The Block World Problem

Maheen Asim, Julie Zaepfel

St.Olaf College

Abstract

In this paper, we explain about the algorithm we developed that could autonomously transform an initial configuration of blocks into a specified target arrangement while showing all the steps it takes. Drawing upon our understanding of predicates, and Goal Stack Planning algorithm we constructed modular classes that could be seamlessly employed to manipulate and reposition the blocks for achieving the desired goal state.

Introduction

The goal of this paper was to consider a state space in the Blocks World that includes a given number of blocks, and to develop an algorithm that would accept any combination of those as the initial state, and another combination of those blocks as the goal state and show the intermediary steps that the blocks have to go through to reach from the initial state to the final state. The algorithm we created would arrange two or more numbers of blocks from initial state to goal state using Goal Stacking and the concept of predicates. By using a stack approach it breaks down complex goals into simpler subgoals and determines the actions needed to satisfy the aforementioned subgoals. The calls at the end of the program set the whole thing into motion.

Related Work

One article resource we found was an article explaining goal stack problems by Joel Anandraj. Anandraj provided a good deal of insight on Goal Planning, which involves strategizing a sequence of actions to reach a solution, particularly valuable for intricate, interdependent problems. In complex, non-decomposable problems with interconnected subgoals, Goal Stack Planning (GSP) emerges as a straightforward yet effective algorithm. It allows us to manage and define the problem's world by utilizing a stack structure, GSP incorporates sub-goals and predicates-based actions, allowing for flexible, sequential resolution of subgoals in any order (Anandraj 2018). For our program, we used a similar structure particularly the planning involving Goal Stack Planning (GSP) that deals with implementing different operations on blocks and checking if their precondition is satisfied. However, in contrast our algorithm separates the steps involved in planning into distinct methods, each with a specific responsibility. This separation of concerns improves code modularity and allows for easier testing and debugging. Moreover, the algorithm provided by Anandraj uses a single stack for storing both goals and actions, which can lead to difficulties in managing the global state and handling multiple subgoals concurrently whereas the algorithm that we created maintains a separate knowledge base (world_state) and enforces a clear distinction between goals and actions, improving the handling of state information. Maintaining separate stacks and knowledge base allows for a more organized and efficient handling of goals and subgoals. This makes it easier to backtrack and explore different goal paths when the planner encounters obstacles.

Another article we looked into was a different AAAI article about the Blocks World problem by John Slaney and Sylvie Thiébaux. In their article, Slaney and Thiébaux explore the optimization of solutions to the Blocks World planning problem. (Slaney, Thiébaux 1996). While some of this article's insights on optimization were not strictly related to our work on it, it was interesting to relate their ideas to our own solution. One of the key elements of this article focuses on planning out of the steps that the world state would go through to get to the goal state. If the plan was optimized, then it follows that the actual run of that plan would be optimal as well. Slaney and Thiébaux go on to discuss how the complexity of the algorithms used affect the time it takes the algorithm to find the solution. The discussion of efficiency was indeed fascinating - even though planning can seem to be an optimal solution, its complexity could be not the preferred one but ultimately the authors find the linear time complexity. Since our project was not based on optimal choice yet we hope to extend our solution on more optimal approaches.

Approach and Results

Our program consists of four files in the folder each with different classes and a main file. Broadly, our files can be outlined have twelve classes, and those would include: a REQUIRED_PREDICATES class and an Allowed_Operation class file, an ON class, an ONTABLE class, a CLEAR class, an AIR class, an ARMEMPTY class, a STACK class, an UnSTACK class, a PICK_UP class, a PutdownOp class, and a GoalStackPlanner class. The initial state and goal states of blocks are given as lists in the main.

The REQUIRED_PREDICATES class defines the methods and structure that the required predicates should have. These include:

1. ON(A,B) : Block A is on B
2. ONTABLE(A) : A is on table
3. CLEAR(A) : Nothing is on top of A
4. HOLDING(A)/AIR : Arm is holding A.
5. ARMEMPTY : Robot Arm is holding nothing

The Allowed_Operation class defines the common structures for the operations which are:

1. STACK(A,B) : Stacking Block A on Block B
2. UnSTACK(A,B) : Picking up Block A which is on top of Block B.
3. PICKUP(A) : Picking up Block A which is on top of the table or another block
4. PutdownOP(A) : Put Block A down.

All these have preconditions from predicates that need to be satisfied and these are given in table no.1. The preconditions for each operation is checked before it is performed and then Delete and add columns show what is followed after the operation about the state of the Block and the robot's arm. For instance, to perform PICKUP(A) that is picking up Block A which is on top of the table or another block, the precondition is that no other block should be on top of block A (CLEAR(A)) and the arm of robot must be empty (ARMEMPTY()). After the Operation has been performed, then we delete the status of (ARMEMPTY()) and add the status (HOLDING(A)).

