AI Assignment #2: A* search and the 8-puzzle

Harrison Downs | B01256060 | CSCI 6285

1: Analysis

1.1: Programming Choices

In order to create a controlled environment for testing both the A* search and the greedy search algorithms on the 8-puzzle problem, I implemented a class structure using Python. **`S`** is a class whose instances act both as representations of puzzle states and as nodes in a search tree. In each instance of S, the puzzle state is mapped into a 1-dimensional list, while its parent state and current depth are stored for use by a search algorithm. Additionally, **`S.expand()`**, **`S.create_state()`**, **`S.get_solution_path()`**, and **`S.h2()`** are used to find neighboring states, to create new states by combining the itself with an action, to generate a path from itself to the starting state, and to generate a heuristic value, respectively.

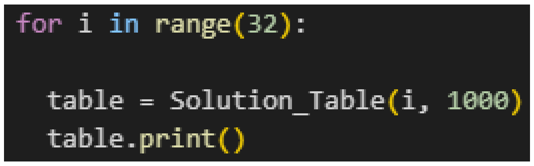
Moreover, **`Solution_Table`** is a class whose instances run search algorithms any given number of times, storing each run's starting state (as an **`S`** object), cost (nodes expanded), and solution depth in parallel lists. Once a **`Solution_Table`** object is initialized, it immediately runs each search algorithm **`num_times_to_run`** times, fully populating each list of data. Afterwards, it's capable of reporting average costs and depths unique to each search algorithm.

Everything else in my program exists as an independent function. For example, both **`a_star_search()`** and **`greedy_search()`** exist independently of any classes, taking in an S object representing the start state and returning another S object, representing the solution. Both functions use a priority queue to decide which state to expand next; **`a_star_search()`** combines the backward cost with the estimated forward cost, while **`greedy_search()`** only considers the estimated forward cost.

A critical component of this program is **`random_state(d)`**. This function outputs a random state of the board, which is **`d`** unique moves away from the goal state. In other words, it starts at the goal state and moves a tile **`d`** actions, careful to avoid any duplicate states. While my initial thought was that any optimal solution from any given state would have a depth of **`d`**, this was not the case, which I will explain in more detail later on.

1.2: Data Generation

`Solution_Table` in conjunction with **S** allows for the main program to easily call massive operations. In fact, only three lines of code were needed to generate all of the necessary data for this report.



```
for i in range(32):

  table = Solution_Table(i, 1000)
  table.print()
```

For **d** from 0-31, each search algorithm was given 1,000 randomized states from `random_state(d)`, resulting in ***64,000 search algorithm executions and 10,743,880 explored nodes***. For the sake of consistency, the same set of randomized states was used by both search algorithms. After each iteration of the above code, the average costs and solution depths were recorded, and serve as the basis of this study.

1.3: Interpretation of the Data

In the environment of the 8-puzzle, the only action that can be taken is to shift a tile in some direction. Whereas other problems may have actions with varying costs (such as the traveling salesman problem), in this particular case, all actions have a uniform cost. This means that the total cost of a search is equivalent to the number of nodes expanded.

As can be seen in Figure 1.3.1, A* and greedy search exhibit quite different resource costs. While **d** < 19, A* search is more cost optimal than greedy search, but that completely changes when **d** >= 19. The results suggest that the time/space complexity of the A* search is ***$O(n^2)$***, while greedy search seems to be closer to ***O(log(n))***.

Regarding average solution depth, greedy search never surpasses the performance of A* search, as reflected in Figure 1.3.2. Although both algorithms' average solution depth is tied from **d** = 0-6, greedy search's average solution depth begins to drastically increase when **d** > 6, while that of A* search stays relatively low. Both algorithms' curves suggest a logarithmic relationship to **d**.

