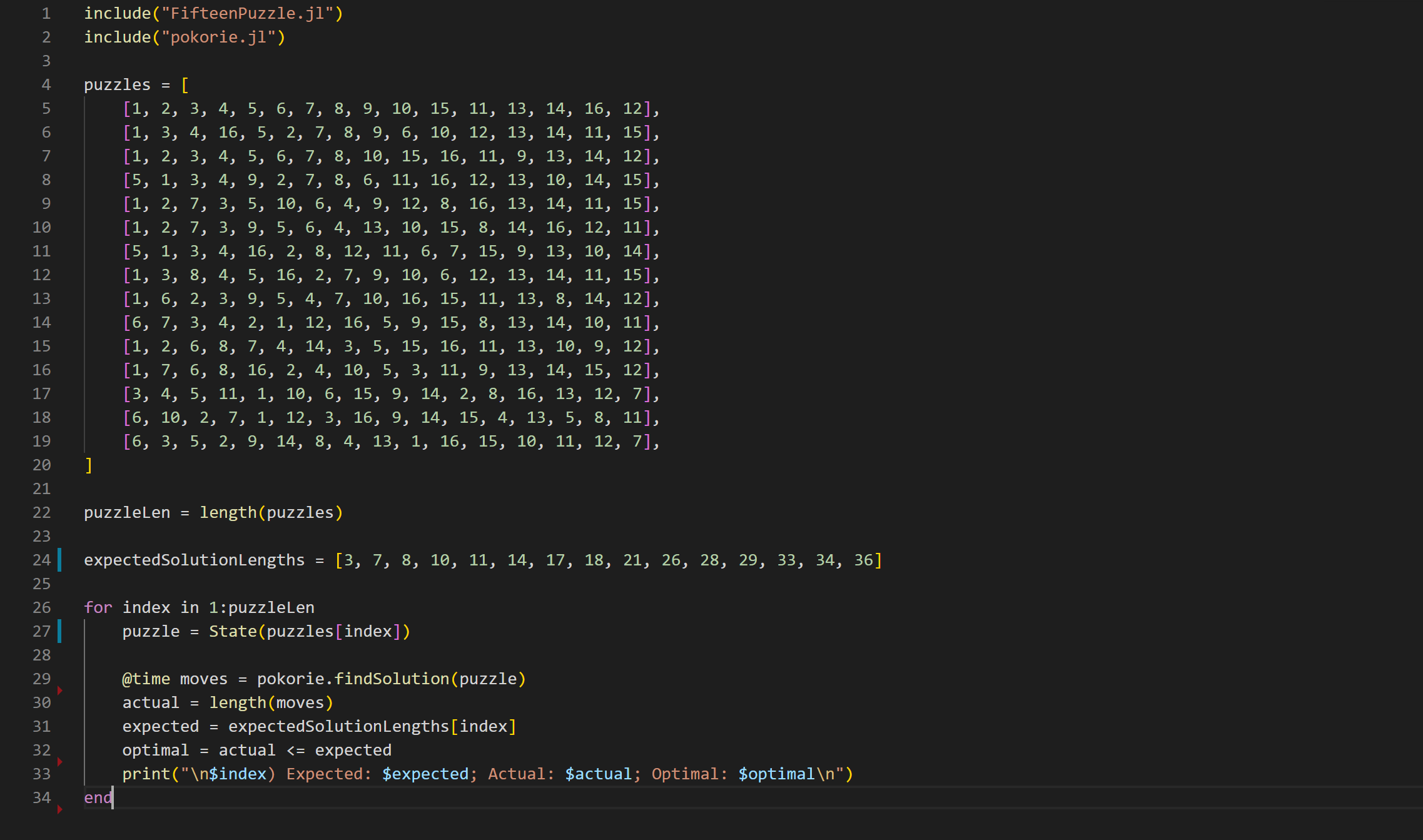
Artificial Intelligence

Fifteen Puzzle Contest

To find the optimal solution for the 15-puzzle game, my code went through a series of changes from a simple breadth-first search algorithm to an a\* heuristic search algorithm. In addition to writing the required module for the search, I created a simple test suite to check that the solution returned by my search module was optimal and took minimum time to execute. The test suite is as follows:



As shown in the figure above, the test suite involves 15 different puzzles to be solved by the search module, *pokorie*. On each run, the expected and actual number of moves are printed. The time to execute are also printed. With these values, I was able to improve the search algorithm.

