Assignment 2
COL864: AI for Robot Intelligence

Navreet Kaur
Indian Institute of Technology,Delhi
2015TT10917
tt1150917@iitd.ac.in

THE ENVIRONMENT

The assumptions of feasibility of an action and how these actions can change the world state are given in the table below in the form of pre and post conditions of an action in a given state:

Action	Pre-Condition	Post-Condition
`moveTo, object`	● *object* should not be *'close'* to the robot ● *object* should not be *'grabbed'* by the robot	● *object* is *'close'* to robot ● if *object* is an enclosure, all the objects inside it are also *close* to the robot
`pick, object`	● nothing should be 'grabbed' ● *'object'* to be picked must not be heavy, eg, table, fridge, cupboard etc ● *'object'* to be picked must not be inside a closed enclosure, eg, if *'apple'* has to be picked, it should not be *'inside' 'fridge'* in that is *'close'*	● *object* is *grabbed* ● *object* is *close* to robot ● if *object* was *inside* or *on* something, it no longer remains so
`drop, destination`	● something should be *grabbed* ● *grabbed* object must not be same as *destination* ● *destination* must not be a *closed* enclosure ● *destination* must be a surface or an enclosure	● if *destination* is an enclosure or surface, place the *grabbed* object *inside* or *on* it respectively ● robot is not *close* to anything ● robot has not *grabbed* anything
`changeState, enclosure, state`	● must be *close* to the *enclosure* ● if state is to be changed to *open*, it should *close* initially and vice versa	● state of the *enclosure* changes to *state* ● robot is not *close* to anything
`pushTo, object, destination`	● nothing should be *grabbed* ● *object* should be *close* to robot ● *object* and *destination* should not be the same ● *object* to be pushed must not be a heavy object ● if *object* is inside an enclosure, it	● if *object* was *inside* or *on* something, it no longer remains so ● if *destination* is a surface, the *object* gets placed *on* it ● robot is close to *object* and *destination* after pushing

	should be in *open* state	● robot has not *grabbed* anything

In the above table, *enclosures* refers to *fridge* and *cupboard*; *surfaces* refer to *table, table2, tray* and *tray2.* It is assumed that the robot can only pick/grab light objects like *apple, banana, orange, tray,* and *tray2.*

The method `checkAction` in `environment.py` checks the pre-conditions of an action in the given state and `changeState` executes the action in a state and gives a new state with the mentioned post-conditions.

The goal checking function `checkGoal` was updated to check the goal according to the input goal json file.

Domain Representation

State is represented as a dictionary as follows:

```
{'grabbed': '', 'fridge': 'Close', 'cupboard': 'Close', 'inside': [], 'on':
[], 'close': []}
```

Actions are represented as:

- `[moveTo, object]` - moves robot close to object
- `[pick, object]` - picks the specified object
- `[drop, destination]` - drops a grabbed object to destination object
- `[changeState, object, state]` - changes the state of an object (open or close)
- `[pushTo, object, destination]` - pushes object close to the destination object

Planning Algorithm

The planner was implemented in `getPlan` method in `planner.py`

Breadth First Search was used for *forward planning*. It expands the shallowest node first. Even though it is complete, optimal for unit step costs, it has exponential space complexity and hence can take a lot of time to generate a plan.

Best First Search was used with **heuristics** for *accelerating* the plan generation. It selects a node for expansion according to an evaluation function.

The pseudocode of the planning algorithm implemented is as follows:

```
function BEST-FIRST-SEARCH (initial_state) returns a solution, or failure

    node ← a node with STATE = init_state, PATH-COST = 0, PARENT=None
    if checkGoal(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element and key as the path-cost
    explored ← an empty set
    while frontier not empty
```

```
        node←POP(frontier) /*chooses the shallowest node in frontier */
        add node.STATE to explored
        actions←GET-VALID-ACTIONS(node.STATE) /*heap of valid actions with key as cost of taking that action*/
        while actions is not empty:
            child_cost, action←POP(actions)
            child←a node with STATE = changeState(node.STATE, action),
                                PATH-COST = cost+child_cost,
                                PARENT = node
            if child.STATE is not in explored or frontier then
                if checkGoal(child.STATE) then return SOLUTION(child)
                frontier←INSERT(child,frontier)
    return failure
```