Fig. 1.3.1 - Average Total Cost of Search Algorithms

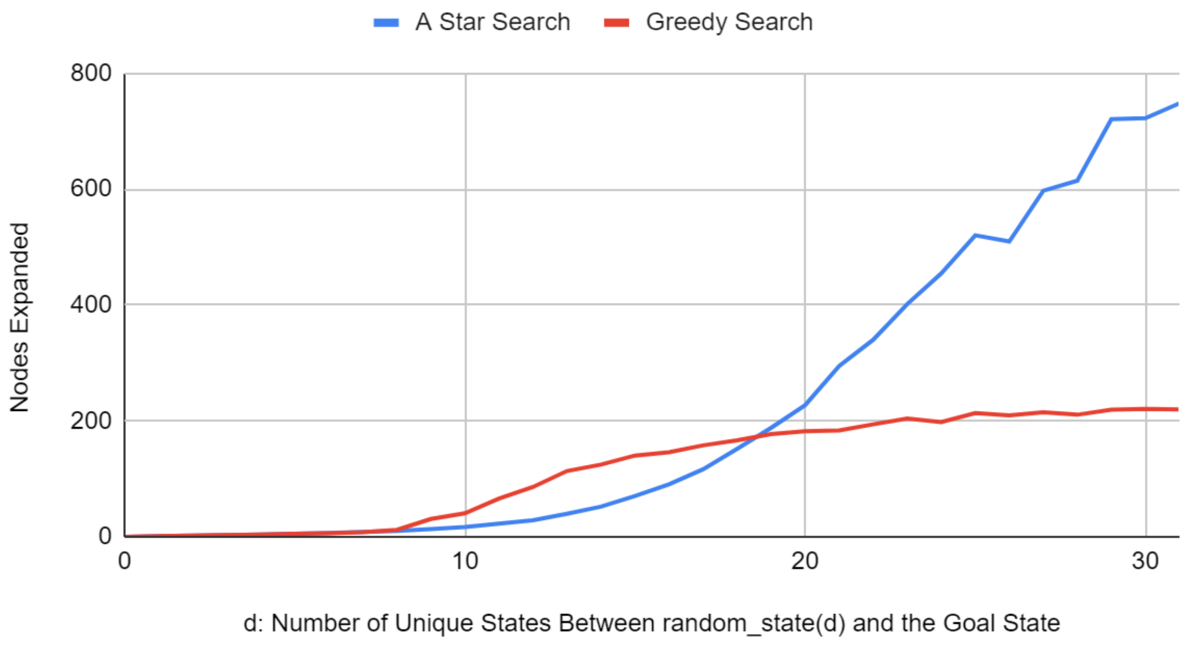
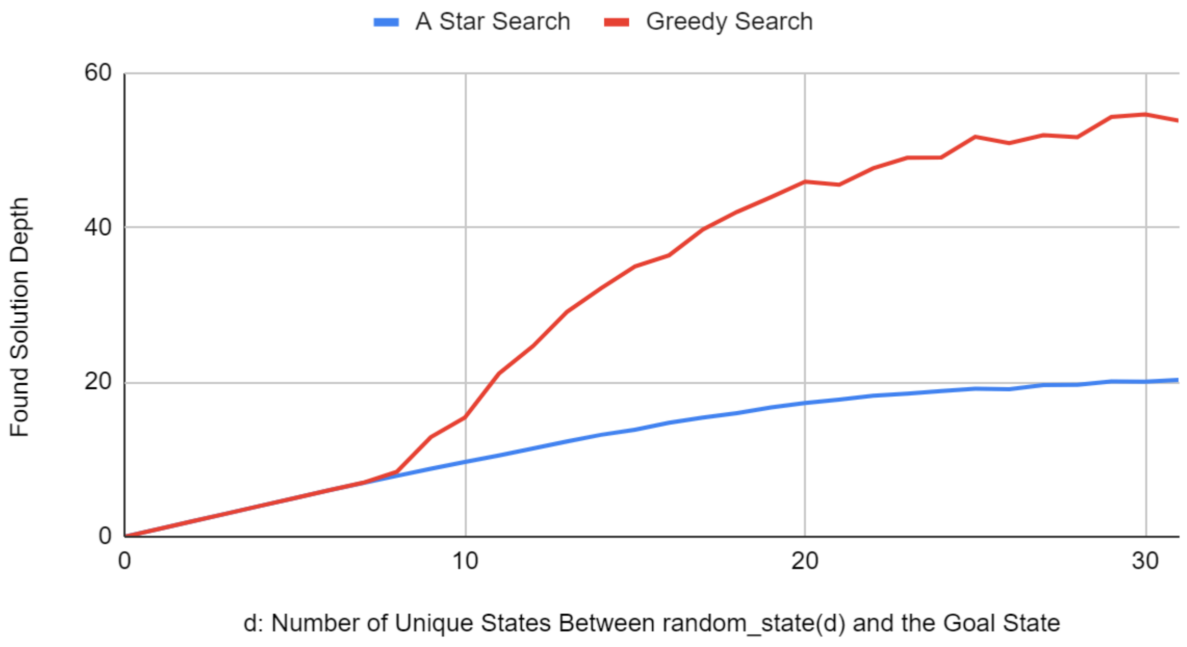


