Juan-Manuel Garcia del Muro Navarro

Artificial Intelligence based Tile Puzzle Solver

Code structure

The project has been built on top of the provided code. Python 2.7.X is necessary to run the code. I decided to create a new file for each search algorithm, in order to make the code easier to read and better structured. All of these files are stored in the folder search_algorithms. To choose a search algorithm, you simply have to add it as a third parameter when executing the project. As an example, the following command would run the A* search algorithm with the second heuristic:

```
python tilepuzzle.py 8 10 astar2
```

Note: The different heuristics have been implemented in the corresponding algorithm files as a function that returns the desired value.

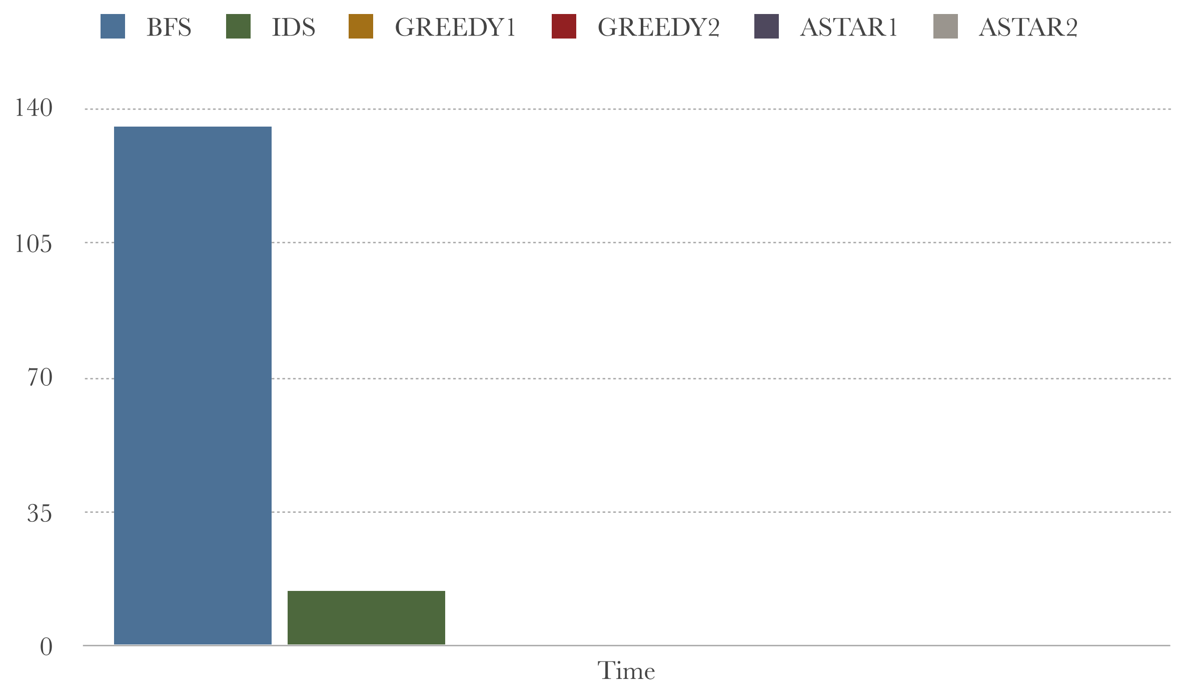
Benchmarking

To evaluate these search algorithms, we have taken as parameters the average and median of the time taken, average and median of the number of steps represented by the number of nodes expanded and the branching factor that each of them takes when solving a randomly generated 8x8 puzzle. As these values are greatly impacted by the initial configuration of the puzzle, I have ran each algorithm 20 times and calculated the average and the median for each of the parameters, which conforms the main part of the benchmarking. The full results are included in the file benchmark.pdf.

To get an idea of the efficiency of the algorithms solving this type of problems, we have started off generating a random initial puzzle and running all of our algorithms.

