COL864: Homework 2
Symbolic planner for mobile manipulator

Deepak Raina
2019MEZ8497

April 5, 2020

1 Symbolic planning

The goal of this exercise is to write a symbolic planner enabling the agent (mobile manipulator) to perform a variety of tasks in the environment requiring navigation and interaction actions. This will endow the robot with a task planner enabling it to perform semantic tasks in the environment. The more details regarding the same is available at [1].

2 Simulation environment

The simulation environment is based on PyBullet physics simulator with a mobile manipulator (a robot called Husky with a manipulator arm, as shown in

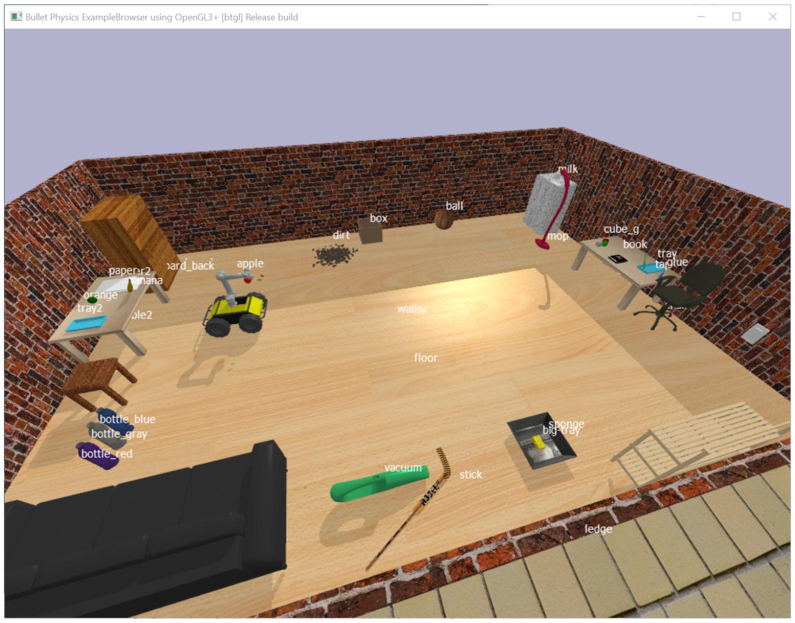

Figure 1: A simulation environment based on PyBullet physics simulator

3 Planning problem

The planning problem concerns with representing the domain of the problem and writing the planning algorithm, so that robot can perform various semantic tasks in different environments without fail. The planning problem representation and algorithm is explained in subsequent sections.

3.1 Domain representation

The given problem of symbolic planning of robot in a given room environment can be represented by the following:

- **States**: Each state in the state space specifies the location of all objects in the environment. The world state at any time instant is of the form:
  {‘grabbed’: ‘’, ‘fridge’: ‘Close’, ‘cupboard’: ’Close’, ‘inside’: [ ], ‘on’: [ ], ‘close’: [ ]}
- **Actions**: Each state can have 5 set of actions: move to, pick, place on, push to, open/close. Given the state s, each action has its own **preconditions** and **effects**, denoted by PRECOND(s,a) and EFFECTS(s,a), respectively. Precondition is a set of constraints on s that allow action a to be executed and effects is the set of the constraints that must be satisfied by next state $s^{'}$ when action a is executed on s
- **Goal test**: This checks whether the state satisfies the goal constraints.
- **Path-cost**: This includes cost of the path from the initial state to the current state. This also includes heuristic cost to accelerate the search. Thus, the total path cost is the cost of step cost and heuristic cost associated with the state.

3.2 Planning algorithm

To generate a feasible plan, planning algorithm needs to search the state-space of the given environment. This search requires a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains following components:

- NODE: It corresponds to the states in the state space of the problem. For each node n of the tree, we have a structure that contains components: ID (as node number), STATE, PARENT, ACTION. PATH-COST. It is represented as:

  {ID: {‘parent’: parent-id, ‘action’: action-type, ‘State’: state, ‘path-cost’: cost-value}}
- *explored*: This is a **queue** data structure that contains nodes already expanded. It helps in avoiding redundant paths.
- *frontier*: This is another **queue** that contains the node to be explored. This is known as **priority queue** as nodes in this are ordered by the PATH-COST.