Fig. 1.3.2 - Average Solution Depth of Search Algorithms



Taking into account the patterns exhibited by both search algorithms, A* search excels in finding optimal solutions, with the disadvantage of requiring excessive resources. Conversely, greedy search prioritizes resource efficiency at the expense of finding non-optimal solutions. Therefore, if the puzzle needs to be solved in as few moves as possible, A* search is the best solution, whereas, if a solution needs to be found as *fast* as possible, greedy search is the prime candidate.

Interestingly enough, while A* and greedy search seem have an inverse relationship regarding resource efficiency and solution optimization, this relationship is not *truly* inverse. Each time **d** increases, the ratio between A* and greedy search's resource demand gradually increases. For notation purposes, let *C* indicate the average total cost of a search algorithm, and let *r()* show the ratio between to given *C* values. Additionally, subscript *a* represents A* search, and subscript *g* represents greedy search.

In Figure 1.3.3, $r(C_a, C_g)$ = 1 when **d** = 19, but $r(C_a, C_g)$ = 2 when **d** = 24, and $r(C_a, C_g)$ = 3 when **d** = 29. In other words, $r(C_a, C_g)$ is increasing at a steady rate of 0.2**d**. Ultimately, this pattern establishes that C_a will continue to exponentially grow as **d** increases.

However, the relationship is quite different regarding average solution depth. Building on the current notation, let *S* represent the average solution depth of a search algorithm. In Figure 1.3.4, $r(S_g, S_a)$ gradually increases when **d** = 0-17, whereas it plateaus at about 2.6 when **d** >= 18. As previously mentioned, both algorithms exhibit a logarithmic relationship between average solution depth and **d**, so if this pattern continues, any solution found by greedy search would only be approximately 2.6 times longer than the solution found by A* search.

Ultimately, the difference in *growth* between cost and depth complicates the decision between using A* or greedy search in the 8-puzzle problem. While A* search's proficiency in finding optimal solutions and greedy search's resource optimization remain established, it should certainly be considered that greedy search's solution depth seems to scale much better than A* search's resource demand.