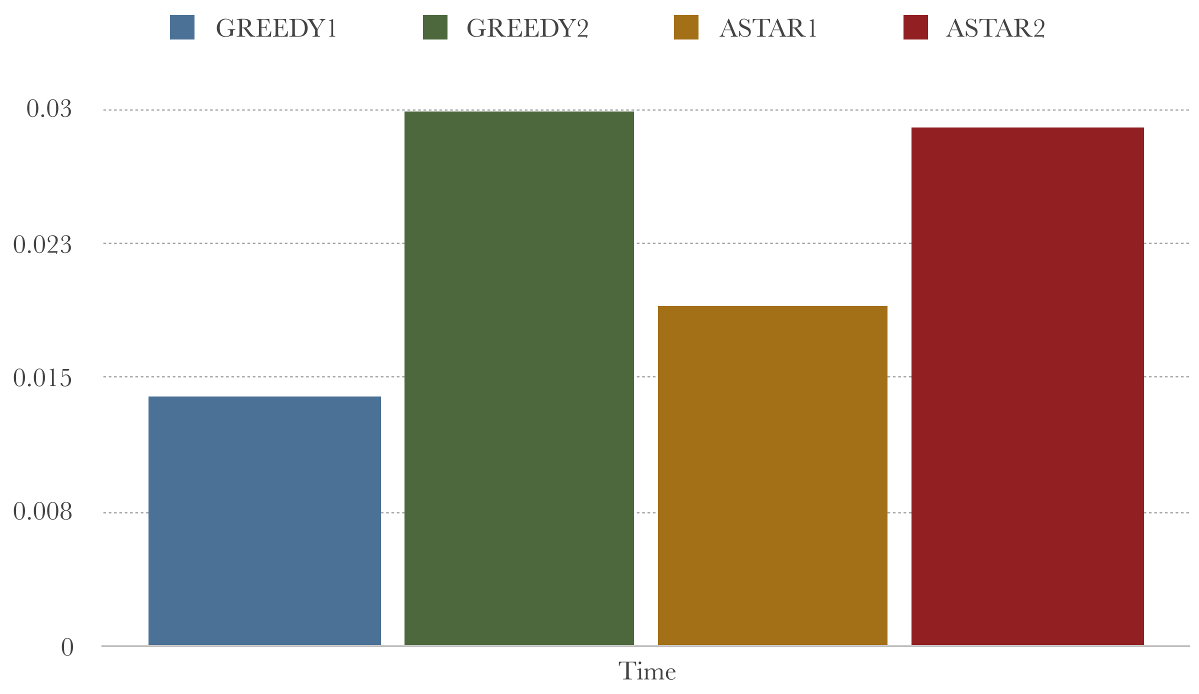
The way we generate these initial 8-puzzles is by performing either 10 or 20 random moves called permutations of the 0 tile in a solved puzzle. We perform 10 permutations to benchmark BFS and IDS, and 20 permutations to test the rest, as 10 initial permutations wasn't enough to fully appreciate the difference between them and the different heuristics. Starting from a solved puzzle, we make sure that it is solvable, as for many configurations there is no set of moves that reaches the goal.

Time taken (8x8 puzzle, 20 perm)

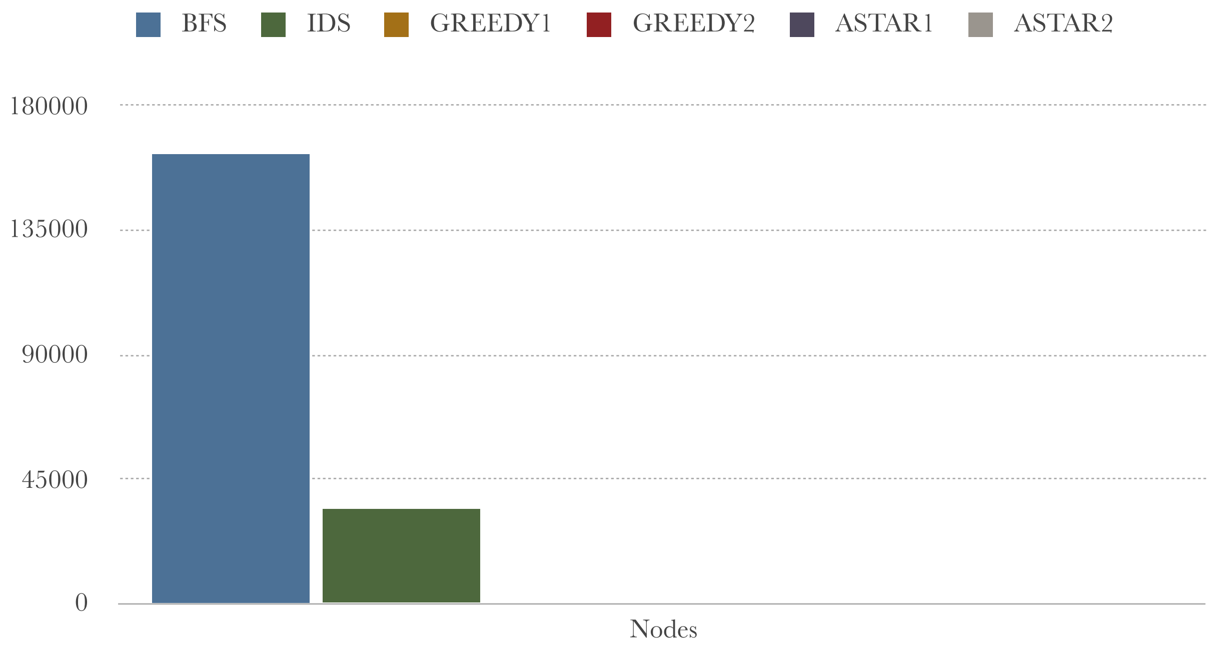


Note: As the times for the last 4 algorithms was below 0, they haven't been represented properly on the graph. In order to avoid this, we have cut them out in the next graph.