The planning can be done using the most commonly known techniques of state-space search i.e. Forward and Backward planning [3]. These techniques are explained in further sections. The code for the same is available at [2].

3.3 Forward Planning

In forward planning, the search starts from initial state and then feasible actions are applied to each state to obtain new states till the goal state is reached. The algorithm used is given below. The **uniform-cost search** strategy is used which expands the node with the lowest path cost. Each step cost has been used as 1 in this planner. The PRECOND and EFFECTS has been encoded in the *checkAction*(s,a) and *changeState*(s,a) function in *environment.py* file. The goal constraints has been encoded in the *checkGoal(s)* function.

Algorithm 1 FP-UCS: Forward planning algorithm with uniform cost search

```
Input: Initial state ($s_i$), Goal constraints (g), PATH-COST= 0
Output: Plan of sequence of actions: $\pi(s)$
Data: PRECOND(S,A) ← preconditions of action $a$ to be feasibe at states $s$
EFFECTS(S,A) ← effects of applying action $a$ on states $s$
REPEATED ← checks if state is already in tree
frontier ← a priority queue ordered by PATH-COST
explored ← an empty set
 1: while goal not reached do
 2:     $s$ ← POP(frontier)
 3:     if g ⊆ s  then
 4:         goal reached return $\pi(s)$
 5:     else
 6:         add $s$ to explored
 7:         find set of feasible actions $A(s)$ at state $s$ using PRECOND(S,A)
 8:         for each a in $A(s)$ do
 9:             $s^{'}$ ← EFFECTS(S,A): the state $s^{'}$ attained after applying action $a$
10:             if $s^{'}$ not in explored or frontier and $s^{'}$ is not REPEATED then
11:                 PATH-COST ← COST-FUNCTION($s, a, s^{'}, g$)
12:                 frontier ← INSERT(($s^{'}$,PATH-COST), frontier)
13:             end if
14:         end for
15:     end if
16: end while
```

3.4 Backward Planning

In backward planning, we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. The **breadth first search** strategy is used in which all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The PRECOND and EFFECTS has been encoded in the *checkActionBwd*(s,a) and *changeStateBwd*(s,a) fucntion in *environment.py* file. The initial state constraints has been encoded in the *checkInitial(s)* function.

Algorithm 2 BP-BFS: Backward planning algorithm with breadth first search

```
Input: Initial state ($s_i$), Goal constraints (g)
Output: Plan of sequence of actions: $\pi(s)$
Data: PRECOND(s,a) $\leftarrow$ preconditions of action $a$ to be feasibe at states $s$
EFFECTS(s,a) $\leftarrow$ effects of applying action $a$ on states $s$
frontier $\leftarrow$ a FIFO queue with node as the only element
explored $\leftarrow$ an empty set
1: while goal not reached do
2:     $s \leftarrow$ POP(frontier)
3:     add $s$ to explored
4:     find set of feasible actions $A(s)$ at state $s$ using PRECOND(s,a)
5:     for each a in $A(s)$ do
6:         $s^{'} \leftarrow$ s - EFFECTS(s,a) $\cup$ PRECOND(s,a)
7:         if $s^{'}$ not in explored or frontier then
8:             if s $\subseteq s_i$ then
9:                 goal reached return $\pi(s)$
10:            else
11:                frontier $\leftarrow$ INSERT($s^{'}$, frontier)
12:            end if
13:        end if
14:    end for
15: end while
```

3.5 Accelerated Planning

To synthesize feasible plans in a short amount of time, heuristic based search strategy has been implemented. It has been analysed that there are lot of irrelevant actions and parent-child pair occuring in the search tree. Such actions and pairs have been given high-cost penalty and thereby avoided for exploration during search. The COST-FUNCTION in Algorithm 1 for forward planning has been changed to include various heuristics. This algorithm is named as **Accelerated Forward Planning - Heuristic Based Search (AFP-HBS)** and is given in Algorithm 3.

