NGUYEN, Réal **SOEN 472 SEC F**
ID#27566263 **DUE 15/10/2018**

MINI-PROJECT 1 REPORT

Introduction

The purpose of this report is to analyse the performance and efficiency based on the results of analysing uninformed and informed search algorithms, the informed algorithms' use of different heuristics on a variation of the classic sliding 8-puzzle, with a dimension of 3x4 (column x row) and with legal diagonal moves.

First, I will compare the use of the different heuristics (or lack thereof) in the search algorithms; second, I will discuss the difficulties I ran into while developing and testing the program; third, I will give details on various experiments I integrated into the program to get different results; and finally, I will conclude the report with short insights on what I have learned from this mini-project.

Heuristics

The two main heuristics I have used in this mini-project are the hamming distance (h_1, referred to in the program as "tiles out of place"), and the Chebyshev distance (h_2). I have also programmed in a third heuristic, the sum of permutation inversions, but this is part of my experimentations, and will be detailed in the "Experiments" section of this report.

First, the hamming distance is a very simple heuristic that counts the number of tiles that are out of place compared to its goal state, not counting the empty tile. For example, compared to the goal state [1 2 3 4 5 6 7 8 9 10 11 0], the hamming distance of state [1 2 3 4 5 6 7 0 9 10 11 8] would be 1, because only the 8 tile is not in its goal spot.

Second, the Chebyshev distance is related to the Manhattan distance we have seen in class. Like the Manhattan distance, for each tile, it counts the number of moves this tile needs to reach its goal state, not counting the empty tile. For example, the puzzle included in the mini-project handout [1 0 3 7 5 2 6 4 9 10 11 8] would have a Chebyshev distance of 5:

$d_1 + d_3 + d_7 + d_5 + d_2 + d_6 + d_4 + d_9 + d_{10} + d_{11} + d_8 =$
$0 + 0 + 1 + 1 + 1 + 0 + 1 + 0 + 0 + 0 + 1 = 5$

I could not use the Manhattan distance, because it assumes that diagonal moves are illegal, but in our case, they are legal. Thus, using the Manhattan distance would overestimate the distance between each tile and their goal spot. For example, the nearly solved puzzle [1 2 3 4 5 6 0 8 9 10 11 7] would have a Chebyshev distance of 1 (one down-right diagonal move) but a Manhattan distance of 2 (one move right, then one move down).

In the following few analytic paragraphs, I have run both best-first search (BFS) and algorithm A* (A*) with both heuristics. The 10 puzzles being solved are all randomly generated from https://random.org/integer-sets. All the puzzles are solvable, and when I write that a puzzle is "unsolved," I mean that I had to terminate the program manually after not getting a solution within 5 minutes. Please check the input file for the puzzles used or the 472_mp1_analysis spreadsheet for more details.

For BFS, using h_1 is already fast enough (average run time of about 0.45 seconds, median run time of 0.124 seconds). Even then, using h_2 shows far superior performance and efficiency.

While not easily perceptible, using h_2 cuts down the average run time to 0.0098 seconds and the median run time to 0.0085 seconds. This is due to its search efficiency vs. h_1. Compare h_1's average and median search path sizes, 1214.7 and 940.5, respectively; to h_2's 116.6 and 96. In addition, the average and median solution path sizes are also decreased, by almost half: 68 and 59.5 for h_1 vs. 37.3 and 37.5 for h_2. In other words, h_2 is more informed than h_1.

For A*, the difference between the choice of heuristics is even more pronounced. Not only is it the difference between searching thousands of states vs. searching tens of thousands of states, it's the difference between minutes of run time vs. milliseconds of run time. This seems to be because A* is much more prone to being stuck in plateaus compared to BFS.

With h_1, A* simply cannot solve any puzzle within a reasonable timeframe. All the puzzles take at least 5 minutes of runtime, but I terminate the program by then, so the exact performance and efficiency statistics are unknown.

With h_2, on the other hand, all except one puzzle are solved within 5 minutes, with half of them being under 2 seconds of run time. The remaining puzzle took over 22 minutes to solve, and while it has been included in the analysis spreadsheet, it will not be included in the statistics in the following paragraphs for the sake of consistency.

However, compared to BFS, which is extremely consistent in its run times, A*'s very wildly: the lowest run time is 0.072 seconds and the highest is 58.219 seconds (not counting the unsolved puzzle). Moreover, the search path sizes are orders of magnitude larger than BFS, both with h_1 and h_2: an average of around 7761 states searched and a median of 5104.

On another note, while A* is noticeably slower than BFS, the solution path sizes are quite shorter. Assuming h_2 is used, BFS has an average solution path size of 37.3 and a median of 37.5. Compare with A*'s average of 21.7 and median of 22.

Finally, it is also important to speak about depth-first search (DFS) because of its lack of heuristics. Predictably, its performance and efficiency are abysmal. Even for a nearly solved puzzle that requires one move to solve, it goes down a branch that goes nowhere and tries to find a solution. This of course also applies to randomly generated puzzles, meaning that all puzzles (except an already solved one) given to DFS will never be solved within a reasonable timeframe.

Difficulties

The bulk of the difficulties were in the development of DFS, because the search algorithm and the tree data structures would be reused throughout the program. However, once that was done, developing BFS and A* was only a matter of tweaking DFS to fit with a priority queue and heuristics.

