Intelligence that strives to determine the optimal path or method steps to progress from an initial state to a final state. In this problem, we consider a 2-dimensional environment of rectangular and triangular blocks

Introduction: Path Planning

Planning is a long-standing field within Artificial Intelligence which seeks to mimic the act of human planning. According to Russell and Norvig, classical planning is the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment" (Russell and Norvig, pg. 657). A plan, produced by the output of a planning function, will calculate an optimal course of action that enables the computer agent to progress from an initial starting state to a predetermined goal state. A successful path planning algorithm will seek to minimize the number of actions or steps an agent must take in order to achieve the goal state.

The planning method offers a useful approach for situations in which the agent has a limited amount of data and must provide a solution to a complex but constant environment. Current advancements in planning algorithm applications aim to solve problems such as course trajectories of NASA's Mars Rover, management of retail warehouse facilities, and public transportation route optimization. As a pivotal sector of Artificial Intelligence, Planning algorithms have enormous potential to enhance path problem solving across numerous applications and revolutionize the way in which humans determine the correct course of action when facing complex real-world scenarios.

Problem Definition: Block World

The Block World problem is a famous Path Planning problem in Artificial Intelligence whose aim is to return the optimal actions an agent should take in order to rearrange a selection of block objects from an initial state to a final state. In the Block World, we assume that the agent has a limited capacity to handle one block at a time through actions pick up, put down, stack, unstack, and move. Since the agent is constrained by its ability to handle one block at a time, we desire to provide an agent with an optimal guide that minimizes the number of steps an agent must execute.

Our team considers a 2-dimensional environment composed of rectangular and triangular blocks arranged on a table. Given an input block configuration and an output block configuration, our algorithm must aim to minimize the number of steps the agent must take to rearrange the environment from an initial to a goal state. In our planning optimization problem, we are constrained by the requirement that rectangular blocks cannot be placed on triangular blocks, but triangular blocks can rest on rectangular blocks.

To construct this block environment planning algorithm, we must define a few pivotal functions. First, we create five action functions that perform

the operations pick up, put down, stack, unstack and move. Second, we define a neighbors function to determine allowable block configuration neighbors, as well as create lists of potential block operations - or environment "states" - the agent can take for each action the agent can perform. Next, we create a heuristic to rank the list of neighbors in an order that aims to achieve the goal state in the shortest number of actions. Finally, we implement a Greedy Best First Search algorithm to perform the exploration through the list of possible paths. In the following sections, we define our fabrication of each of these functions, and discuss the results of the algorithm for different test scenarios.

Neighbors Function

In the code we would need a neighbors function that would return every neighboring state from the current state. The possible moves would then get scored by the heuristic function based on how it compares to the goal state.

We have two main scenarios: If a block is in the air, and if it is not. If a block is in the air, the moves that will get put in the list of possible moves would be only putdown, stack, and move whereas if there are no blocks in the air, the possible moves would be pickup and unstack. In each case we must create a deep copy from the current state as we do not actually want to change the current state but a hypothetical state to pass through to the heuristic.

When a block is in the air, the putdown function only has the table to put the block down on so this would only return one neighboring state. However, for the stack function, we would need to first find all clear blocks then create hypothetical states for each possible combination of putting the block down onto another block. For this we utilize the intertools function which essentially performs a matrix multiplication on two lists to provide every combination between the elements. We find the product between the blocks in the air with every open block and then create a deep copy to put into our possible moves list.

When a block is not in the air, we have the option to pick up a block from the table or unstack a block from another block. Either way, we could traverse a list of blocks that are either on table and clear for pickup, and not on table and clear for unstack. Finally, we concatenate all the lists which contain the move set for the current state and return this to be used in a heuristic.

Heuristic Approach

Our initial approach was to use the same heuristic we would use in a maze problem with an A-star algorithm. We would calculate the Manhattan distance from the goal state for every neighboring state and combine that with a path cost from the initial state to make up our heuristic. However, in the block world, calculating the direct Manhattan distance from the goal state would be difficult to do, as we would have to keep track of and score every state based on how many moves away that state would be from the goal state. Based on time complexity and optimization we concluded that this would not necessarily be the most effective heuristic.

The next approach we took in our heuristic was to make one that was appropriate to the context of the block world. We understood that within every state, what defines the relationships between the blocks would be the properties .on() and .clear(). Each block had this property and in order to get our initial state to the goal state these properties would have to match that of the goal state. Therefore, in our heuristic, we decided to give each block a score based on if they match the. on() and .clear() of the goal state. The total score of each state became the sum of the scores of all the blocks in that state. We would be able to calculate a score for every state based on its relationship with the goal state.

This heuristic approach was useful in a way that we could use this as an application towards a stop condition of the planning. By scoring every

state, we would know if the state has reached the goal state by comparing the scores of the two states. The goal state would have a full score of 2*# of blocks and the current state would have a score based on its relationship with the goal state blocks'. on() and .clear() condition.