**First try (Using out-of-place tiles as a heuristic)**

After implementing a simple breadth-first search algorithm, I quickly realized that although I was finding the solution to the puzzles, it wasn’t optimal and it took so much time as the difficulty increased.

To remedy this, I introduced a heuristic function that calculated the number of out-of-place tiles for each state and used it the priority indicator. This ensured that the state with the lowest out-of-place tile was picked next, since that would get us closer to the goal state.

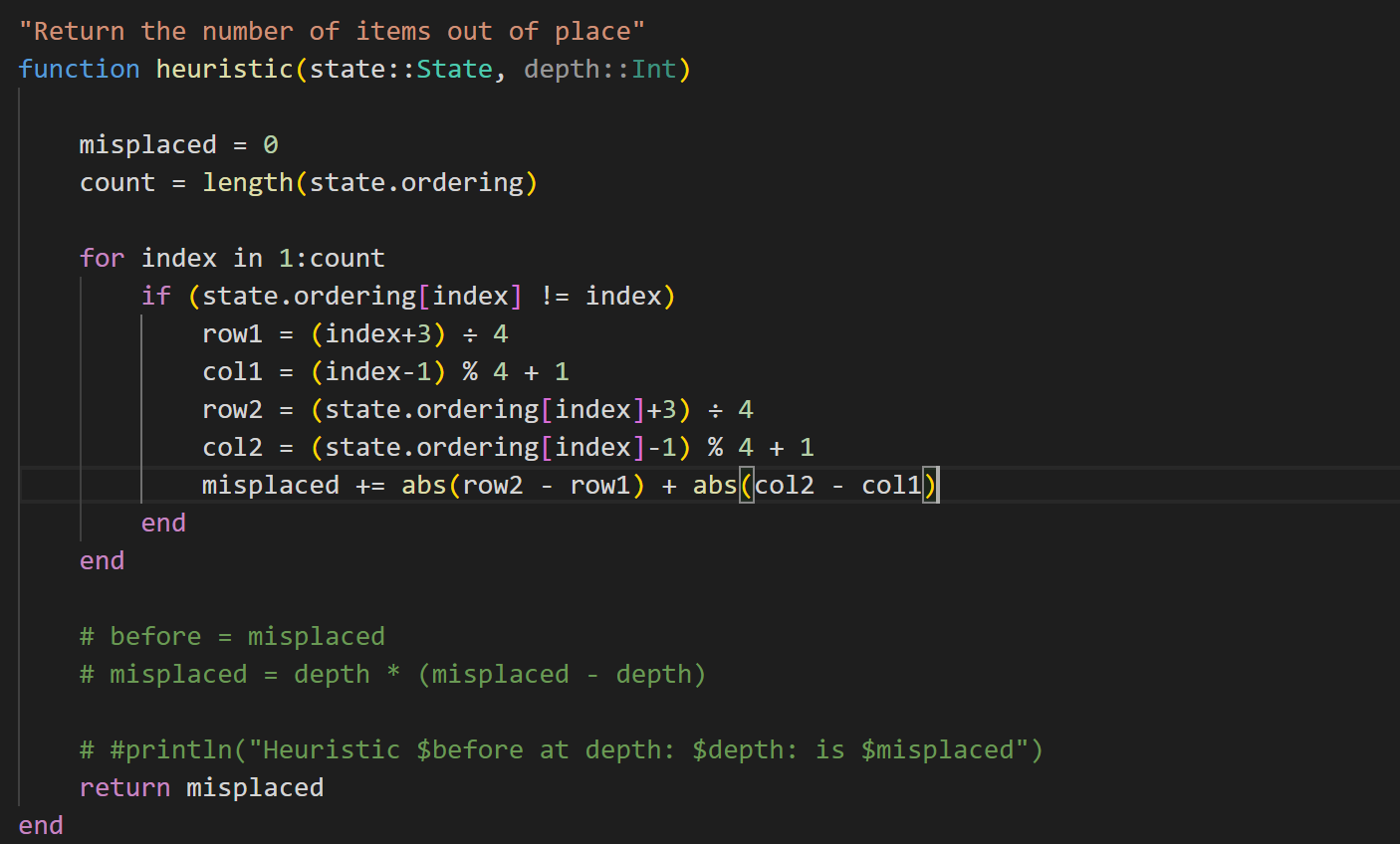
Upon testing this, I was able to run the entire test suite in a reasonable time, an improvement from the simple breadth-first search which took forever to complete. Nevertheless, the results were not optimal as shown below

