PLANNING HW3

NINAAD DAMIS
ndamis@andrew.cmu.edu

INDEX

1. COMPILATION AND RUNNING CODE

1. g++ -std=c++11 planner.cpp -o planner.o
2. ./planner.o

2. IMPLEMENTATION DETAILS

Data Structures :

State -

State object is defined as a typedef of `unordered_set<GroundedCondition, GroundedConditionHasher, GroundedConditionComparator>`.

Open List -

The open list priority queue is defined as a set. Set < pair < cost, State > > . Each element in the set is a pair containing the f value (cost) of that state and the corresponding State. The elements are automatically ordered in ascending order of their costs.

Closed List -

The closed list is implemented as an unordered_set. unordered_set<State,StateHasher>.
The hashing function StateHasher is used to hash the State objects.

Node -

A class called Node has been defined to store the required information for the A* search algorithm. It has the following parameters - f (cost) , g, h (heuristic value), parent state and the grounded action that generated the state.

Graph -

The graph is implemented as an unordered_map. unordered_map<State,Node>. The key is a given State and the value is of type Node which stores the information of that State.

Functions :

generateSymbolsPermutation() :

For each given action in the environment object, this function takes in the list of symbols and the number of arguments required for the action, and returns a vector<vector<string>> which contains all possible permutations of symbols for the given action.

generateSymbolsPermutationWithRepetition()

Similar to above function, but generates permutations with repetitions allowed. This is the function which has been used to generate the list of all actions. Allowing repetition of symbols increase the action space, but makes the planner more domain independent (none of the given domains need repetition of symbols)

generateAllPossibleActions()

This function takes in the given environment as input, and generates a list of all possible actions that are valid for the given environment. It returns a vector of pointers to Action object (vector<Action*>).

generateSuccessorState()

This function takes in a given state and action as input, and generates the successor state.

getValidActions()

This function takes in a given state and a vector of all possible actions, and returns every valid action from the given state. The valid actions are returned as a vector<Action*>.

getHeuristic()

This function takes in the current state and the goal state, and returns the value of the heuristic of the current state.

3. RESULTS

Without Heuristic

Environment	Time Taken (s)	Number of Expanded States
Blocks	0.0230471	29
Blocks and Triangles	56.9882	10,023
Fire Extinguisher	1.84509	1500

With Heuristic (#goal literals not matching heuristic)

Environment	Time Taken (s)	Number of Expanded States
Blocks	0.0159121	5
Blocks and Triangles	1.13616	272
Fire Extinguisher	1.62216	1376

4. DISCUSSION

We see an overall improvement in the planning time and the number of expanded states in all three domains when a heuristic is used.

The heuristic I have used is the number of goal literals missing.

In the Blocks environment, using the heuristic reduces the planning time by 30% and reduces the number of expanded states by ~6X.

The effectiveness of the heuristic is most drastically seen in the Blocks and Triangle environment. There is a 50X decrease in the planning time and ~37X reduction in the number of expanded states.

The heuristic used is not very effective in the Fire Extinguisher environment, as the heuristic is uninformative for this particular domain. As there is only one literal in the goal state, the heuristic value for all states (except goal) will be 1, and thus there is not much improvement in the planning time and number of expanded states.