Time taken (8x8 puzzle, 20 perm, last 4 algorithms)

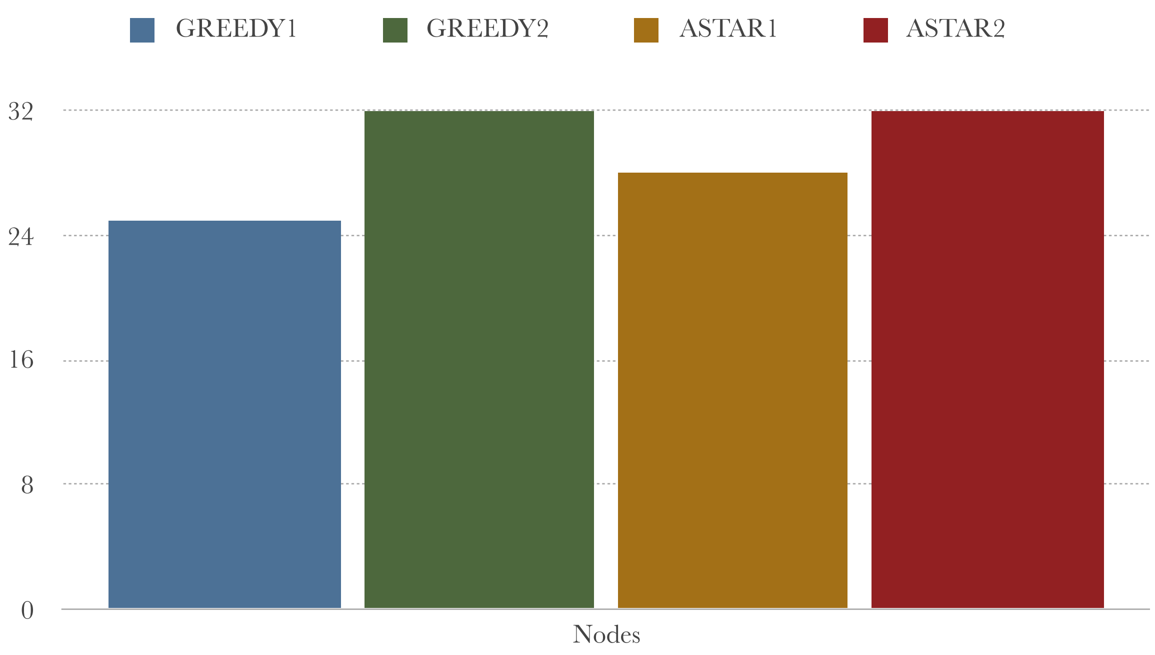


Expanded nodes (8x8 puzzle, 20 perm)