* 1) Expected: 3; Actual: 3; Optimal: true
* 2) Expected: 7; Actual: 7; Optimal: true
* 3) Expected: 8; Actual: 46; Optimal: false
* 4) Expected: 10; Actual: 10; Optimal: true
* 5) Expected: 11; Actual: 119; Optimal: false
* 6) Expected: 14; Actual: 14; Optimal: true
* 7) Expected: 17; Actual: 27; Optimal: false
* 8) Expected: 18; Actual: 192; Optimal: false
* 9) Expected: 21; Actual: 147; Optimal: false
* 10) Expected: 26; Actual: 1224; Optimal: false
* 11) Expected: 28; Actual: 206; Optimal: false
* 12) Expected: 29; Actual: 309; Optimal: false
* 13) Expected: 33; Actual: 381; Optimal: false
* 14) Expected: 34; Actual: 230; Optimal: false
* 15) Expected: 36; Actual: 1114; Optimal: false

Furthermore, I tried subtracting the depth of each state from the out-of-place value, but that did not improve the result at all. On the contrary, the number of moves needed to find a solution increased significantly.

**Second Try (Manhattan distance)**

Upon learning about the Manhattan distance approach, I rewrote the heuristic function as shown below



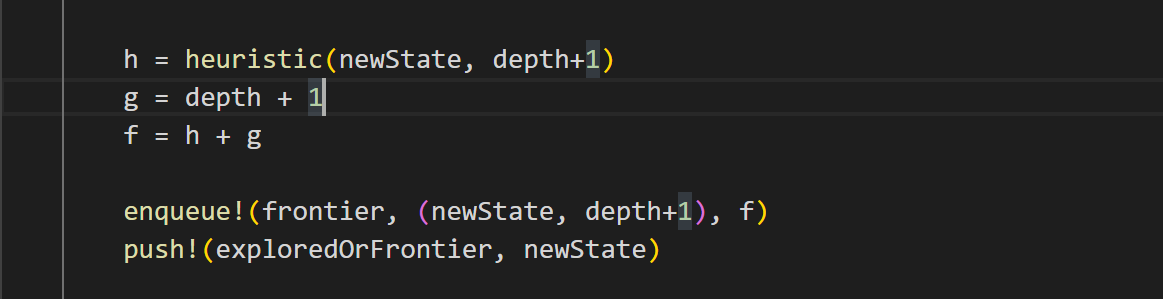
The Manhattan distance calculates the minimum number of vertical and horizontal positions an out-of-place tile will need to move through to be in its desired position if all other tiles were ignored. Using the Manhattan distance provides a better lower-bound heuristic to aid the search algorithm to choose a state more likely to lead to the goal state. Below are the test results

* 1) Expected: 3; Actual: 3; Optimal: true
* 2) Expected: 7; Actual: 7; Optimal: true
* 3) Expected: 8; Actual: 8; Optimal: true
* 4) Expected: 10; Actual: 10; Optimal: true
* 5) Expected: 11; Actual: 11; Optimal: true
* 6) Expected: 14; Actual: 14; Optimal: true
* 7) Expected: 17; Actual: 35; Optimal: false
* 8) Expected: 18; Actual: 22; Optimal: false
* 9) Expected: 21; Actual: 49; Optimal: false
* 10) Expected: 26; Actual: 118; Optimal: false
* 11) Expected: 28; Actual: 254; Optimal: false
* 12) Expected: 29; Actual: 273; Optimal: false
* 13) Expected: 33; Actual: 265; Optimal: false
* 14) Expected: 34; Actual: 232; Optimal: false
* 15) Expected: 36; Actual: 680; Optimal: false

In comparison to the out-of-state approach, the search module now takes fewer moves to get to a solution, but not the optimal solution yet.

