Analysis:

As expected, breadth-first Search is a lot slower than Astarh1 which in turn is a lot slower than Astarh2. For “simple” puzzles, all three algorithms take about the same time. By simple, I mean a puzzle in which the number of displaced tiles is very small. Consider the puzzle in row major form, “1 2 3 5 4 6 8 7 0”. This puzzle has a parity zero hence there are only four displaced pieces. Namely – 4,5,7,8. I’m not considering 0 to be displaced. For this initial configuration, breadth-first search finds a solution in 0.033 seconds and expands 3395 states. Astarh1 takes 0.019 seconds and expands 327 states – a huge improvement! For the same configuration, Astarh2 takes 0.014 seconds and expands 155 nodes – an improvement on Astarh1 even! What I find truly interesting is that for simple puzzles I don’t observe a correlation between compilation time and the number of moves to the goal state. For the above-mentioned initial configuration, it takes 14 moves to achieve the goal state. Yet, this doesn’t slow down any of the search algorithms. I wasn’t expecting any of the Astar algorithms to slow down (because they “strategically” traverse and expand the graph) but it’s awesome that BFS doesn’t slow down either. From my test cases, in more complex initial configurations, there exists a direct correlation between number of moves to the goal state and compilation time needed.

The analysis gets far more interesting for more complex (possibly randomly generated) initial configurations. For example, consider the puzzle (listed in row major order) “8 7 3 1 0 6 4 2 5” which was an actual randomly generated initial configuration. Breadth-first Search took about 4.3 minutes to find the solution and expanded 867351 nodes! Compare that to Astarh1 which took only 0.2 minutes or about 12 seconds to find a solution and expanded 67281 nodes. Better than BFS but still not the best. Now consider Astarh2 – which uses the Manhattan distances as the heuristic function. For the same configuration, Astarh2 only takes 0.139 seconds to find a solution and expands only 7945 nodes! The number of moves to the goal state for the above initial configuration is 24. Now you can see why I draw a correlation between compilation time and number of moves to the goal state for complex initial configurations. For a “simple” puzzle that took only 14 moves to the goal state, BFS compiled and found the solution in 0.033 seconds. For a more complex puzzle as the one above, BFS took 4.3 minutes to find a solution which required 24 moves. Even Astarh1 and Astarh2 were considerably slower than in the case for the “simple” puzzle. My hunch as to the reason behind breadth-first search being so slow is the huge amount of memory the algorithm requires – particularly in maintaining the explored set. The above puzzle required 867351 states of the puzzle to be stored in memory as the explored set. Even with modern CPUs, managing and manipulating that much data is a huge task. Astarh1 is a lot better. It uses the number of misplaced tiles as a heuristic and the sum of the misplaced tiles and depth as its evaluation function. Thus, Astarh1 strategically pops nodes from the frontier. In doing so it explores a lot fewer nodes than breadth-first search – 67281 to be exact for the above initial configuration. Astarh2 is pretty similar to Astarh1 but uses a better heuristic function than Astarh1 – it uses the Manhattan distance as a heuristic. Because it uses such a great heuristic function, when Astarh2 is popping nodes from the frontier, it continuously pops nodes that have the least evaluation function (depth + sum of Manhattan distance) and thus, pops less nodes than both Astarh1 and BFS in getting to the goal state.

It should be noted here that BFS occasionally takes far too long to find a solution. I’ve randomized an initial configuration and run the tester program for ten minutes and yet BFS found no solution. As an example, consider the initial configuration of, “7 8 4 6 3 0 5 2 1” in row major form. Astarh1 found the solution in 14.9 seconds expanding 74024 nodes and Astarh2 found the solution 0.04 seconds and expanded only 2033 nodes. BFS on the other hand was left to compile for 10 minutes and couldn’t find a solution. Both Astarh1 and Astarh2 usually find a solution in a reasonable amount of time. There were very few test cases I ran in which Astarh1 and Astarh2 could not find a solution after running the tester programs for about five minutes. Even though that’s still a long time, it’s still a lot better than sitting around for hours on end waiting for BFS to find a solution.

Finally, there are a number of conclusions one could state from testing all three algorithms. Clearly for “simple” configurations, all three algorithms do pretty well – the size of the search tree is small and hence the explored set is easily managed and manipulated by the memory. In more complex cases, there is a stark difference between all three algorithms solely based on the number of search tree nodes each respective algorithm expands. Astarh1 and Astarh2 do better than BFS. In fewer words, one can say that as the initial puzzle become increasingly complex, the size of the search tree grows exponentially and as such, algorithms that “strategically” pop nodes(Astarh1 and Astarh2) do a lot better than algorithms that basically expand every node on a level by level basis continuously looking for a goal state because in the Astar algorithms memory is a lot easier to manage (the number of nodes to store are a lot less!).