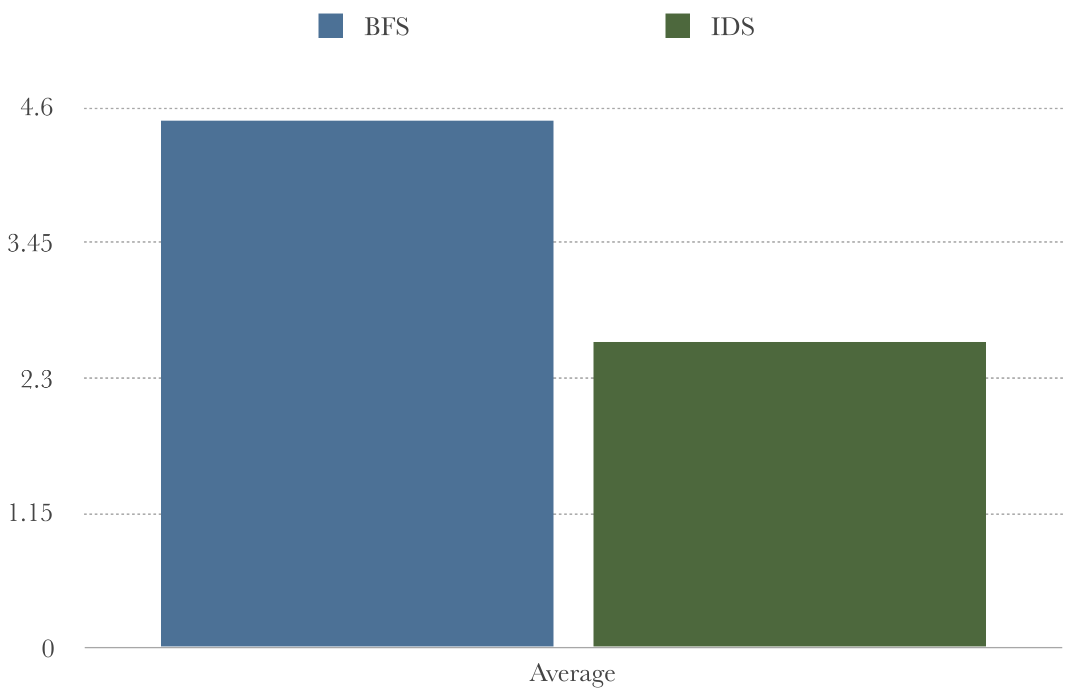


Note: For the same reason as before, we have cut the first two algorithms in the next graph.

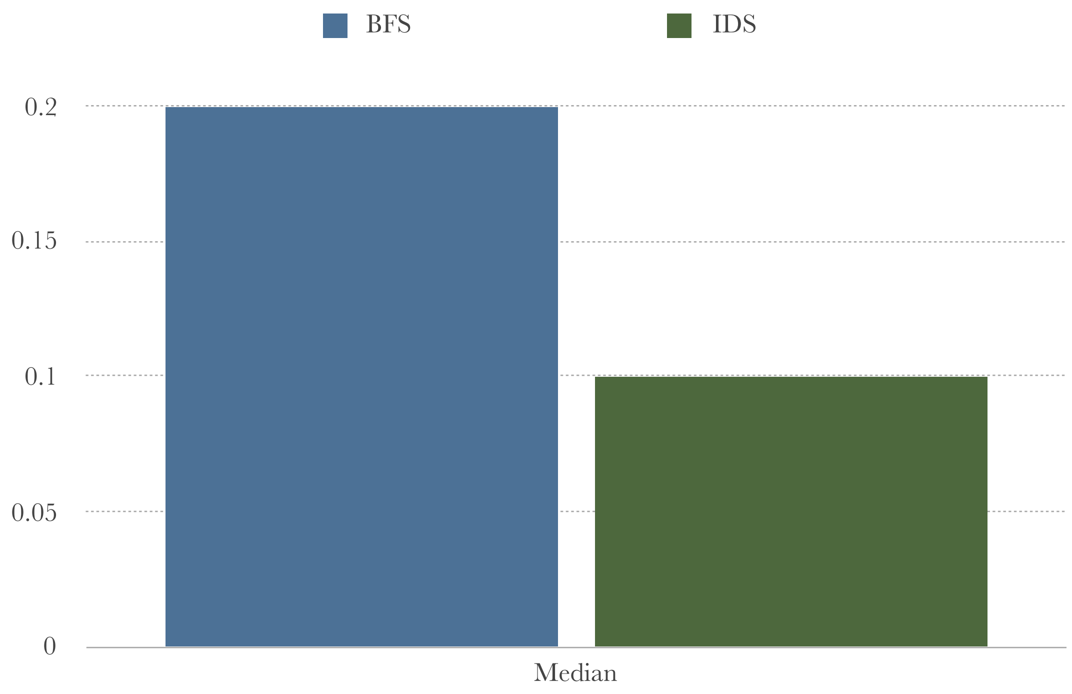
Expanded nodes (8x8 puzzle, 20 perm, last 4 algorithms)



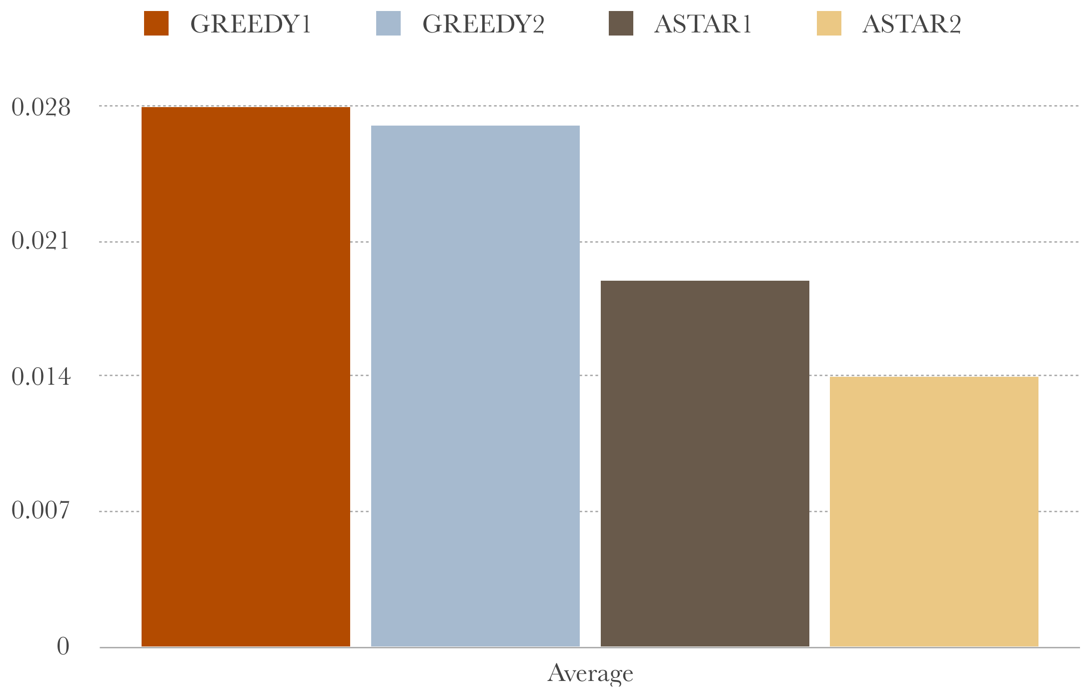
Average time taken (20 executions, 8x8 puzzle, 10 perm)