**Third try (A\* heuristic search - Introducing depth)**

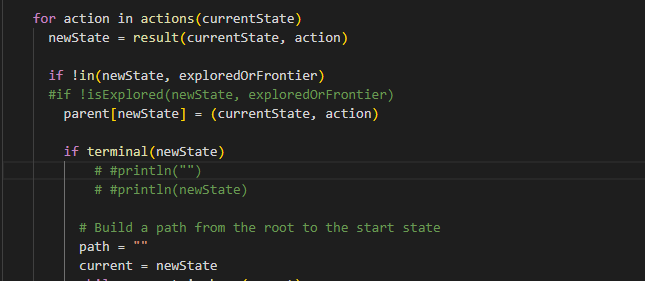
More reading led me to consider introducing to the heuristic function the depth already explored before arriving to a state. As such, I included it in the evaluation of the priority of a state as shown below

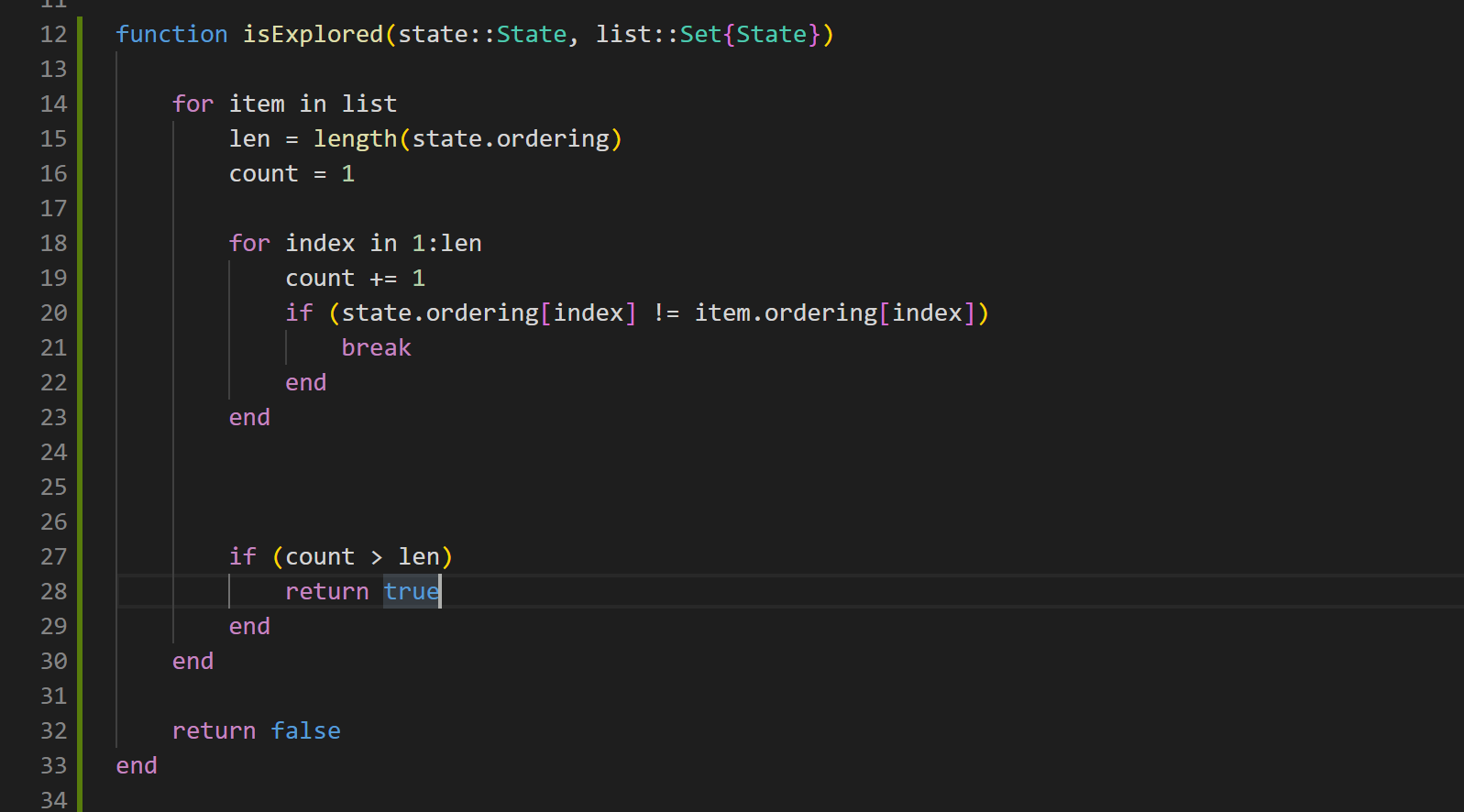


This led to a significant jump in the accuracy of the search algorithm as shown in the result below. 12/15 of the tests were coming back as optimal. I knew that I was close to have it working as expected.

