AI PLANNING (CMSC722)

Project 1

Comparing Domain-Specific and Domain-Independent Planning

Author
Aalap Parimalkumar Rana

March 14, 2020

Contents

0.1 Introduction & Objectives 2
0.2 Approach 2
0.3 Generating PDDL file to represent the problem 2
0.4 Remodeling PDDL domain definition 3
0.5 Visualization and simulator for domain independent planner . 4
0.6 Comparison and Analysis 5
0.6.1 Analysis1: 5
0.6.2 Analysis 2 6
0.6.3 Comparison 1
GBFS-h_walldist Vs Domain Independent Planner . . 7
0.6.4 Comparison 2
GBFS-h_esdist Vs Domain Independent Planner . . . 8
0.7 Results & Conclusion 10

0.1 Introduction & Objectives

The assignment provides modified version of racetrack problem where all the walls are formed by straight lines and robot has to travel from start to end location under constraints. The problem is solved using domain independent planner and domain specific planner. The main aim of the assignment are:

1. Compare domain independent planner and domain specific planners for "Racetrack" problem.
2. Develop a code that generates a PPDL file to completely represents the problem under consideration with all specification.
3. Modification of domain independent planner to represent the same problem as specified by the domain specific planner.

0.2 Approach

To accomplish the above led objective the following steps were followed:

1. Understanding the domain independent planner defined for racetrack problem provided by "Patrik Haslum".
2. Setting-up required repositories and dependencies for smooth running of ENHSP.
3. Generate the PDDL files for different problems enlisted in sampleprob.py.
4. Developed a python code for visualization of the path generated by ENHSP for different problems.
5. Remodeling the PDDL domain definition provided by "Patrik Haslum" to represent the same problem as stated in NAU-plan.
6. Comparing the Nau-plan (domain specific) to domain independent plan based on the number of node generated, CPU run time and other parameters.

0.3 Generating PDDL file to represent the problem

To run the domain independent planner using ENHSP we require mainly two files:

1. PDDL domain file, and

2. PDDL problem initialization file.

This section covers description of python code to generate the PDDL problem initialization file. The code uses basic file operation and string manipulation operation to extract all critical information. Few important processes are mentioned below:

1. Extracting the problem : It was observed that each problem in sample_prob.py had a space after its name. Thus, "problem-name " sub string was passed to the find() to fetch the line where the problem was defined by using a counter.

2. Extracting start point and goal point: It was observed that the first line obtained by using "linecacheġetline()" had the start and goal. To obtain the start point a user defined function "start_point()" was used. Similarly, "goal_points()" was used to get the goal points. Both the function used "partition()" and "res.translate()" to remove unwanted symbols and text.

3. Extracting walls: A user defined function "walls()" was used to fetch the wall co-ordinates and this method functioned similar to "start_point()". It was observed that the conditions in the the PPDL domain definition wanted the walls to be defined as (bx,by)>(ax,ay).
   It was seen that the wall co-ordinates had either the x or the y co-ordinate same and thus, the required format was obtained by putting inequality constrains on other variable.

0.4 Remodeling PDDL domain definition

The domain defined for racetrack problem by Patrik Haslum is extremely simple with only a single horizontal wall and less number of moves. However, the racetrack problem given in the Nau-plan has walls with different slopes, larger number of valid move actions and different goal and start definition. Thus, to compare both the planners, Patrik Haslum's model was modified to represent the Nau-plan version of racetrack.
Firstly, the goal definition for given problem was to enter a specified region in space and move vertically. This goal definition was changed to reaching a finish line which could be either vertical or horizontal with zero velocity in both x and y direction.
Secondly, new moves were introduced to cover all the possible action that