Fig. 1.3.3 - Ratio of A* Search to Greedy Search Avg. Cost

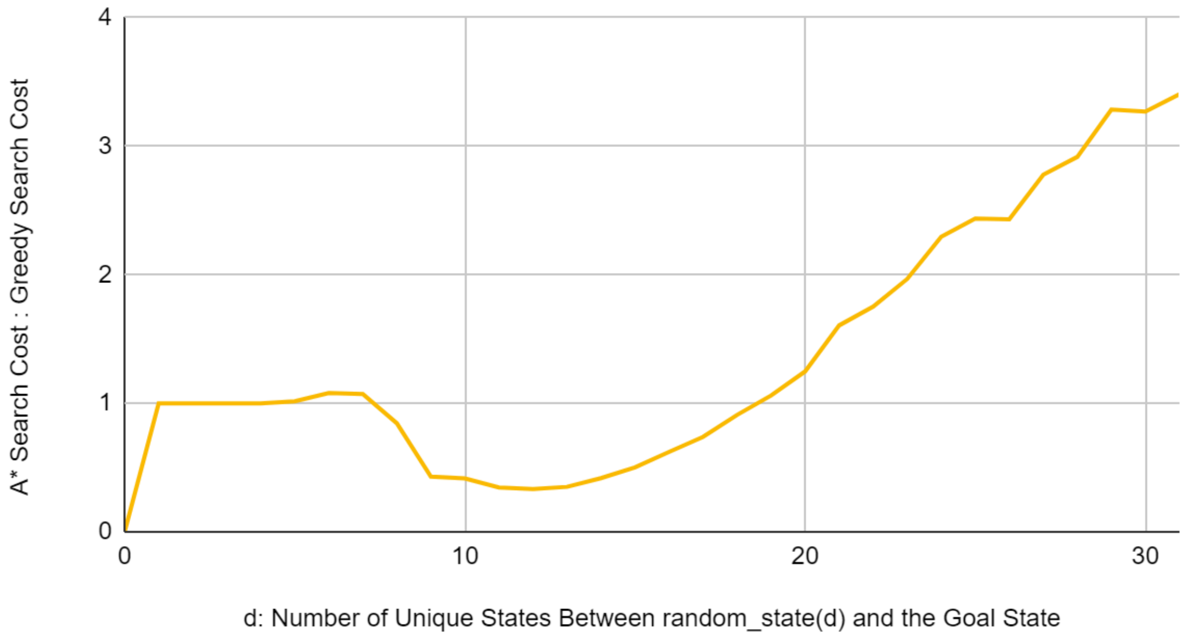
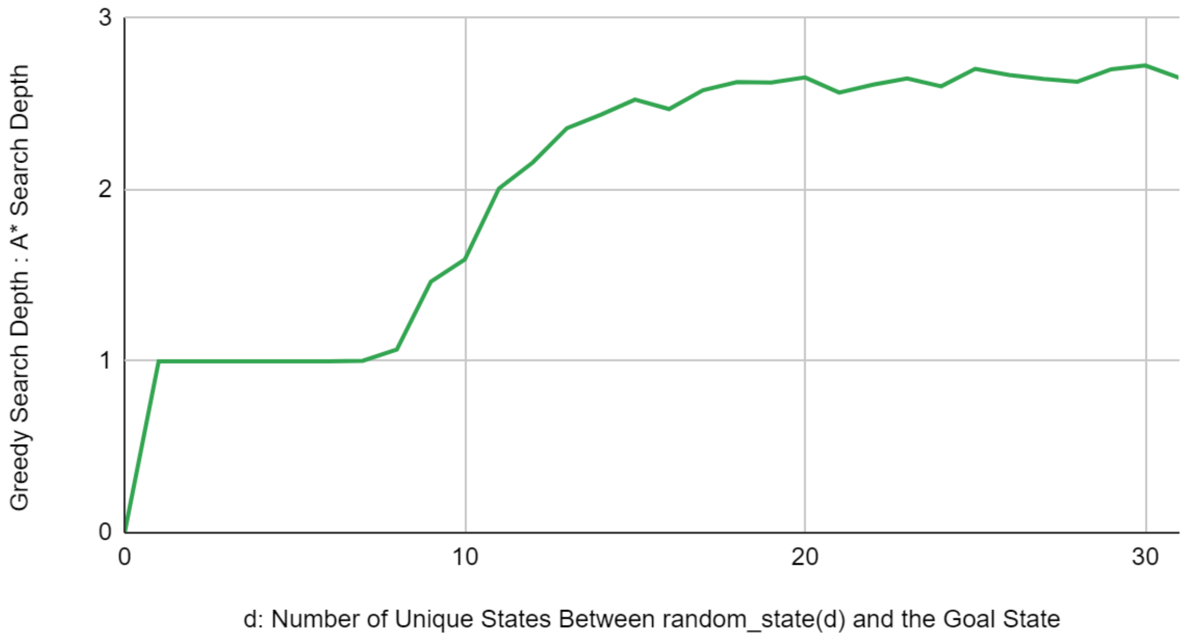