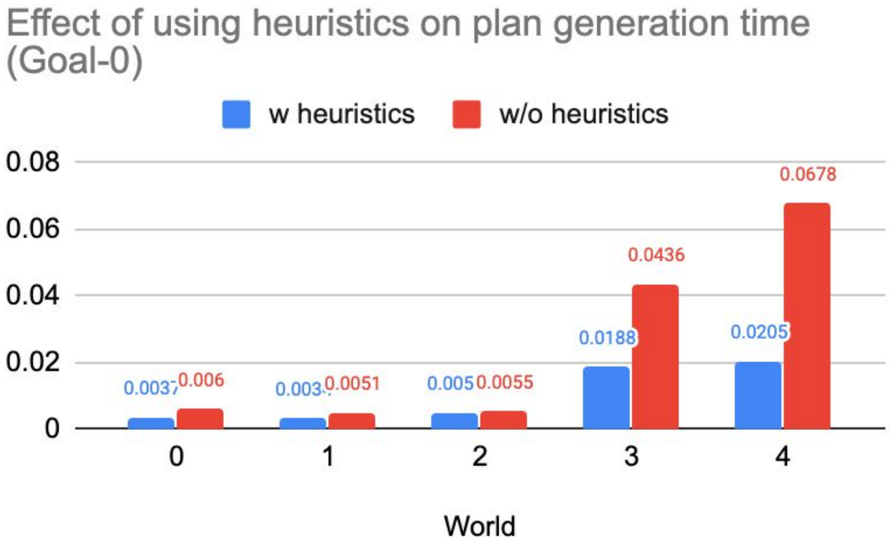
For each node *n* of the search tree, the structure is as follows:

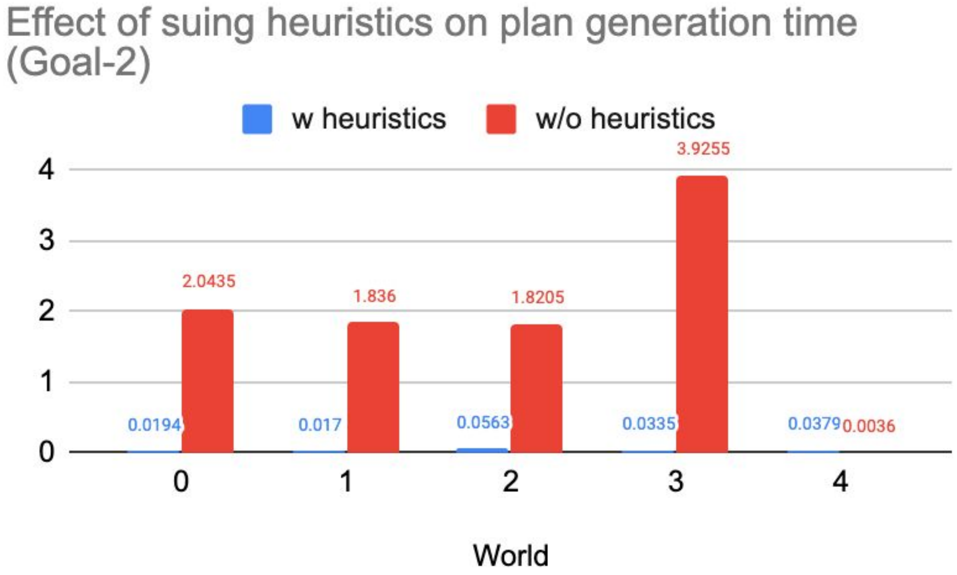
- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost of the path from the initial state to the node, as indicated by the parent pointers.

The PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found; the SOLUTION function is used to return the sequence of actions obtained by following parent pointers back to the root.

The method GET-VALID-ACTIONS checks the valid actions from a state and also provides the cost of taking those actions, which acts as a heuristics to guide the search. Some examples of the heuristics used are giving more reward to *moveTo* objects which are *on* a surface in the goal state, *moveTo* an enclosure if has objects inside it in goal state which are not present in current state, *pick* an object if it is *inside* or *on* something in the goal state etc. The amount of reward to be given to each of such actions was decided by hit and trial and extensive experimentation on different worlds and goal states.

A comparison between the times for plan generation for Goal-0 using forward planning with and without heuristics is shown below:





It can be noticed that using Heuristics leads to significant improvement in the time taken for plan generation. This is due to the reduction in the effective branching factor of forward planning due to selective expansion of nodes which are closer to the goal. The running complexity of BFS, without any heuristics is exponential in terms of the branching factor and it also requires large memory, again exponential in terms of branching factor. On the other hand, Best First Search with greedy expansion of nodes has less complexity in terms of both time and space, it's only disadvantage being that it may return suboptimal plans. Since a heap was used for storing the *frontier*, the worst case time complexity of accelerate forward planning is O(n*log(n)), n being the number of nodes in the search tree. Performance of this algorithm wlaos depends on how well the cost function is designed. The branching factor for the search depends on the average number of valid actions at a given state. For the forward planning, the valid actions at a state range from 10 to 40 actions. Using accelerated planning results in a speeds up the plan generation by 50x