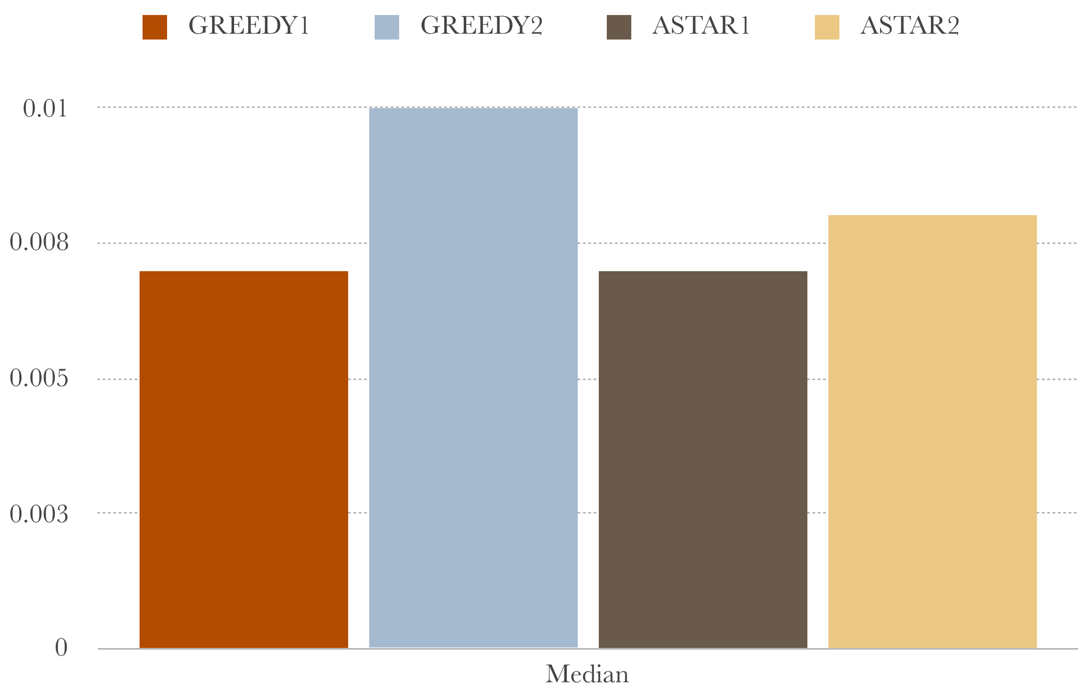
Median time taken (20 executions, 8x8 puzzle, 10 perm)



