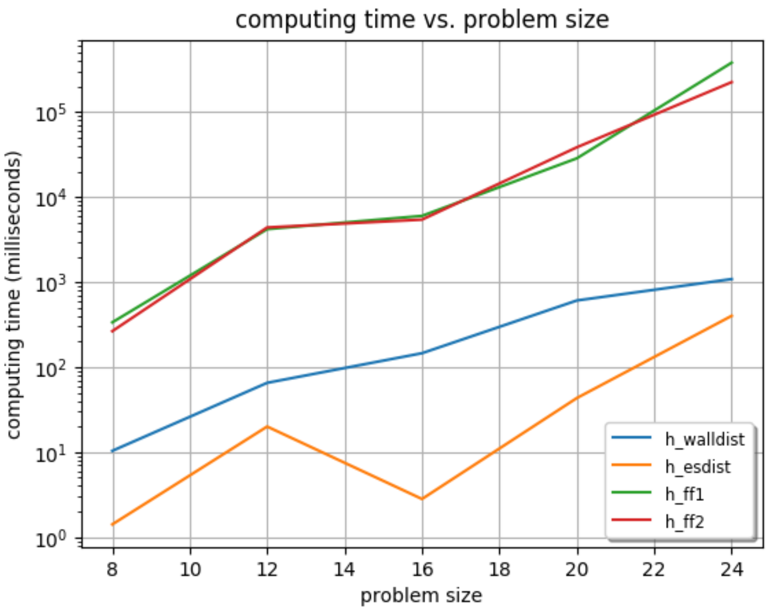
CMSC722 Project Part 1

Fan Yang

a) The computing time vs. problem size plot for each of the four heuristics is shown to the right. Each data point is averaged over 20 randomly generated problems.



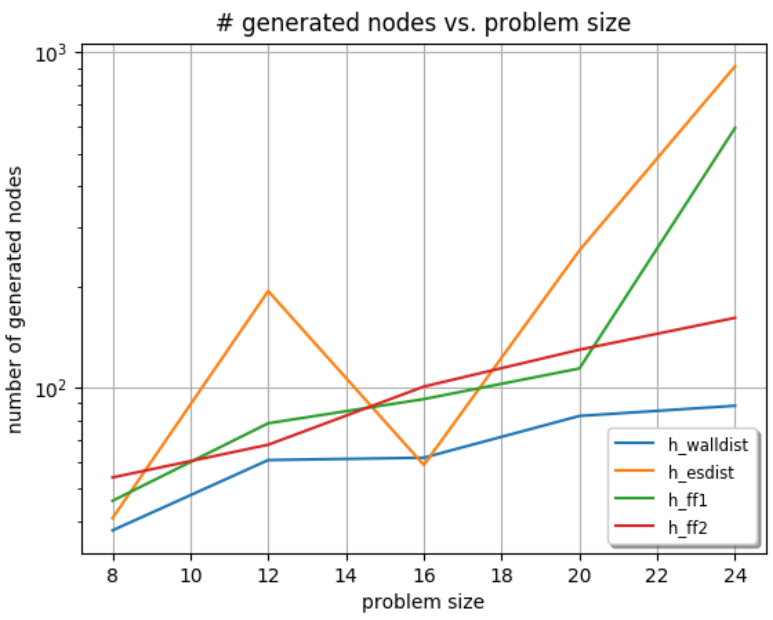
h_ff1 and h_ff2 take about the same amount of time, while h_walldist is roughly 100 times faster, and h_esdist is the fastest. This observation agrees with my expectations. The reason is explained below.

h_esdist only calculates the Euclidean distance plus an estimate of the number of steps needed to stop, so each call to h_esdist finishes in constant time. h_walldist does a bread-first search when it's called for the first time and it caches a rough estimate of the shortest path length to the finish line for each point. So the first call to h_walldist is expansive, but the subsequent calls are cheap because they simply retrieve the values from the cached table and add an estimate of the number of steps it takes to stop. Note that in the provided implementation of h_walldist, only the location variable is considered when doing the bread-first search, so the branching factor is at most 4 and the size of the state space is at most n^2 (n is the problem size). Thus, the first call to h_walldist costs at most $O(4n^2)$, which is quite acceptable considering that n is not large.

In contrast, h_ff1 and h_ff2 run much slower. This is because these two heuristics consider both the location variable(s) and the velocity variable(s) and really try to compute a solution (though relaxed) to the problem every time they are called. Considering the two while loops in the HFF algorithm: at every iteration of the first while loop, the algorithm checks all currently applicable actions and do the crash test and goal test as necessary. Since the number of applicable actions can be much larger than n^2 when the velocity variable(s) are taken into account, this while loop can cost much more than quadratic time. Similar analysis holds for the second while loop (i.e., the backtracking while loop). Therefore, every call to h_ff1 or h_ff2 can possibly take more than $O(n^2)$. We can improve the performance significantly by caching the explored states to avoid computing multiple times for the same state, but still the algorithm won't finish fast.