Similarly, it has been noticed during backward planning that *moveTo* action can have many predecessor states. For example, *'moveTo, fridge'* can bring any object from *fridge, cupboard, table, table2*, etc. in the environment to *close_list*. To avoid this, weight function has been used to prioritize random selection of object based upon the initial object location available from the initial state. This algorithm is named as **Accelerated Backward Planning - Heuristic Based Search (ABP-HBS)** and is given in Algorithm 4.

3.6 Comparison of forward and backward planning

The following advantages and disadvantages of forward and backward planning has been observed:

Algorithm 3 AFP-HBS: Heuristics based search strategy for forward planning

```
a(s): action to achieve state s
HELPFUL-ACTIONS: actions relevant to reach g
PICKABLE-OBJECTS: objects in the environment that can be grabbed
c[i]: conditions returning TRUE or FALSE
COST(s): count of number of action steps to reach state s
THRESHOLD: estimate of maximum number of action steps to reach goal state
HEURISTIC-COST(c[i]): heuristic cost of state based on condition c[i]
1:  function COST-FUNCTION($s, a, s', g$)
2:      $c[0] \leftarrow a$ not in HELPFUL-ACTIONS
3:      $c[1] \leftarrow a = a(s') =$ moveTo
4:      $c[2] \leftarrow a =$ [moveTo, PICKABLE-OBJECTS] and s[grabbed]
5:      $c[3] \leftarrow a = a(s') =$ changeState
6:      $c[4] \leftarrow (a =$ pick and $a(s') =$ drop) or ($a =$ drop and $a(s') =$ pick)
7:      $c[5] \leftarrow$ COST(s) $>$ THRESHOLD
8:      if c[i] then
9:          COST($s'$) $\leftarrow$ COST($s$) + HEURISTIC-COST(c[i])
10:     end if
11: end function
```

Algorithm 4 ABP-HBS: Heuristics based search strategy for backward planning

```
OBJECTS-LIST: list of the objects in the given environment
WEIGHT: weight vector for prioritizing random selection of object
RANDOM: function to select random entry from a list
1: function EFFECTS($s, a, g$)
2:     if a is moveTo then
3:         obj ← a[1]
4:         $s'$[‘close’] ← RANDOM(OBJECTS-LIST,WEIGHT)
5:     end if
6: end function
```

Advantages of forward planning:

a) It can be used in conjunction with any search strategy: breath-first, depth-first, uniform-cost, greedy-search, A^*, etc.

b) It is complete i.e. it returns a solution if there is one

c) It is sound i.e. any solution found is a good solution.

Disadvantages of forward planning:

a) It is prone to exploring irrelevant actions.

b) Average branching factor is huge for large state spaces

Advantages of backward planning:

a) It can also be used in conjunction with any search strategy: breath-first, depth-first, iterative-deepening, greedy-search, A^*, etc.

b) Backward search keeps the branching factor lower

Disadvantages of backward planning:

a) It is difficult to implement when regression from state description to predecessor state description is not known

b) Since backward search uses state sets rather than individual states, thus makes it harder to come up with good heuristics.

4 Results and Analysis

In this section, the various algorithms are applied on different goals in various environment to check their robustness.
For example, the plan obtained for a goal: *"Put all fruits in a fridge"* in a world 3 is as follows:

Initial State: 'grabbed': '', 'fridge': 'Close', 'cupboard': 'Close', 'inside': [('apple', 'cupboard'), ('orange', 'cupboard'), ('banana', 'cupboard')], 'on': [('tray', 'table'), ('tray2', 'table2')], 'close': [ ]

Goal state: 'grabbed': '', 'fridge': 'Close', 'cupboard': 'open', 'inside': [('apple', 'fridge'), ('orange', 'fridge'), ('banana', 'fridge')], 'on': [('tray', 'table'), ('tray2', 'table2')], 'close': ['fridge']