Testing brought about grievances more than difficulties. Initially, all three algorithms used recursion to search. That is decent enough while developing, because the algorithms would cause a stack overflow within seconds if it could not find a solution quickly, but that convenience was also its downfall – it was impossible to get performance and efficiency statistics. I could also not figure out how to make the program give up on the search without having it throw a stack overflow error (when searching recursively), or manually terminating the program (when searching iteratively). This made testing extremely tedious, as I could not know how long the algorithm would run for when giving them randomly generated puzzles.

Another difficulty was figuring out how/if my methodologies and heuristics were effective or not.

For methodologies, I gave the program many puzzles, but I could not figure out if the puzzles were solvable or not, because all the proofs that exist do not consider diagonal moves. Fortunately, I later found out that BFS solved every single puzzle I gave it.

For heuristics, it was a matter of finding out if they were admissible or not. To check, I tried running breadth-first search, because it is known to be admissible. It does work for puzzles with short solution paths, like the example given in the handout. However, it proved to be as ineffective as DFS in terms of performance when it came to randomly generated puzzles.

Experiments

For the experimentation part, I have tried three things: different sizes, a different heuristic, and implementing breadth-first search. Please note that for this section only, best-first search will be abbreviated BeFS and breadth-first search will be abbreviated BrFS.

I have experimented with different puzzle sizes, 3x3 and 4x4. I chose these sizes because 3x3 is more easily testable because of its smaller state space, especially for running uninformed search, and I chose 4x4 simply I was curious about the solvability of the famous 15-puzzle we saw in class if we were to introduce diagonal moves.

For 3x3 puzzles the comparisons between h_1 and h_2 for BeFS and A* are similar: h_2 is significantly more efficient than h_1 in its search, and thus has better performance. While h_1 is still relatively inefficient for A* on 3x3 puzzles, the puzzles are now at least solved, compared to the results I had with 3x4 puzzles. It is interesting to note that A* returns the same solution path sizes for both heuristics.

As for the classic 4x4 puzzle [1 2 3 4 5 6 7 8 9 10 11 12 13 15 14 0], it is indeed solvable with diagonal moves.

I tried implementing another heuristic: the sum of permutation inversions (h_x). This heuristic was introduced as the “best” heuristic for the 8-puzzle without diagonal moves (according to the slides), but surprisingly, it does not run very well in the 3x4 puzzle with diagonal moves. My hypothesis is that it is good for the classic 8-puzzle configuration precisely because diagonal moves are not allowed, and the sum can be used to prove whether the puzzle is solvable, but this point is obviously moot for our application. The proof of this hypothesis is outside of the scope of this report.

h_x is not quite as bad as h_1, but far worse than h_2. Notably, when comparing h_x with h_2, h_x’s solution path sizes (for the puzzles it could solve) can be almost twice as long, which may indicate that h_x is not admissible. This was unexpected, but a good experiment nonetheless.

I tried implementing BrFS because it is known to be optimal, that is, it will always return the shortest solution path. This is to check that h_1 and h_2 are admissible, and by extension check that algorithm A* is indeed A*, and not algorithm A. However, this was only feasible with 3x3 puzzles, because the state space for 3x4 puzzles is too large. Even then, the run time for any puzzle with a solution path longer than 9 (assuming A* with h_2) was consistently longer than 5 minutes each. Ultimately, I could only get solution paths for 5 out of the 10 puzzles I gave BrFS.

Tying it all together, I tested the 15-puzzle with all the search algorithms and all heuristics. Predictably, DFS could not solve the puzzle. BrFS did solve it and returned the shortest solution path size of 6. BeFS was very fast as always, but only h_2 had the shortest path. A* had the shortest path size for both h_1 and h_2, proving that they are admissible, but not for h_x, proving that it is not admissible.

Conclusion

The main purpose of this mini-project was the differences in performance and efficiency of both uninformed and informed search algorithms. This report writes in detail about the difficulties I ran into, that data I collected while analysing algorithm performance, and experiments I ran to get different results.

DFS is terribly inefficient in our current application and without fail could not solve anything.

BFS is very fast, but not optimal. The choice of heuristics does influence performance and efficiency but is not easily perceptible unless data is collected.

For A*, however, the choice of heuristics is crucial – it is the difference between solving a puzzle in minutes or solving it in milliseconds. A*'s search paths are orders of magnitude larger than BFS' but given a good heuristic, it will always return the shortest path.

In terms of difficulties, most of them were in the implementation – I had to write a good code base that would be easily tweaked for experimentations. Apart from that, data collection for performance analysis was quite tedious.

The experiment that was the most eye-opening was implementing the 15-puzzle we saw in class: not only does our configuration manage to solve it because of diagonal moves, it neatly compiled everything I learned from this mini-project: heuristic fit (especially for A*) is crucial, and just because a heuristic is good for one application, does not mean it is a good fit for another.

References

Each reference is also linked in the code where it is used.

https://stackoverflow.com/questions/2885173/how-do-i-create-a-file-and-write-to-it-in-java

https://www.baeldung.com/java-measure-elapsed-time

https://github.com/gferrer/8-Puzzle-Solver

https://en.wikipedia.org/wiki/Chebyshev_distance

https://www.geeksforgeeks.org/overriding-equals-method-in-java/

https://stackoverflow.com/questions/731365/reading-and-displaying-data-from-a-txt-file

https://stackoverflow.com/questions/409784/whats-the-simplest-way-to-print-a-java-array