could be generated from different combination of vx and vy velocities. Thus, a set of 9 action was obtained which had the following new action:
1. move_up (vx, vy+1) 2. move_down (vx, vy-1) 3. move_left (vx-1, vy)
4. move_right (vx+1, vy) 5. move_up_right (vx+1, vy+1)
6. move_up_left (vx-1, vy+1) 7. move_down_right (vx+1, vy-1)
8. move_down_left (vx-1, vy-1) 9. move_none (vx, vy).
Thirdly, the domain used slope intercept form to represent wall and therefore, vertical wall could not be defined as it had infinite slope. Thus, to represent vertical walls different set of preconditions were added using basic linear algebra and geometric concepts of lines to move actions. It was observed if the intersection point of the vertical wall and movement line was below or above the wall the movement could be taken without crashing into wall. Additionally any movement to the right or left of the wall was valid if it remained entirely to the any one side of the wall.

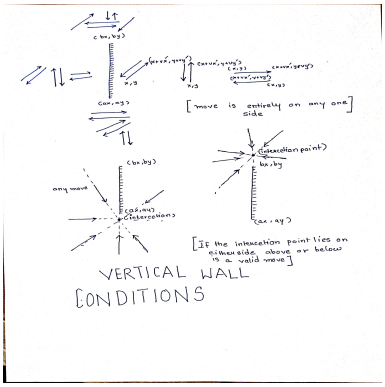

Figure 1: Vertical Wall and Movements

0.5 Visualization and simulator for domain independent planner

The output obtained after running ENSHP included information about number of node explored, path-length, and other important parameters but it didn't have a visual output of path formed and node exploration as obtained while running domain specific planner, displaying the path from start to goal. Thus, a simulator was developed in python.

The program takes in the different movements generated by the ENSHP to reach goal and stores it in a list called "action". This "action" list is converted to respective vx and vy velocities which are then stored in different lists (here vx and vy). The list of velocities are cumulatively added to gener-

ate a new list of movement components in x and y direction. This generated movement component list is appended by it's start co-ordinates, and this newly formulated list is again cumulatively added to obtain x co-ordinate and y co-ordinate of the car. Using few in-build "matplot.lib" functions a graph displaying walls and path is modeled. Moreover, this visual output also helped in debugging many pre-conditions in domain definition.

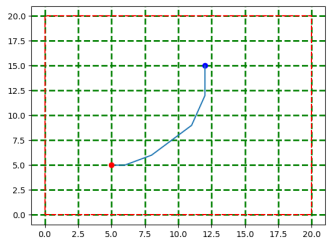

Figure 2: Simulation

0.6 Comparison and Analysis

If we consider "h_walldist" this heuristic function computes the approximate distance to goal considering the walls.It is more advance heuristics as compared to "h_esdist" as it cache a rough estimate of the length of the shortest path to the finish line when it is called for the first time. It implements breadth-first search in the backwards direction from the finish line, to each free-point in the space at a time. If we consider "h_esdist" it takes into consideration the euclidean distance from present state to finish line ignoring all the wall and adds the stop distance calculated using " m(m-1)/2".

0.6.1 Analysis1:

1. "h_esdist" generates less nodes as compared to "h_walldist" for simple obstacel like rect20 and others where there no or few walls in between finish line and start.However, when compared for complex problem like rectwall16, and pdes30 "h_esdist" has a larger exploration than "h_walldist" nearly 100 times more.

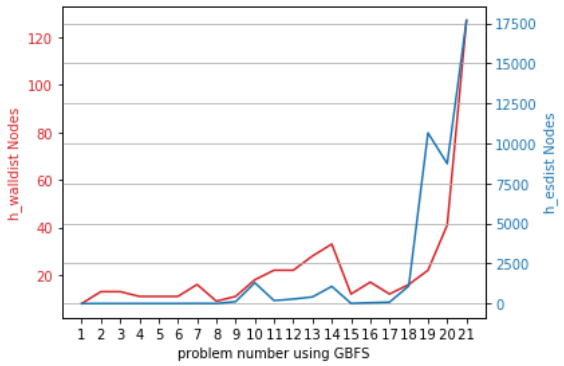

Figure 3: Node generation GBFS-h_walldist and h_esdist

0.6.2 Analysis 2

1. "h_walldist" takes less CPU runtime as compared to "h_esdist" for complex obstacle space with more walls.However, when compared for simple problem "h_esdist" and "h_walldist" have same results.