* 1) Expected: 3; Actual: 3; Optimal: true
* 2) Expected: 7; Actual: 7; Optimal: true
* 3) Expected: 8; Actual: 8; Optimal: true
* 4) Expected: 10; Actual: 10; Optimal: true
* 5) Expected: 11; Actual: 11; Optimal: true
* 6) Expected: 14; Actual: 14; Optimal: true
* 7) Expected: 17; Actual: 17; Optimal: true
* 8) Expected: 18; Actual: 18; Optimal: true
* 9) Expected: 21; Actual: 21; Optimal: true
* 10) Expected: 26; Actual: 26; Optimal: true
* 11) Expected: 28; Actual: 28; Optimal: true
* 12) Expected: 29; Actual: 29; Optimal: true
* 13) Expected: 33; Actual: 35; Optimal: false

After looking at my code again, I removed a custom code that I had written to check if a state had been explored as shown below





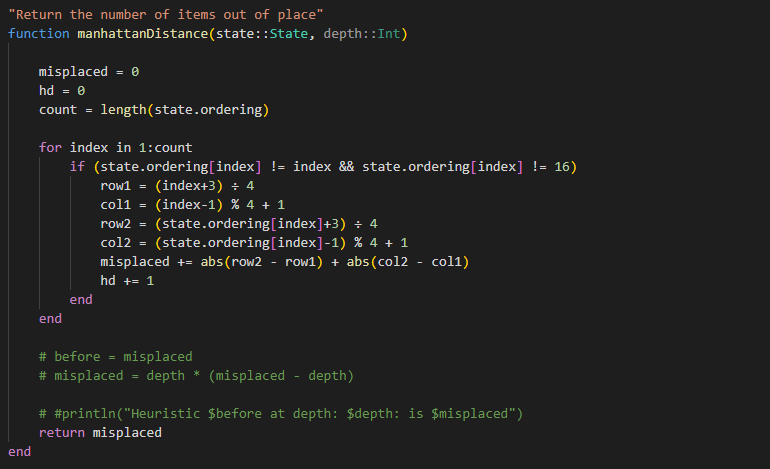
After running the test suite, all the puzzles were solved, and the optimal moves were also found as shown in the test result below.

* 1) Expected: 3; Actual: 3; Optimal: true
* 2) Expected: 7; Actual: 7; Optimal: true
* 3) Expected: 8; Actual: 8; Optimal: true
* 4) Expected: 10; Actual: 10; Optimal: true
* 5) Expected: 11; Actual: 11; Optimal: true
* 6) Expected: 14; Actual: 14; Optimal: true
* 7) Expected: 17; Actual: 17; Optimal: true
* 8) Expected: 18; Actual: 18; Optimal: true
* 9) Expected: 21; Actual: 21; Optimal: true
* 10) Expected: 26; Actual: 26; Optimal: true
* 11) Expected: 28; Actual: 28; Optimal: true
* 12) Expected: 29; Actual: 29; Optimal: true
* 13) Expected: 33; Actual: 33; Optimal: true
* 14) Expected: 34; Actual: 34; Optimal: true
* 15) Expected: 36; Actual: 36; Optimal: true

Nevertheless, it was still taking some time. I realized this when I pushed my code to git and checked on the contestant website. I had the following score:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 87.72 | 4.510 | pokorie | 0.000 | 0.000 | 0.000 | 0.017 | 2.136 | 1.057 | 7.867 | 1.153 | 32.836 | 0.030 |

I took another look at my code and realized that I was counting the spaces (identified as the number 16) in my heuristic function. Therefore, I removed it and that brought my timing down to the following:



