Symbolic Planning

Notations:

- **Class** Node:

Members:

1. Literal: Unordered set of all the conditions at a particular state of the environment.
2. Parent: Parent Node of that Node.
3. Name: If any name is assigned (Just for reference)
4. h: Heuristic cost
5. g: optimal cost of the node.
6. f: g + h
7. ga: ground action that led to the existence of this node.
8. Print(): Prints all conditions of the Node.

2 nodes are equal if their unordered_set of literals are equal meaning each condition is present in both nodes and no extra conditions are present.

- Updated the GroundAction class.

1. To incorporate the default constructor and a new constructor that takes an object of the "Action" class and converts that object into GroundAction class with required symbols.
2. New member variables: grounded_preconditions and grounded_effects.

Also wrote getter functions for them. (Redundant as I finally made those members public).

- Updated Env Class to get initial and goal conditions.

Functions and Implementation:

1. Heuristic:
   Calculates the number of unsatisfied conditions compared to goal node conditions.

2. IsValidAction:
   Checks if the action is valid on a particular node. Checks if all the preconditions for the action are present in the node literals.

3. IsGoal:

Checks if the current node is the goal Node by comparing literals.

4. IsVisited :
   Checks if the current node has already been created and saved into memory. If yes, it return the saved version of the node to recover other members of that node (g,h,f etc). Else it returns the input node.

5. ReturnNewLiterals:
   Returns the new literals (unordered set of new conditions) which have been created after performing a certain action.

6. Get_all_combinations:
   Given a vector of strings of symbols (total symbols = n) and the number of elements in the combination (k), returns a list of all the (n choose k) combinations of symbols.

7. Get_all_symbol_permutations:
   Given an unordered set of all symbols (total symbols = n) and the required number of elements (k) in each permutation, the function finds all the permutations and combinations of that size. Returns list of all the lists containing permutations.

8. Process_valid_actions:
   Given a parent node, an action, and a set of all symbols:

a. Finds all the permutations of symbols with a size equal to the number of symbols required for the action.
b. Checks if the action is valid on the parent node considering each set of symbols from the permutations obtained in step (a).
c. If even not a single action is not valid in step (b), returns false.
d. If the action is valid from step (b), for every action:
   i) Find the new set of conditions obtained by performing the action on the parent node.
   ii) Create a new node with conditions from (i) as literals.
   iii) Check if the new node already exists or not. If it does not exist, continue to step iv). If it exists, replace the new node with it. Also, check if the cost of

reaching the replaced node is less than its original cost. If not, continue to step (b).
iv) For the node received from (iii), update all member functions. Save the action that led to its existence, save the h_cost, g_cost and f_cost.
v) Check if the new node is in fact the goal. If yes, break the function and directly return true from the Process_valid_function.
vi) If not goal and if the node was new (not visited from step iii), add it to the priority queue to be used in the function and also add it to the vector of all the nodes and return false.

9. Expand_nodes:
   This is the main loop of the planner. It keeps on expanding the states until the goal is found.

a. Find/Get all the symbols that are present in the environment provided.
b. Find/Get all the actions that are present in the environment provided.
c. Create a priority queue that returns a node with the highest f-cost. If f_costs are the same, the queue returns the node with minimum h_cost.
d. Start a while loop until the priority queue is empty or a certain number of iterations are reached (For avoiding infinite looping in case of some bug).
e. In each while loop, the top() from the priority queue is expanded, meaning Process_valid_actions() function is called for every available action.

10. When the goal is found, the path is backtracked using the parent member of the nodes, and the actions are recorded. In the end, the path is reversed and returned.

Compilation:
g++ planner.cpp –o planner.out

Note:
- Early stopping: A* stops when the goal is in the open list.
- While submitting, I am submitting the 0 heuristic setting with no early stopping. Instructions to uncomment particular code for using heuristic and/or early stopping are mentioned in the code.

Results and Conclusion:

Using the basic heuristic that counts the number of not-satisfied literals reduces time in cases where we have multiple conditions to be satisfied at the goal. More the conditions at the goal, the better will this heuristic work. The difference in time and nodes explored with and without the heuristic is visible in BlockTriangle.txt.

But that heuristic is useless in cases such as FireExtinguisher.txt where we only have 1 condition to be satisfied at the goal. In that case, the heuristic value is always constant (equal to 1) throughout the path until the node satisfying all the preconditions for the last action arrived in the open list. As a result of that, using that heuristic reduced a very insignificant amount of time which is visible in the results.

1. Blocks.txt

Environment: Blocks	Time (s)	Number of Total Nodes	Number of Nodes Expanded	Number of Actions	Plan
0 heuristic	0.024	16	13	3	MoveToTable(A, B) Move(C, Table, A) Move(B, Table, C)
Early Stopping with 0 Heuristic	0.018	16	11	3	MoveToTable(A, B) Move(C, Table, A) Move(B, Table,C)
Type 1 heuristic: # Unsatisfied states	0.009	13	5	3	MoveToTable(A, B) Move(C, Table, A) Move(B, Table,C)
Early Stopping with type 1 heuristic	0.007	11	4	3	MoveToTable(A, B) Move(C, Table, A) Move(B, Table,C)

2. BlocksTriangle.txt

Environment: Blocks Triangle	Time (s)	Number of Total Nodes	Number of Nodes Expanded	Number of Actions	Plan
0 heuristic	21.612	2147	968	6	MoveToTable(T1, B3) MoveToTable(T0, B0) MoveToTable(B0, B1) Move(B1, B4, B3) Move(B0, Table, B1) Move(T1, Table, B0)
Early Stopping with 0 Heuristic	6.843	996	336	6	MoveToTable(T1, B3) MoveToTable(T0, B0) MoveToTable(B0, B1) Move(B1, B4, B3) Move(B0, Table, B1) Move(T1, Table, B0)
Type 1 heuristic: # Unsatisfied states	4.273	598	188	7	MoveToTable(T1,B3) MoveToTable(T0,B0) MoveToTable(B3,B2) Move(B0,B1,B2) Move(B1,B4,B3) Move(B0,B2,B1) Move(T1,Table,B0)
Early Stopping with type 1 heuristic	3.543	557	179	7	MoveToTable(T0,B0) MoveToTable(B0,B1) Move(T1,B3,B0) Move(B1,B4,B3) Move(T1,B0,B4) Move(B0,Table,B1) Move(T1,B4,B0)

3. FireExstinguisher.txt

Environment: Fire Extinguisher	Time (s)	Number of Total Nodes	Number of Nodes Expanded	Number of Actions	Plan
0 heuristic	3.739	386	366	21	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)
Early Stopping with 0 Heuristic	3.222	355	337	21	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)

Type 1 heuristic: # Unsatisfied states	3.15	380	342	21	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)
Early Stopping with type 1 heuristic	3.038	355	337	21	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)