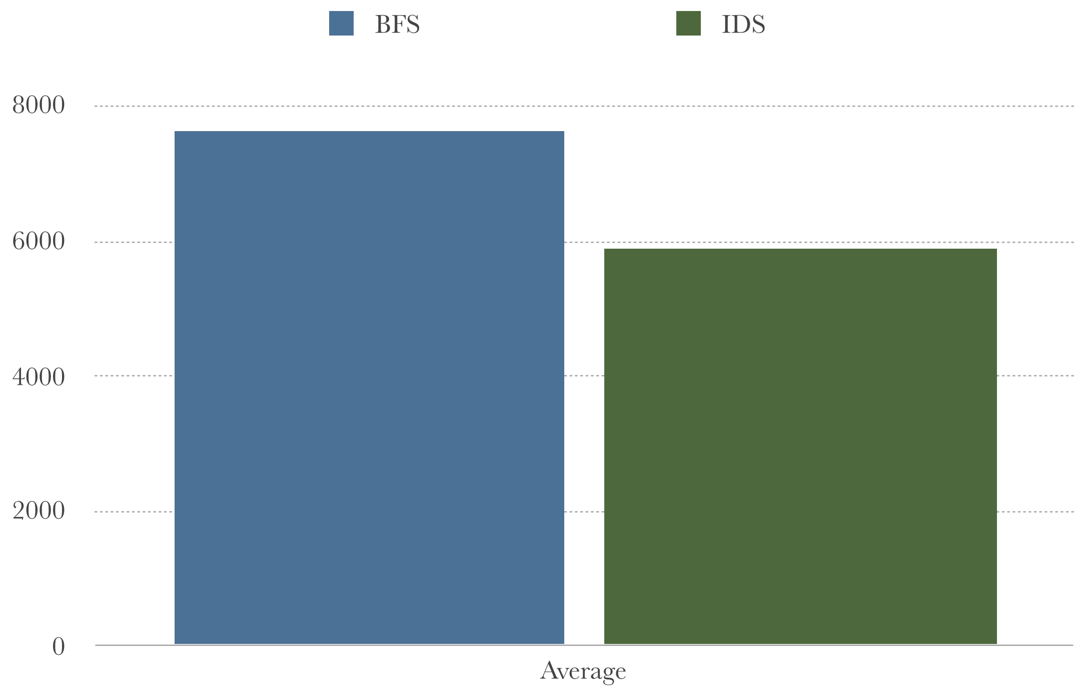
Average time taken (20 executions, 8x8 puzzle, 20 perm)



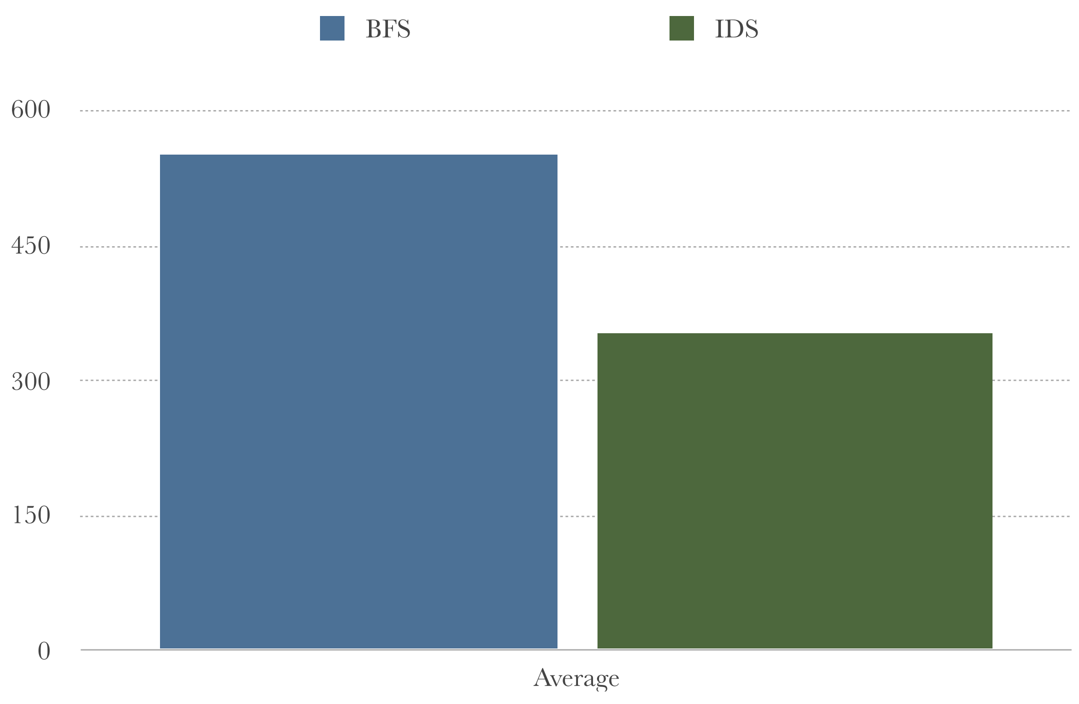
Median time taken (20 executions, 8x8 puzzle, 20 perm)