2. "h_walldist" requires more CPU time in the first run as compared to "h_esdist" but for all new problems with similar obstacle space but different goal "h_walldist" requires less time due to the property of storing rough estimate of the length of the shortest path to the finish line.

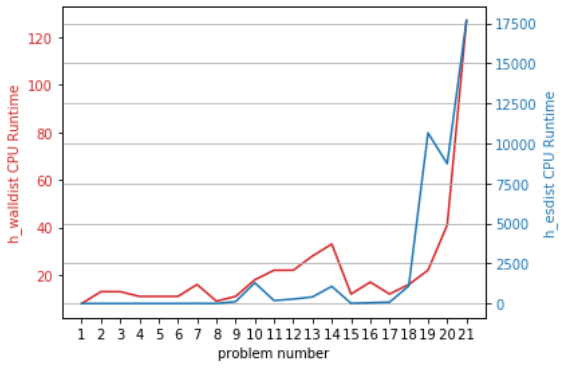

Figure 4: CPU runtime GBFS-h_walldist and h_esdist

0.6.3 Comparison 1 GBFS-h_walldist Vs Domain Independent Planner

Problem Name	GBFS-Node Using h_walldist	GBFS-CPU Runtime Using h_walldist (ms)	Domain-Independent Planner Node	Domain Independent Planner CPU Runtime (ms)
Rect20	8	1525	8711	8711
Rect20a	13	3210	5780	5780
Rect20b	13	1563	5925	5925
Rect20c	11	1534	4990	4990
Rect20d	11	1577	13650	13650
Rect20e	11	1550	10115	10115
Rect50	16	1735	16156	16156
Wall8a	9	3786	5395	5395
Wall8b	11	3711	4245	4245
Lhook16	18	4780	16783	16783
Rhook16a	22	6614	10438	10438
Rhook16b	22	7120	15505	15505
Spiral16	28	9429	19081	19081
Spiral24	33	35550	68672	68672
Pdes30	12	17055	27660	27660
Pdes30b	17	15525	35857	35857
Rectwall8	12	5973	4502	4502
Rectwall16	16	5613	23135	23135
Rectwall32	22	48606	214082	214082
Rectwall32a	41	49206	196021	196021
Twisty0	127	3600000	3600000	3600000

