Planning & Decision-making in Robotics

Manuj Trehan
Andrew ID: mtrehan

16-782

Homework 3

November 30, 2022

Late Days

Late days available before this assignment: 3

Late days used in this assignment: 3

Compiling Code

Command to compile

g++ planner.cpp -o planner.out

Results

Blocks.txt

1. Without heuristic:
   - Number of states expanded: **16**
   - Average time required to plan (over 10 runs): **5 ms**

   Plan:

   MoveToTable(A,B)

   Move(C,Table,A)

   Move(B,Table,C)

2. With heuristic:
   - Number of states expanded: **6**
   - Average time required to plan (over 10 runs): **3.7 ms**

   Plan:

   MoveToTable(A,B)

   Move(C,Table,A)

   Move(B,Table,C)

BlocksTriangle.txt

1. Without heuristic:
    - Number of states expanded: **422**
    - Average time required to plan (over 10 runs): **471.5 ms**

    Plan:

    MoveToTable(T0,B0)

    MoveToTable(T1,B3)

    MoveToTable(B0,B1)

    Move(B1,B4,B3)

    Move(B0,Table,B1)

    Move(T1,Table,B0)

2. With heuristic:
    - Number of states expanded: **73**
    - Average time required to plan (over 10 runs): **82.3 ms**

    Plan:

    MoveToTable(T0,B0)

    MoveToTable(T1,B3)

    MoveToTable(B0,B1)

    Move(B1,B4,B3)

    Move(B0,Table,B1)

    Move(T1,Table,B0)

FireExtinguisher.txt

1. Without heuristic:
    - Number of states expanded: **363**
    - Average time required to plan (over 10 runs): **100.1 ms**

    Plan:

    MoveToLoc(A,B)

    LandOnRob(B)

    MoveTogether(B,W)

    FillWater(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourOnce(F)

    LandOnRob(F)

    MoveTogether(F,W)

    FillWater(Q)

    Charge(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourTwice(F)

    LandOnRob(F)

    MoveTogether(F,W)

    FillWater(Q)

    Charge(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourThrice(F)

2. With heuristic:
    - Number of states expanded: **336**
    - Average time required to plan (over 10 runs): **94.2 ms**

    Plan:

    MoveToLoc(A,B)

    LandOnRob(B)

    MoveTogether(B,W)

    FillWater(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourOnce(F)

    LandOnRob(F)

    MoveTogether(F,W)

    FillWater(Q)

    Charge(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourTwice(F)

    LandOnRob(F)

    MoveTogether(F,W)

    FillWater(Q)

    Charge(Q)

    MoveTogether(W,F)

    TakeOffFromRob(F)

    PourThrice(F)

Implementation Details

The heuristic I used is just a simple heuristic which compares how many conditions of the goal state have not been satisfied yet in the current state. As observed from the results above, adding the heuristic lead to an improvement for all the environments, with less states being expanded, and the planner taking less time to plan. In some cases, like *BlocksTriangle.txt*, the improvement was significant.

For *Blocks.txt*, it expanded 62.5% fewer states, and was 26% faster

For *BlocksTriangle.txt*, it expanded 82.7% fewer states, and was 82.5% faster

For *FireExtinguisher.txt*, it expanded 7.5% fewer states, and was 6% faster

My approach for solving this problem involves first going through all the Actions in the given Environment, and creating a map storing all possible Grounded Actions, based on the specific preconditions as the map key, and effects as value. This map to store Grounded Actions, called the *action_map*, is a multi-dimensional map. For each Action in the Env, I first get the number of arguments required. I then create all possible permutations of that number (*nPx*, where n is the set of all symbols, and x is the number of arguments required in the action), from the given set of symbols. I then create preconditions, effects and Grounded Actions for every permutation, with the corresponding symbol set, and store it in the action_map. I created a struct for the action_map, called *MultiDimMap*, with the following structure,

struct ***MultiDimMap***

$unordered\_map < GroundedCondition, shared\_ptr < MultiDimMap >,$
$GroundedConditionHasher, GroundedConditionComparator >$ **multimap**;

$vector < pair < unordered\_set < GroundedCondition, GroundedConditionHasher,$
$GroundedConditionComparator >, GroundedAction >>$ **value**;

bool ***end;***

The multimap inside *MultiDimMap* is an unordered map with a Grounded Condition as a key, and a pointer to another MultiDimMap as a value. Once a Grounded Action is created for a particular combination of symbols, along with its preconditions, I first sort the preconditions in alphabetical order by storing them in a *set* instead of an *unordered_set.* I then iterate over this set, and for each precondition, I progressively add it to the next layer in the map. Once the last condition of the precondition is reached, the map corresponding to that stores a pair of the resultant effect and the name of the Grounded Action. An example can be seen in the next page.

For example, for the Grounded Action *MoveToTable(A,B)*, the preconditions are *Block(A),Block(B),Clear(A),On(A,B)* in alphabetical order

In the action map, this will be stored as follows,

Add to level 1 map:

Key: Block(A)

Value: Initialize empty MultiDimMap if it is a new key, otherwise keep old value. This value is the level 2 map

Add to level 2 map:

Key: Block(B)

Value: Initialize empty MultiDimMap if it is a new key, otherwise keep old value. This value is the level 3 map

Add to level 3 map:

Key: Clear(A)

Value: Initialize empty MultiDimMap if it is a new key, otherwise keep old value. This value is the level 4 map

Add to level 4 map:

Key: On(A,B)

Value: Initialize empty MultiDimMap if it is a new key, otherwise keep old value. This value is the level 5 map

Add to level 5 map:

Now, since the preconditions are complete, this level stores a **value**, which is a vector of pairs of (effect, grounded action) - refer to structure of MultiDimMap. It also stores a bool *end*, to mark whether this is a leaf node or not. A vector of pairs was used, and not just a single pair, in case multiple preconditions ended at the same node.

After the action_map is created, I start the A* search based on the initial state. Each state is defined by a *Node* struct, which stores the conditions of the state, g, h and f values, a pointer to the parent *Node*, and the Grounded Action which resulted in the change from the parent to the child. For every state, I search through the action_map to see which preconditions are satisfied based on the state conditions, since the preconditions could be a subset of the current state. The set of all Grounded Actions, whose preconditions are satisfied, are considered the children of the current node. I create new child nodes (if they don't already exist) after adding/subtracting the effects for each Grounded Action. Following this, the rest of the A* steps take place, including cost, f-val, g-val and parent update and then added to the open queue. Each state popped from the open queue is compared with the goal. If it contains all the goal conditions, it is considered as a goal state, and a path is returned.

Creating a multi-dimensional map for the Grounded Actions and preconditions helped save a lot of planning time, since I do not need to search the entire set of Grounded Actions and preconditions. I can match the keys of the map directly with the conditions present in my current state.