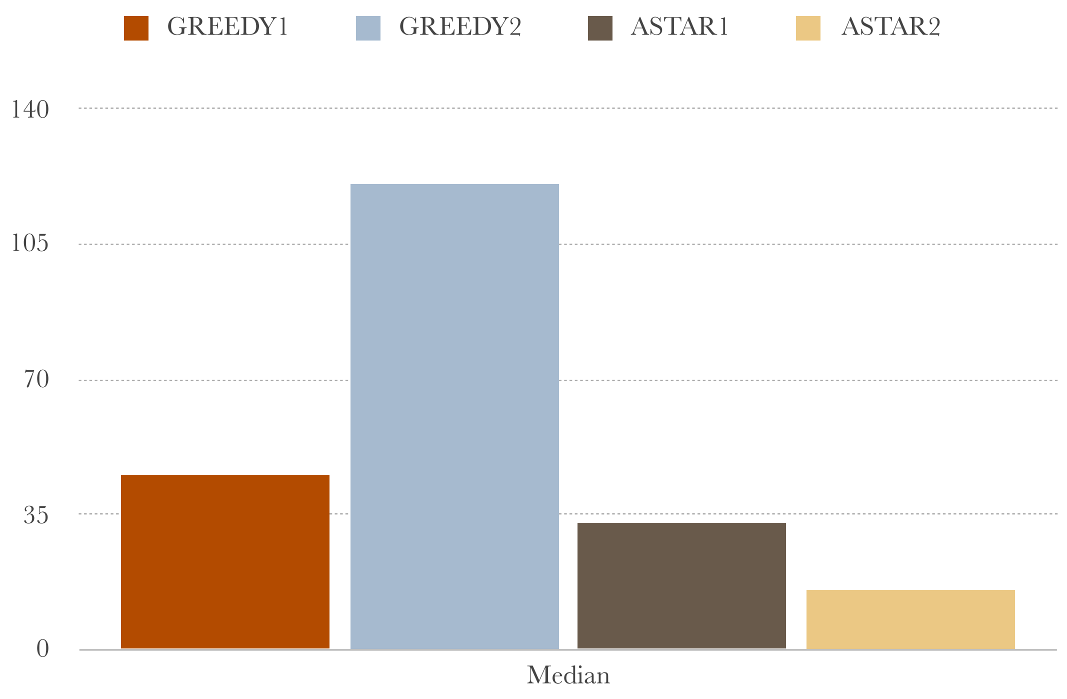
Average expanded nodes (20 executions, 8-puzzle, 10 perm)



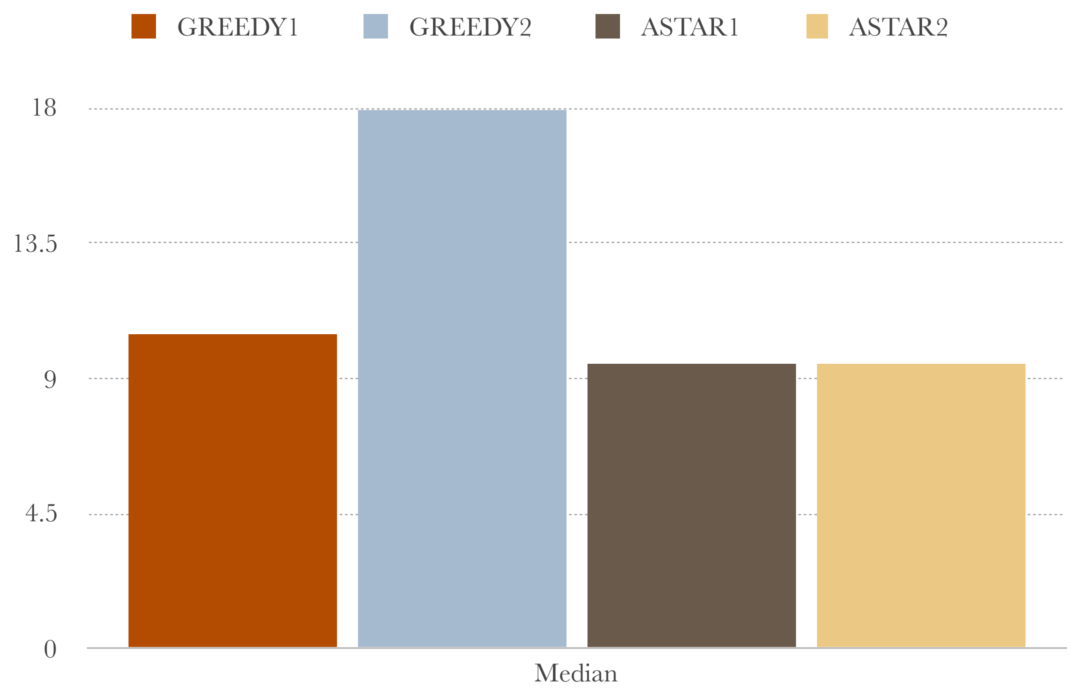
Median expanded nodes (20 executions, 8-puzzle, 10 perm)