|  |  |  |
| --- | --- | --- |
| **Test** | **Before** | **After** |
| 1 | 0.000021 seconds (50 allocations: 6.859 KiB) | 0.000022 seconds (50 allocations: 6.859 KiB) |
| 2 | 0.000031 seconds (137 allocations: 23.703 KiB) | 0.000028 seconds (108 allocations: 18.406 KiB) |
| 3 | 0.000066 seconds (254 allocations: 58.141 KiB) | 0.000021 seconds (132 allocations: 22.469 KiB) |
| 4 | 0.000275 seconds (1.32 k allocations: 312.656 KiB) | 0.000028 seconds (166 allocations: 27.453 KiB) |
| 5 | 0.000076 seconds (338 allocations: 73.844 KiB) | 0.000024 seconds (171 allocations: 27.953 KiB) |
| 6 | 0.000064 seconds (310 allocations: 64.938 KiB) | 0.000027 seconds (180 allocations: 27.938 KiB) |
| 7 | 0.000088 seconds (394 allocations: 80.672 KiB) | 0.000037 seconds (226 allocations: 44.219 KiB) |
| 8 | 3.471400 seconds (4.80 M allocations: 1.154 GiB, 16.04% gc time) | 0.632997 seconds (1.10 M allocations: 276.772 MiB, 21.52% gc time) |
| 9 | 0.105555 seconds (262.37 k allocations: 64.788 MiB) | 0.010968 seconds (37.80 k allocations: 10.255 MiB) |
| 10 | 32.460887 seconds (30.99 M allocations: 7.958 GiB, 28.11% gc time) | 0.870001 seconds (1.41 M allocations: 353.510 MiB, 23.51% gc time) |
| 11 | 17.820014 seconds (19.80 M allocations: 4.731 GiB, 18.09% gc time) | 0.322676 seconds (671.16 k allocations: 153.417 MiB, 17.93% gc time) |
| 12 | 30.335153 seconds (27.72 M allocations: 6.385 GiB, 29.89% gc time) | 0.577142 seconds (968.07 k allocations: 248.955 MiB, 24.55% gc time) |
| 13 | 13.862246 seconds (12.75 M allocations: 2.993 GiB, 21.44% gc time) | 0.194055 seconds (434.07 k allocations: 94.130 MiB, 25.80% gc time) |
| 14 | 156.284483 seconds (100.69 M allocations: 23.826 GiB, 48.90% gc time) | 8.755848 seconds (12.71 M allocations: 3.028 GiB, 21.29% gc time) |
| 15 | 38.178768 seconds (36.39 M allocations: 8.976 GiB, 31.08% gc time) | 17.425024 seconds (21.80 M allocations: 5.094 GiB, 29.22% gc time) |

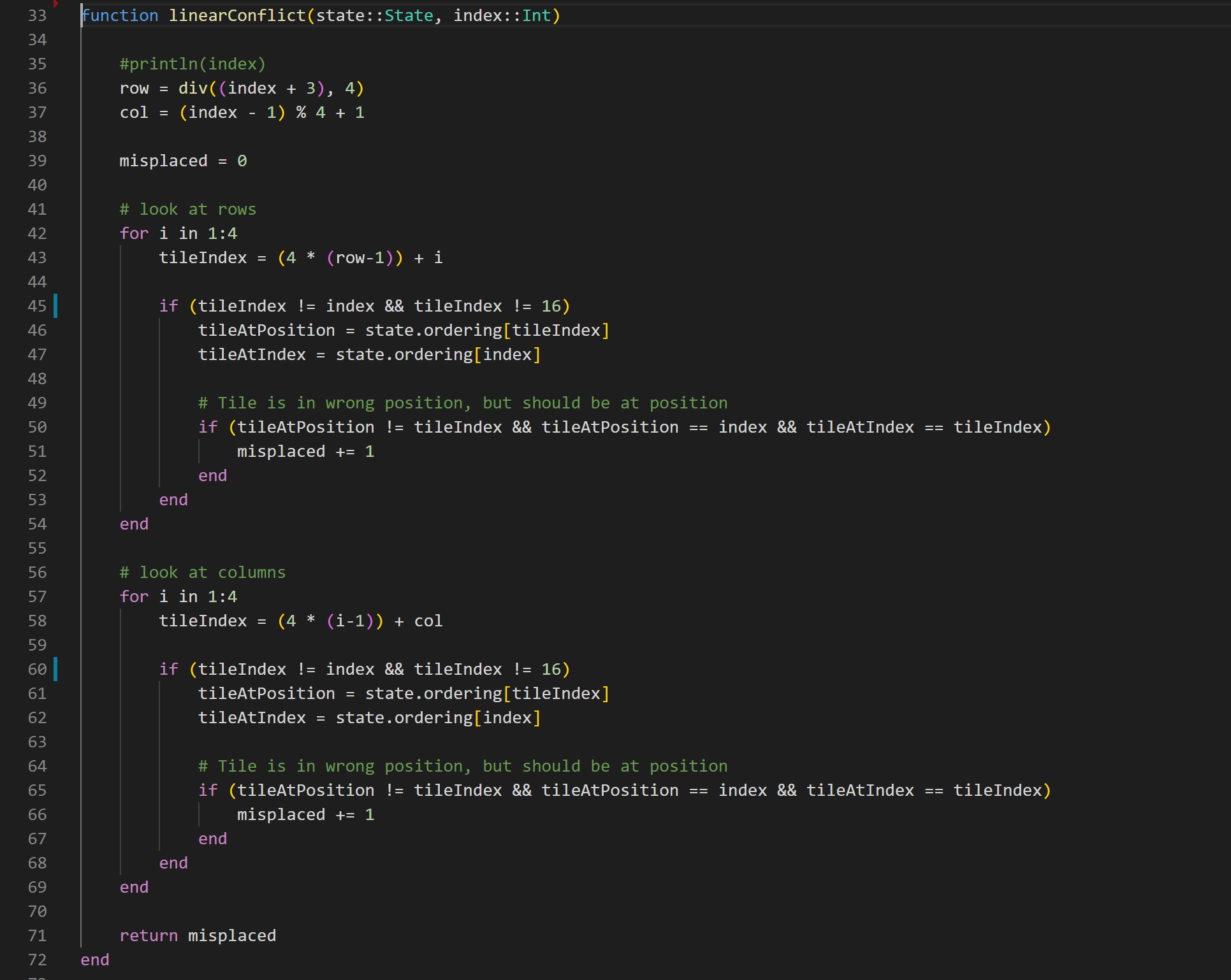
Something to note as well is the big reduction in the amount of memory allocated to the search tree. My performance on the contest website improved to the following:

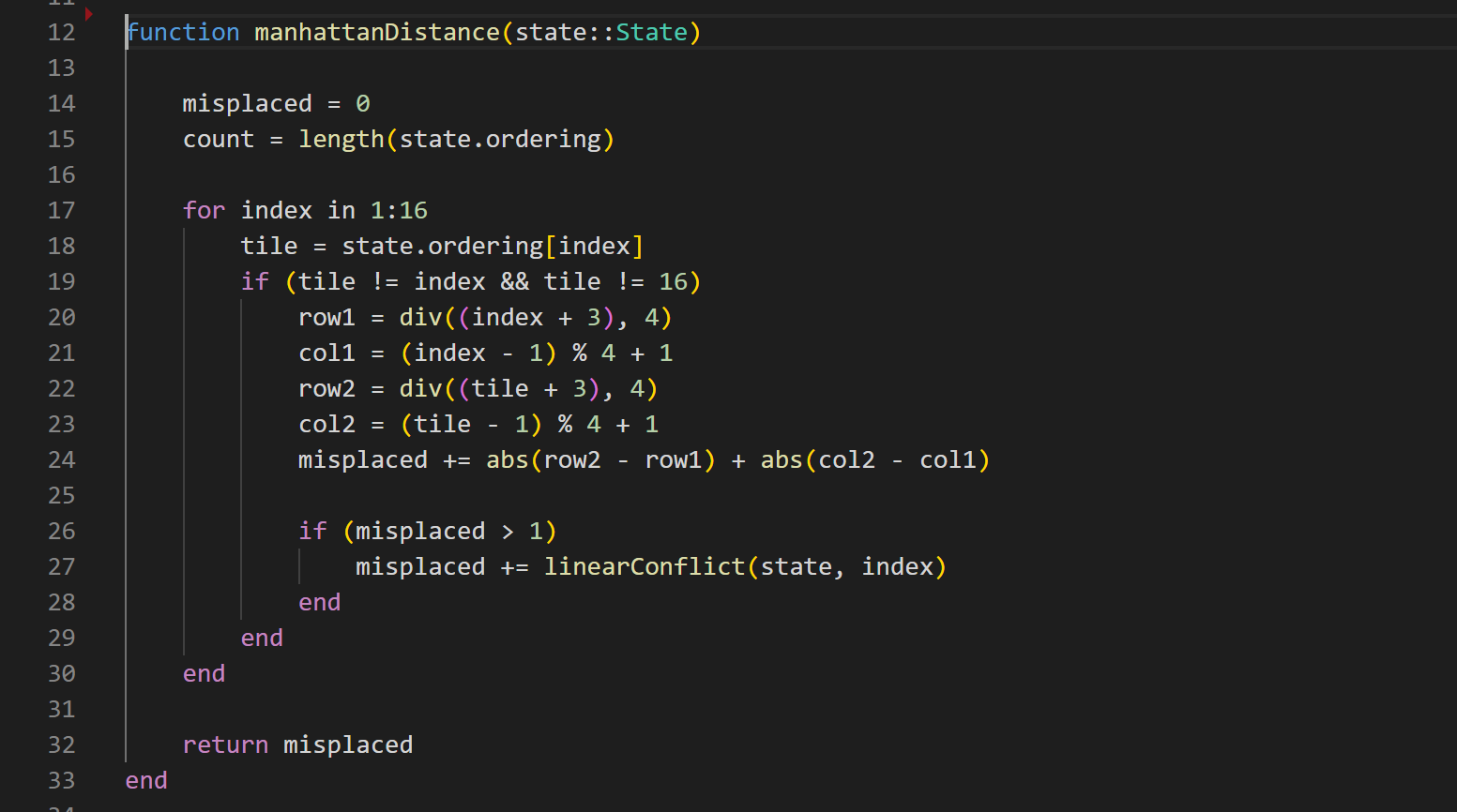
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 92.01 | 1.481 | pokorie | 0.000 | 0.000 | 0.000 | 0.015 | 2.143 | 0.904 | 4.471 | 1.856 | 5.407 | 0.013 |