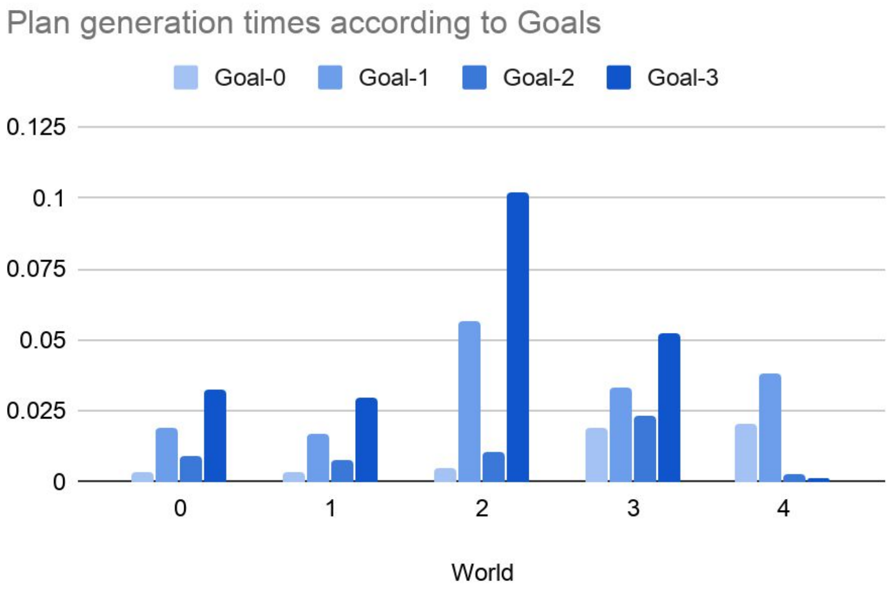
The algorithm for *backward planning* is as follows:

```
function BACKWARD-SEARCH (initial_state, goal_state) returns a solution, or failure
    PLAN = []
    g ← goal_state
    loop
        if checkGoal(init_state) return PLAN
            A' ← {a ∈ A | a is relevant for g}
            if A' = Φ then return failure
            non-deterministically choose a ∈ A'
            g ← changeStateInverse(g, a)
            PLAN += {a}
    return PLAN
```

The method *changeStateInverse(g ,a)* gives outputs state on which if action *a* is executed will go to state *g*

The time for taken for plan generation in different worlds for different goals are shown in the following table:

World	Goal-0	Goal-1	Goal-2	Goal-3
0	0.0037	0.0194	0.0092	0.0325
1	0.0034	0.017	0.0077	0.0299
2	0.0051	0.0563	0.0106	0.1017
3	0.0188	0.0335	0.0236	0.0526
4	0.0205	0.0379	0.0027	0.0015



It is noticed that for more complex goals, more time is taken for plan generation.
Forward planning is faster for more complex goals as compared to backward planning. Backward planning is faster for less complex goals and shorter plans.

Plans generated by accelerated forward search for all goals for worlds 0 and 3 are given below:

World	**0**	**3**
Goal-0	[['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'table'], ['drop', 'table']]	[['moveTo', 'cupboard'], ['changeState', 'cupboard', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'table'], ['drop', 'table']]
Goal-1	[['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'box'], ['drop', 'box'], ['moveTo', 'banana'], ['pick', 'banana'], ['moveTo', 'box'], ['drop', 'box'], ['moveTo', 'orange'], ['pick', 'orange'], ['moveTo', 'box'], ['drop', 'box']]	[['moveTo', 'cupboard'], ['changeState', 'cupboard', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'box'], ['drop', 'box'], ['moveTo', 'banana'], ['pick', 'banana'], ['moveTo', 'box'], ['drop', 'box'], ['moveTo', 'orange'], ['pick', 'orange'], ['moveTo', 'box'], ['drop', 'box']]
Goal-2	[['moveTo', 'fridge'], ['changeState', 'fridge', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'fridge'], ['drop', 'fridge']]	[['moveTo', 'fridge'], ['changeState', 'fridge', 'open'], ['moveTo', 'cupboard'], ['changeState', 'cupboard', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'fridge'], ['drop', 'fridge']]
Goal-3	[['moveTo', 'fridge'], ['changeState', 'fridge', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'banana'], ['pick', 'banana'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'orange'], ['pick', 'orange'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'fridge'], ['changeState', 'fridge', 'close']]	[['moveTo', 'fridge'], ['changeState', 'fridge', 'open'], ['moveTo', 'cupboard'], ['changeState', 'cupboard', 'open'], ['moveTo', 'apple'], ['pick', 'apple'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'banana'], ['pick', 'banana'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'orange'], ['pick', 'orange'], ['moveTo', 'fridge'], ['drop', 'fridge'], ['moveTo', 'cupboard'], ['changeState', 'cupboard', 'close'], ['moveTo', 'fridge'], ['changeState', 'fridge', 'close']]