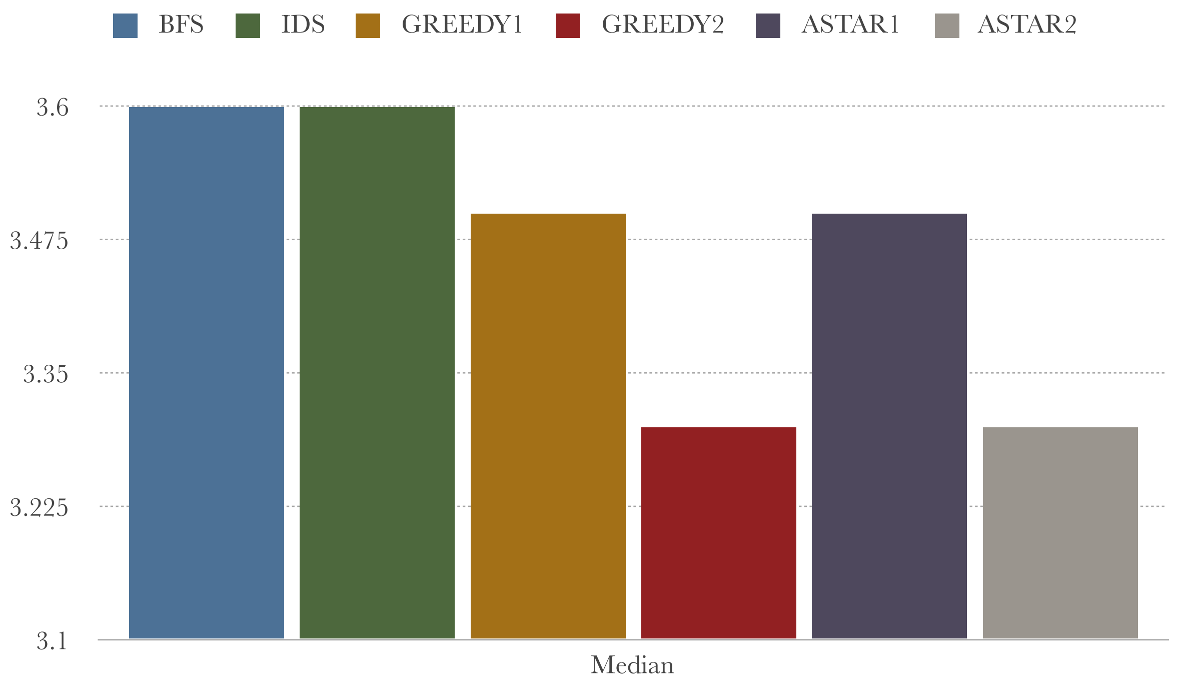
Average expanded nodes (20 executions, 8-puzzle, 20 perm)



Median expanded nodes (20 executions, 8-puzzle, 20 perm)



Average Branching factor (20 executions, 8-puzzle)



Conclusion

As we can clearly see in the benchmark section, **BFS** is not an optimal solution for this problem. For the initial 8x8 puzzle, it took over 2 minutes and expanded 162.599 nodes. That would only get worse for bigger puzzles, as the time and space complexity is terrible for this type of problems.

On the other hand, **IDS** is slightly better, as it doesn't expand the nodes in the same level of the solution. But that doesn't change the fact that it still took 14,2 seconds and expanded 33.943, which might seem like a reasonable solution after testing BFS, but its by no means an optimal algorithm for this problem.

To design an algorithm fit for this problem, we have to take into account the quality of the potential moves, in order to choose the best way possible to expand first. We do this through what is called the heuristic, which is a function that given a specific board configuration, it will return a value that represents how desirable that position is. We have chosen two algorithms and two posible heuristics: The Greedy algorithms and the A* algorithm, with one heuristic based on the amount of tiles in the incorrect places and the other based on the sum of the total distance between the tiles in the wrong places and their actual places.

Even tough in the first individual execution, **Greedy** search performs slightly better than the **A*** search, if we compare that to the 20 execution chart time taken and node expansion average and median, we can confidently say that the most efficient search algorithm is the **A*** and the best heuristic is the second one, the one that is based on the sum of the distances to the correct solutions. I consider the 20 executions average and median the best measurement to benchmark the performance of these algorithms, as an individual result can be impacted by a specific board configuration.

Limitations

We have already stated that the limitations of BFS and IDS are quite low, as they expand a very high number of nodes even for relatively small puzzles. The superiority of Greedy search and A* search is evident when solving bigger puzzles. For a 30x30 puzzle, applying 30 permutations, A* with the second heuristic manages to finish in under 1 second, expanding 50 nodes in average. For a 40x40 puzzle, it takes under 2 seconds and expands under 100 nodes. For a 50x50 puzzle, applying 50 permutations, it tends to finish under 3 seconds and expands under 150 nodes. If we compare this results to BFS and IDS, we can state that the execution complexity of **BFS** and **IDS** is close to **exponential execution complexity**, where as for **Greedy** and **A*** its close to **linear execution complexity**, being practically linear for **A*** with the **second heuristic**.