Table no. 1: Predicates conditions for the Operations

Operators	Preconditions	Delete	add
STACK(A,B)	CLEAR(B) ∧HOLDING(A)	CLEAR(B) HOLDING(A)	ARMEMPTY ON(A,B)
UNSTACK(A,B)	ARMEMPTY() ∧ON(A,B) ∧ CLEAR(A)	ARMEMPTY ∧ ON(A,B)	HOLDING(A) ∧CLEAR(Y)
PICKUP(A)	CLEAR(A) ∧ ARMEMPTY()	ARMEMPTY()	HOLDING(A)
PutdownOP(A)	HOLDING(A)	HOLDING(A)	ONTABLE(A) ∧ ARMEMPTY()

After these preconditions are met and operations are ready to be used we implement the GoalStackPlanner class. This class takes an initial state, goal state and initializes an empty stack to manage goals and actions. The GoalStackPlanner class is a central planner for this program; it is designed to facilitate the achievement of a specific goal within the context of a problem-solving domain here. Given an initial state and a desired goal state, it systematically generates a sequence of steps or actions necessary to transition from the initial state to the goal state. Employing a goal stack approach, it effectively breaks down complex goals into more manageable subgoals and orchestrates the execution of actions required to fulfill these subgoals. The class maintains a program stack to manage its goals and actions, manipulates a knowledge base representing the world state, and continually evaluates and resolves dependencies among goals and actions to reach the desired outcome. It dynamically assesses the satisfaction of preconditions, ensuring all necessary conditions are met before executing actions. If preconditions are unsatisfied, the class adds them to the stack to be recomputed, and if preconditions are met, it pops the action from the stack, appends it to a list of steps, and updates the world state accordingly. Ultimately, the GoalStackPlanner serves as an integral tool for solving problems in dynamic environments by orchestrating a goal-driven, iterative planning process. The pseudocode for the GoalStackplanner class is given in table no.2 below.

Table no. 2: Pseudocode for the GoalStackplanner class

```
class GoalStackPlanner:
  Initialize:
    initial_state = Provided initial state
    goal_state = Provided goal state

  Method get_steps:
    Initialize an empty list called steps
    Initialize an empty stack called stack
    Initialize a knowledge base called world_state and set it to a copy of the
initial state
    Push a copy of the goal_state onto the stack

    While the stack is not empty:
      stack_top = Get the top element of the stack
      If stack_top is a compound goal:
        compound_goal = Pop stack_top from the stack
        For each goal in compound_goal:
          If goal is not in world_state:
            Push goal onto the stack

      If stack_top is an action (allowed operation):
        allowed_operation = Get the top element of the stack
        all_preconditions_satisfied = True

        For each REQUIRED_PREDICATES in allowed_operation.delete():
          If REQUIRED_PREDICATES is not in world_state:
            all_preconditions_satisfied = False
            Push REQUIRED_PREDICATES onto the stack

        If all preconditions are satisfied:
          Pop allowed_operation from the stack
          Append allowed_operation to the steps list

          For each REQUIRED_PREDICATES in allowed_operation.delete():
            Remove REQUIRED_PREDICATES from world_state
          For each REQUIRED_PREDICATES in allowed_operation.add():
            Append REQUIRED_PREDICATES to world_state

      If stack_top is a single satisfied goal:
```

```
    Pop stack_top from the stack

  If stack_top is a single unsatisfied goal:
    unsatisfied_goal = Pop stack_top from the stack
    action = unsatisfied_goal.get_action(world_state)
    Push action onto the stack
    For each REQUIRED_PREDICATES in action.precondition():
      If REQUIRED_PREDICATES is not in world_state:
        Push REQUIRED_PREDICATES onto the stack
  Return the steps list
```

Lastly, the steps that are provided by the GoalStackplanner class are printed using the display_state() in the main when initial and goal states are taken as lists. A run of the program is shown below in Fig no.1 and Fig no.2:

Fig. 01: Run trial of Code

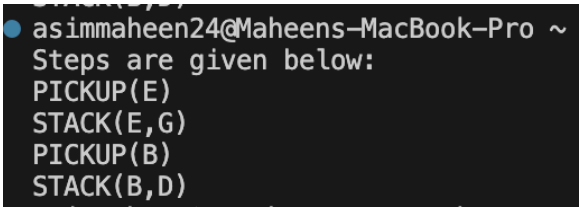

Fig. 02: Result of Run trial of Code

Conclusion and Future Work

Our program was successful in its goal: converting one given world state containing five or more blocks into another with a good runtime. We ended up using many classes to achieve this, drawing from our research, particularly Anandraj's article.

In the future this work could be extended into several areas; the first being the creation of visualizations for the steps given by the program so that it is more convenient to understand. Another possible extension would be making the solution more optimal in terms of time complexity. A third possibility for the future of this work would be the inclusion of other shapes of blocks as possibilities, and reckoning with what that means for our solution.

References

Anandraj, J. AI - The Future. (2018). *Goal Stack Planning*. Available at: https://aithefuture.wordpress.com/2018/02/26/goal-stack-planning . Accessed: 28 October 2023.

Slaney, J. (1996). Linear Time Near-Optimal Planning in the Blocks World. Available at: https://cdn.aaai.org/AAAI/1996/AAAI96-179.pdf. Accessed: 30 October 2023.