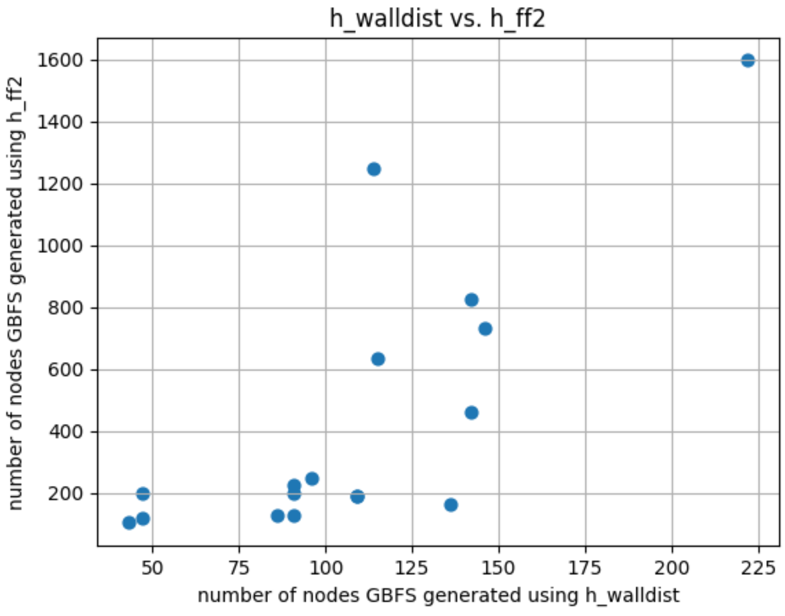
b) Roughly speaking, h_esdist generates the most number of nodes while h_walldist generates the fewest; h_ff1 and h_ff2 generates about the same number except for problem size = 24.

As discussed in (a), h_walldist returns the length of the shortest path to the finish line plus an estimate of the number of steps needed to stop. Though not admissible, this heuristic turns out to be a good estimate and works efficiently and consistently (with respect to problems of varying structures). In comparison, the performance of h_esdist depends significantly on the problem structure, since it only returns an estimate of the Euclidean distance to the finish line. This is to say that it can underestimate by large if there exist a wall between the current position and the finish line, and this is probably the reason why it generated so many nodes when problem size = 24 (some of the 20 randomly generated problems may have walls between the starting points and the finish lines).



We observe that h_ff1 also performed bad when problem size = 24. This can possibly be explained from the same perspective as discussed above. We used state-variable representation 1 for h_ff1. A relaxed state in that representation can give many combinations of the values of the four state variables x, y, u, v, and each of the combinations is considered valid for planning and goal test. Therefore, h_ff1 converges fast but tends to underestimate, especially when the problem structure is complex. Hence, if some of the problems of size = 24 have walls between the starting points and the finish lines, h_ff1 will frequently return very inaccurate estimates which leads to GBFS searching in the wrong direction and generating more nodes than necessary. In comparison, h_ff2 converges much slower and is less likely to underestimate (or will underestimate less) in the case of complex problems because it uses state-variable representation 2. Consequently, we see that h_ff2 behaves more consistently and generates far fewer nodes than h_ff1 does when problem size = 24.

c) The scatter plot is to the right.



The numbers of nodes generated by these two heuristics are roughly linearly proportional (with a large variance), and that h_ff2 always generates more nodes than does h_walldist. A possible explanation for this observation is that there is some randomness in the value returned by h_ff2. That is to say, the number of steps calculated by h_ff2 depends to some extent on the random choice of which minimal set of actions to take when backtracking from the goal to the starting point. This randomness in the returned value can cause GBFS to run off its current path frequently (even when the current path is the most promising one) and spend a lot of time, more than necessary, to explore the nearby region. In comparison, h_walldist does the calculations much more consistently, thus much less time is spent on the unnecessary swaying.

d) Comparing column 2 and column 3 of the below table we see that the number of nodes generated by h_ff2 is usually 2 to 5 times as many as generated by h_walldist. There are two conspicuouse exceptions. For lhook16 and spiral24, h_ff2 generates roughly 10 time more nodes than does h_walldist. It's worth noting that lhook16 and spiral24 are arguably the two most complex problems among the given sample problems in view of their structures. This implies that h_ff2 might not be an efficient choice for complex problems, possibly because in that case there is more room for h_ff2 to sway around, as discussed in (c).

Problem name	# nodes GBFS generated using h_walldist	# nodes GBFS generated using h_ff2
wall8a	43	105
wall8b	47	202
rectwall8	47	120
rhook16a	142	460
rhook16b	142	827
spiral16	146	733
rectwall16	115	634
lhook16	114	1247
rect20a	109	190
rect20b	109	190
rect20c	91	199
rect20d	91	226
rect20e	91	127
spiral24	222	1596
pdes30	86	127
pdes30b	96	250
rect50	136	163