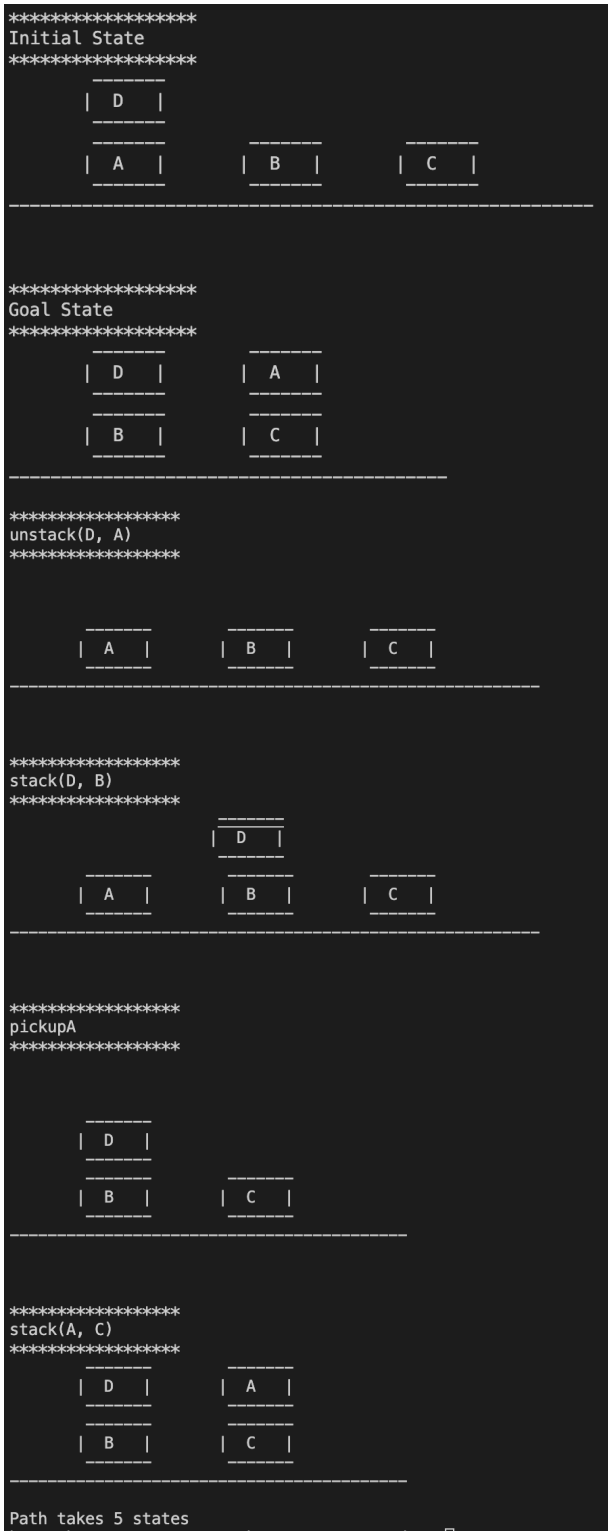
Greedy Best First Search

The Greedy Best First Search algorithm explores an environment by visiting the nodes that are closest to the goal. Each node is evaluated by a heuristic function and given a numerical value. When evaluating multiple nodes, the Greedy algorithm will always choose the node closest to the goal state, making the algorithm 'greedy.'

Greedy Best First Search is effective when evaluating the block's world. Given the initial state, Greedy Best First Search evaluates all the possible neighboring "nodes" or in the case of the block's world, states. The Greedy algorithm uses a heuristic to assign a score to each of these states, and then stores the state along with the score into a priority queue. Using a priority queue, the state closest to the goal is stored in the front of the queue. The state in the front of the queue is then popped from the queue and is checked to see if it is the goal state. If it is the goal state, we return that state from the Greedy Best First Search algorithm. If it is not the goal state, we add that state to a list of visited states. This process is then repeated until we find the goal state.

Results and Conclusion

The block's world problem was solved using the Greedy Best First Search algorithm along with the heuristic function, and neighbors function. The heuristic function is used to evaluate how close a current state is to a goal state based on our original algorithm. The neighbors function is used to evaluate all the possible moves based on a current state. The Greedy Best First Search algorithm was used to traverse through the block world states until we find the goal. The goal was found using the greedy approach; Therefore, we are able to confirm the usage and efficiency of the Greedy Best First Search algorithm in the blocks world problem. Below is a visual of an example initial state and goal state representation of the block space, along with the necessary ordered moves to achieve this goal state:



A potential drawback of using GBFS would be the randomness associated with choosing the highest score. In many cases, our states have the same score as other states in the frontier. In this case there is a chance that the frontier will choose a state that will not lead to the goal state. This could result in not finding the optimal path but taking a longer path to get to the desired goal state.

References

Russel, Stuart. Peter, Norvig. 2021. *Artificial Intelligence A Modern Approach, Fourth Edition.* Pearson; 4th edition.

Acknowledgements

Thank you to professor Sravya Kondrakunta for supplying us this version of the Block World Problem, and for extending elements of pseudocode to aid with the coding process.