**Fourth try (Addition of linear constraints)**

The next evolution to my code was to introduce linear conflicts. Two tiles ***‘a’ and ‘b’*** are in a linear conflict if they are in the same row or column, also their goal positions are in the same row or column and the goal position of one of the tiles is blocked by the other tile in that row[1].

I added the following code to account for the linear constraint





This indeed improved the execution time for the tests with higher moves as shown in the table below

|  |  |  |
| --- | --- | --- |
| **Test** | **Before** | **After** |
| 1 | 0.000022 seconds (50 allocations: 6.859 KiB) | 0.000022 seconds (50 allocations: 6.859 KiB) |
| 2 | 0.000028 seconds (108 allocations: 18.406 KiB) | 0.000033 seconds (108 allocations: 18.406 KiB) |
| 3 | 0.000021 seconds (132 allocations: 22.469 KiB) | 0.000027 seconds (118 allocations: 20.219 KiB) |
| 4 | 0.000028 seconds (166 allocations: 27.453 KiB) | 0.000072 seconds (152 allocations: 25.203 KiB) |
| 5 | 0.000024 seconds (171 allocations: 27.953 KiB) | 0.000036 seconds (157 allocations: 25.703 KiB) |
| 6 | 0.000027 seconds (180 allocations: 27.938 KiB) | 0.000047 seconds (180 allocations: 27.938 KiB) |
| 7 | 0.000037 seconds (226 allocations: 44.219 KiB) | 0.000053 seconds (226 allocations: 44.219 KiB) |
| 8 | 0.632997 seconds (1.10 M allocations: 276.772 MiB, 21.52% gc time) | 0.190979 seconds (382.76 k allocations: 87.445 MiB, 11.94% gc time) |
| 9 | 0.010968 seconds (37.80 k allocations: 10.255 MiB) | 0.016855 seconds (40.45 k allocations: 10.675 MiB) |
| 10 | 0.870001 seconds (1.41 M allocations: 353.510 MiB, 23.51% gc time) | 0.206008 seconds (272.94 k allocations: 68.536 MiB, 36.62% gc time) |
| 11 | 0.322676 seconds (671.16 k allocations: 153.417 MiB, 17.93% gc time) | 0.164199 seconds (352.81 k allocations: 81.917 MiB) |
| 12 | 0.577142 seconds (968.07 k allocations: 248.955 MiB, 24.55% gc time) | 0.521824 seconds (894.41 k allocations: 203.168 MiB, 20.37% gc time) |
| 13 | 0.194055 seconds (434.07 k allocations: 94.130 MiB, 25.80% gc time) | 0.365636 seconds (626.46 k allocations: 144.843 MiB, 16.18% gc time) |
| 14 | 8.755848 seconds (12.71 M allocations: 3.028 GiB, 21.29% gc time) | 4.768720 seconds (6.82 M allocations: 1.586 GiB, 14.20% gc time) |
| 15 | 17.425024 seconds (21.80 M allocations: 5.094 GiB, 29.22% gc time) | 10.015753 seconds (12.20 M allocations: 2.946 GiB, 16.31% gc time) |

**Conclusion**

Through iteration, testing, and the introduction of different heuristic approaches, I was able to improve my search algorithm to solve a 15-puzzle game optimally. There’s differently room for improvement with techniques like pattern databases.

**References**

[1] <https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/>