Plan: [[moveTo , fridge], [changeState , fridge , open], [moveTo , cupboard], [changeState , cupboard , open], [pick , apple], [moveTo , fridge], [drop , fridge], [moveTo , orange], [pick , orange], [moveTo , fridge], [drop , fridge], [moveTo , banana], [pick , banana], [moveTo , fridge], [drop , fridge], [changeState , fridge , close]]

4.1 Comparison of running time

The running time is the plan generation time. The planner is made to run on the available machine with following configuration: *Intel Core i3-3227U CPU @ 1.90 GHz 4, Ubuntu OS 64 bit, 8 GB memory, 500 GB HDD, AMD Radeon(TM) R3 Graphics.* The results will differ on other machines. The running time of different planners described above are compared for different goals in all the given environments, as given in Table 1

Note: + sign in the superscript indicates that time given is for subgoals and real time for planning would be more than this. For example, when it is asked to place 3 fruits, I noted the time with two fruits only. The actual time was difficult to compute while using non-accelerated search strategies due to hardware limitations.

Table 1: Comparing running time of different planning algorithms

Planner	FP-UCS					BP-BFS				
Worlds	w_0	w_1	w_2	w_3	w_4	w_0	w_1	w_2	w_3	w_4
goal0	.24	.18	.15	.07	.49	.04	.08	.01	.03	.03
goal1	166^+	160^+	130^+	335^+	298^+	49	81	95	102	109
goal2	8.9	8.2	4.5	8.4	0	.05	.09	.01	1.05	0
goal3	355^+	351^+	403^+	333^+	0	271.6^+	145.3^+	178^+	302^+	0

Planner	AFP-HBS					ABP-HBS				
Worlds	w_0	w_1	w_2	w_3	w_4	w_0	w_1	w_2	w_3	w_4
goal0	.04	.01	.06	.03	.03	.01	.01	.01	.02	.02
goal1	.23	.15	1.9	.54	.55	.53	.17	3.2	2.7	2.8
goal2	.02	.01	.07	.07	0	.01	.00	.01	.01	0
goal3	.67	.59	8.7	2.5	0	10	1.5	55.1	88	0

4.2 Comparison of branching factor

Branching factor is used to characterize the quality of search algorithm. In tree data structures, the branching factor is the number of children at each node. If this value is not uniform, an average branching factor can be calculated. The **Average Branching Factor (ABF)** can be quickly calculated as the number of non-root nodes (the size of the tree, minus one; or the number of edges) divided by the depth of tree. This quantity is calculated for above mentioned planners and tabulated in Table as shown below.

Planning Problem 1: Initially banana is inside cupboard. Goal is to put banana inside the fridge

Planning type	Number of leaves	Depth of tree	ABF
AFP-HBS	125	9	13.89
ABP-HBS	370	8	46.25
BP-BFS	906	10	90.6
FP-UCS	1221	9	135.67

Planning Problem 2: Initially apple on table 2 and tray on table. Goal is to put apple on tray and put tray in fridge.

Planning type	Number of leaves	Depth of tree	ABF
AFP-HBS	138	11	12.54
ABP-HBS	2628	9	292

5 How to run the code?

The syntax for code are as follows:

```
python planner.py –world jsons/home_worlds/world_home0.json –goal jsons/home_goals/goal0.json
```

Note 1: The accelerated forward planner (AFP-FBS) is set as default. This can be changed to other planners using *planner_type* varibale in *planner.py* file.
Note 2: Please try to switch planners 0 and 1 if plan is taking too much time in any one of them. In case of planner 1, try re-running if it's taking too much time. There is some randomness involved in this planner

References

[1] Assignment problem statement for the COL864 - Special topics in AI course at IITD. `https://github.com/reail-iitd/COL864-Task-Planning`. Accessed: 04-Apr-2020.

[2] Assignment solution code for the COL864 - Special topics in AI course at IIT Delhi. `https://github.com/reail-iitd/homework-2-deepakraina99`. Accessed: 04-Apr-2020.

[3] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.