Rishi Basdeo
16-782: Planning and Decision Making

Homework 3 Report

For my domain-independent planner, I first modified the provided Action class to make the process of evaluating the validity of an input for a given action. This was done by adding an unordered map class variable, where all symbols are mapped to themselves (ie. representing both the key and the value) in the constructor, along with any arguments from the function. For example, in the block world problem for the MoveToTable function, A,B,C,Table, x, and b are mapped to themselves. Then, in the newly added generate_symbol_map function, a copy of this map is made and modified to associate the original function inputs (eg. b and x) to the actual arguments being evaluated (eg. A and B). This added functionality enabled me to include other methods such as preconditions_satisfied, which makes these substitutions into the preconditions of the action, and then compares them to the current state to determine whether the action can be performed. A similar process is completed for evaluating the effects of an action. It is worth noting that I did it this way so that no additional Action objects would be required in my planner other than the ones generated during the initial import process.

In addition to modifying the originally provided code, I wrote a custom planner class (symbo_planner), which contains a private class, called symbo_node. The latter forms the building blocks of the map that would be built during the planning process. Each node contains a state (set of conditions), the name of the action leading to that state along with its inputs, its f, g, and h values (for A* planning). Lastly, it contains a pointer to the preceding node (parent), and a vector of pointers to subsequent nodes (children).

These nodes are manipulated by my symbo_planner class, which contains all of the data structures from the environment such as the sets of actions and symbols. The open list is represented by a priority queue of pointers to each node which is sorted based on the f value assigned to each node; the closed list is an unordered set of node pointers. Whenever a new node is pulled from the open list, neighbors are generated by iterating over all available actions, along with all possible permutations of input symbols to determine their compatibility with the state associated with that node. Only those action/input pairs whose preconditions are satisfied are used to generate subsequent nodes, which are then added to the open list. I weighed the cost of each action as 1, and used a heuristic epsilon (weight) of 3.

I evaluated three possible heuristics. The first was zero/no heuristic, where the h values for all nodes were set as zero. The second simply counted the number of goal conditions that were not satisfied by the current state. The last was the empty-delete-list heuristic, where the entire planner was replicated with two modifications: (1) the new start state was the state whose heuristic was being evaluated and (2) all actions are incapable of deleting conditions. The heuristic was defined as the number of actions required to get to the goal state under these conditions. A comparison of these found that the best performing heuristic overall was the missing conditions heuristic, which was the fastest across all environments, even though the generated plan was not completely optimal. It was also found to have the fewest number of expanded states overall in more complex environments. This data is summarized in Table 1.

The zero and empty-delete-list heuristics performed significantly slower on the blocks and triangles and fire extinguisher environments, respectively.

Table 1: Planning Time by Heuristic and Planning Environment

Heuristic	**Blocks**			**Blocks and Triangles**			**Fire Extinguisher**		
	Planning Time (s)	**# of Actions**	**# Expanded States**	**Planning Time (s)**	**# of Actions**	**# Expanded States**	**Planning Time (s)**	**# of Actions**	**# Expanded States**
None/ Zero	0	3	16	153	6	946	8	21	348
Missing Conditions	0	3	5	1	7	7	7	21	332
Empty-Delete-List	0	3	4	4	6	9	>120	N/A	N/A

To compile and run my code, follow these instructions:

```
>> g++ planner.cpp –o planner.out
>> ./planner.out  <path to environment description file>
```