Fig. 1.3.4 - Ratio of Greedy Search to A* Search Avg. Depth



2: Limitations

The findings of this report have been affected by some limitations. Firstly, **`random_state()`** did not behave in the way it was initially intended. My initial thought when making this function was that, for any state generated by **`random_state(d)`**, **`d`** would represent its optimal solution depth. This was due to the fact that **`random_state()`** carefully avoids visiting states more than once . However, A* search repeatedly found solutions whose depth was less than **`d`**. In retrospect, **`random_state()`** never had any guarantee of generating a state whose optimal path was equal to **`d`**; **`d`** only served as a measure of the complexity of the starting state. For example, if x = **`random_state(5)`** and y = **`random_state(10)`**, $S_a(x) \approx 5$, while $S_a(y) \approx 9.6$.

The true effect that **`d`** has on determining the optimal solution depth of a given state should be further investigated. Furthermore, more research could be done utilizing different means of generating random states. For example, states could be visited multiple times in the randomization process, and the effects could be different on the end results.

The scope of **`d`** is limited to 0-31 in this report, so further research could be done on larger ranges of d to see if the patterns exhibited in this report persist over a larger scope. Additionally, seeing as a sample size of 1000 was used for each search algorithm for each value of **`d`** from 0-31, using a higher sample size could lead to more accurate findings.

3: Reflection

As I reflect about this project, the first thing that comes to mind is that this process served to hone my skills with Python. This past year, I've been a post-baccalaureate student taking Computer Science I & II along with Data Structures, all of which used C++ as the programming language. While it was difficult to implement my class structure using a language I'm not familiar with, I found that many concepts from C++ naturally translated to Python. Although it was a challenge, as long as I had Python documentation at my disposal, I was able to write the program relatively quickly.

Additionally, figuring out how to get **`random_state()`** to never visit duplicate states was pretty difficult. The functions I created for search algorithms utilized the attributes of **`S`** to evaluate nodes and make decisions of which ones to expand next, but I just couldn't figure out how to keep track of visited states while also building in the ability to backtrack if no valid moves were found. Eventually, I had the idea of simultaneously using a set data structure to store visited states, while using a stack to record the current path. If a dead end was encountered, states could be popped out of the stack until a state was encountered that had at least one valid move. Ultimately, this is what **`random_state()`** needed to properly function.

Altogether, I thoroughly enjoyed this whole process. As we have discussed before, my background is in music education, and in the education industry, it's all about quantifying student's understanding of the material and measuring whether or not instructional methods contribute effectively to that understanding. While I've always been a proponent of this approach, it was oftentimes frustrating to implement in practice. Especially regarding my experience working in high school education, there are so many unseen variables with students–their home lives, interests, friend groups, etc. When I implemented strategies to quantify my students' progress towards my learning objectives, I sometimes felt like I was grasping at straws. For example, to measure their readiness for all-region band auditions, I would listen to them perform their audition material individually and score them on a rubric. However, I was the only judge, so their results had a good chance of being affected by my bias. Additionally, I taught less than 100 students, so I felt like my data may have been lacking a sufficient sample size.

However, with this report, it was satisfactory to work within a completely controllable environment. For once, it feels like my findings are reliable and accurate to the true reality of the situation. Going forward, I hope to have more opportunities to apply what we learn in class by conducting research like this.