Table 1: Comparison GBFS "h_walldist" and Domain Independent Planner

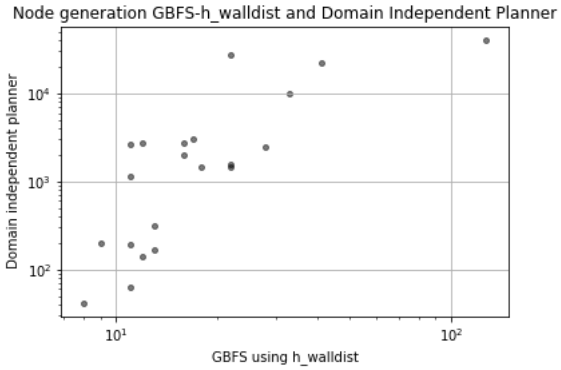

Figure 5: Node generation Domain independent planer and GBFS-h_walldist

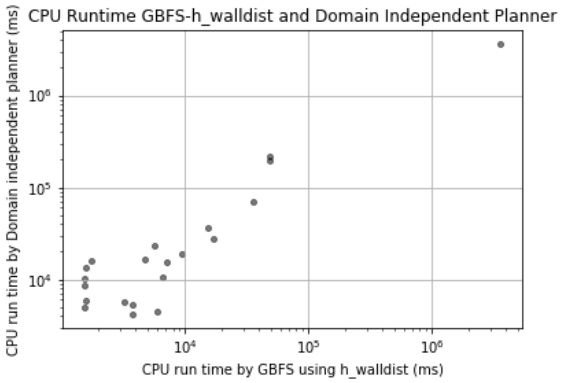

Figure 6: Runtime -Domain independent planer and GBFS-h_walldist

0.6.4 Comparison 2 GBFS-h_esdist Vs Domain Independent Planner

Problem Name	GBFS-Node Using h_esdist	GBFS-CPU Runtime Using h_esdist (ms)	Domain-Independent Planner Node	Domain Independent Planner CPU Runtime (ms)
Rect20	6	6469	8711	8711
Rect20a	9	8285	5780	5780
Rect20b	9	5712	5925	5925
Rect20c	9	945	4990	4990
Rect20d	9	6522	13650	13650
Rect20e	9	6482	10115	10115
Rect50	14	1387	16156	16156
Wall8a	9	3923	5395	5395
Wall8b	125	6387	4245	4245
Lhook16	1302	3400	16783	16783
Rhook16a	192	2070	10438	10438
Rhook16b	285	3784	15505	15505
Spiral16	420	63000	19081	19081
Spiral24	1077	49469	68672	68672
Pdes30	18	1070	27660	27660
Pdes30b	56	3619	35857	35857
Rectwall8	91	3963	4502	4502
Rectwall16	1100	27070	23135	23135
Rectwall32	10658	3522756	214082	214082
Rectwall32a	8734	1951001	196021	196021
Twisty0	17673	3600000	3600000	3600000

Table 2: Comparison GBFS "h_esdist" and Domain Independent Planner

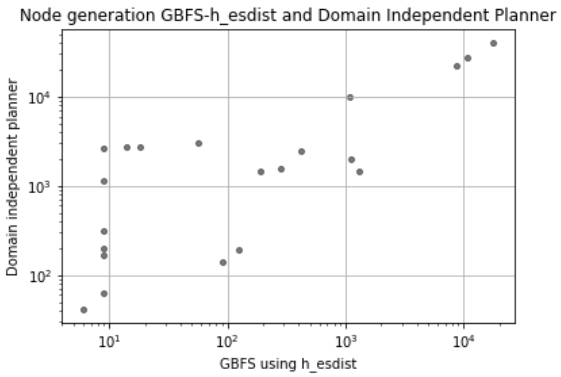

Figure 7: Node generated by Domain independent planer and GBFS-h_esdist

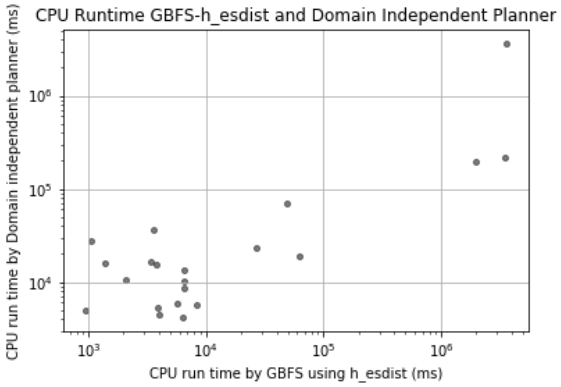

Figure 8: Runtime -Domain independent planer and GBFS-h_esdist

Form the above data it is observed in both comparison 1 and comparison 2 that domain independent planer produces more nodes as compared to domain dependent case. This can be related to the search algorithm and the heuristics. Domain independent planner uses AIBR and A*, while the domain dependent planner use GBFS and h_walldist or h_esdist. As we know that that A* performs exhaustive search and keeps a record of all the nodes expanded previously while exploring and thus has large number of nodes.However GBFS only keeps the track of the frontier nodes and removes all the previously expanded non-frontier nodes thus, has to deal with few nodes and thus has lower node number. Run time is also less for domain specific planners.

0.7 Results & Conclusion

From the above experiments it can be concluded that h_walldist performs better than h_esdist for problems it has seen before i.e problem with same environment but changing goal.Additionally, the CPU run-time and node formed are always more for domain independent planner than domain dependent.