Alessio Mazzone
CS 1571 – Artificial Intelligence
Project 1 Report

1) Briefly describe your project setup: which version of Python? Basic computer configurations.

Python Version: 2.7.10
Computer: MacBook Pro Early 2015
IDE: Visual Studio Code with Microsoft Python Extension
Operating System: macOS Mojave version 10.14

To run the program, type the following in the command line:
python puzzlesolver.py XXX.config YYY ZZZ

XXX = name of the config file
YYY = algorithm name. Can be one of the following: "**bfs**", "**dfs**", "**unicost**", "**greedy**", "**astar**"
ZZZ = optional heuristic, to be used for greedy and astar algorithms. Can be one of the following values:

For water jugs	→ "**proximity**"
For path finding cities	→ "**euclidean**"
For tiles	→ "**misplaced**" or "**manhattan**"

Example runs:
To run the cities puzzle with the Euclidean heuristic and astar algorithm, type:
python puzzlesolver.py test_cities.config astar euclidean

To run the jugs puzzle with unicost, type:
python puzzlesolver.py test_jugs.config unicost

2) If you discussed the project with someone, list their names and briefly describe what you talked about. Relatedly, if you consulted any sources outside of the class notes and the textbook, please give citations to them here.

I discussed string/tuple/state/board/path representation with Hassan Syed.

No outside sources other than class notes, the textbook, and the Python 2 documentation (https://docs.python.org/2/contents.html) were used.

3) Report any known bugs you have.

There are no known bugs in my program.

4) For each puzzle type:

a) Enumerate your action expansion order

b) Give the name of the heuristic function (so that the grader can supply it as an argument in the command line). If you made up any heuristic function(s) in addition to the ones we specified, explain what it is and why you designed it that way.

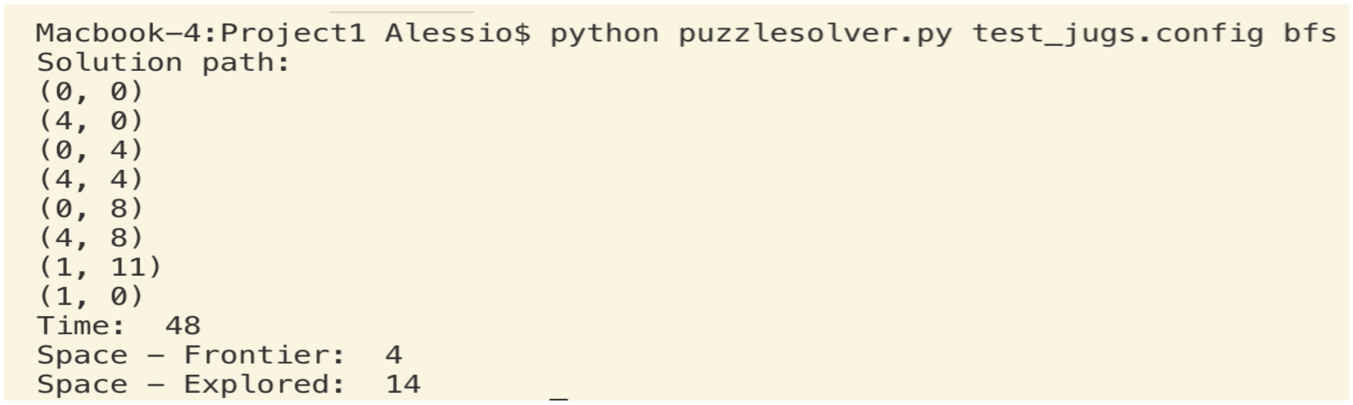
c) show a transcript of the expected run.

Water Jugs:

4a) My actions are added to the actions array in the following order:
If jug1 is empty, it is filled from the tap.
If jug1 is not empty but not at capacity, it is filled from the tap or dumped to ground.
If jug2 is empty or not at capacity, jug1 is dumped into jug2.
If jug2 is empty, it is filled from the tap.
If jug2 is not empty but not at capacity, it is filled from the tap or dumped to ground.
If jug1 is empty or not at capacity, jug2 is dumped into jug1.

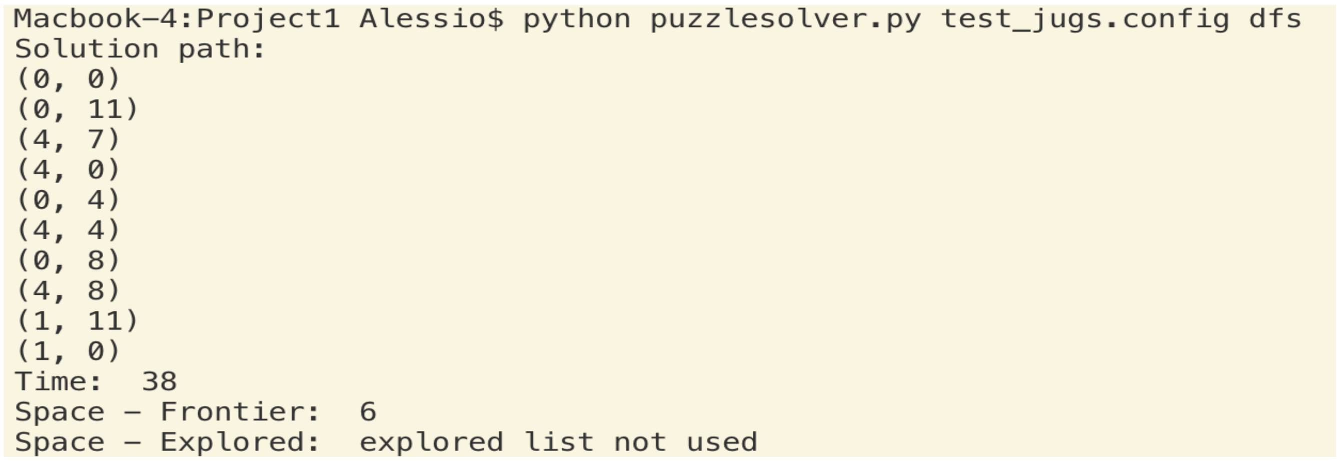
4b) Heuristic function: "proximity"
Proximity function checks the state for how close it is to the goal state. It checks how far away jug1 is from the goal and how far away jug2 is from the goal. For three jug problem, jug3 is also checked. The cost is calculated as:
Cost two jugs = abs(curr_jug1 - gloal_jug1) + abs(curr_jug2 - goal_jug2)
Cost three jugs = abs(curr_jug1 - gloal_jug1) + abs(curr_jug2 - goal_jug2) + abs(curr_jug3 – goal_jug3)

4c)
test_jugs.config with bfs:



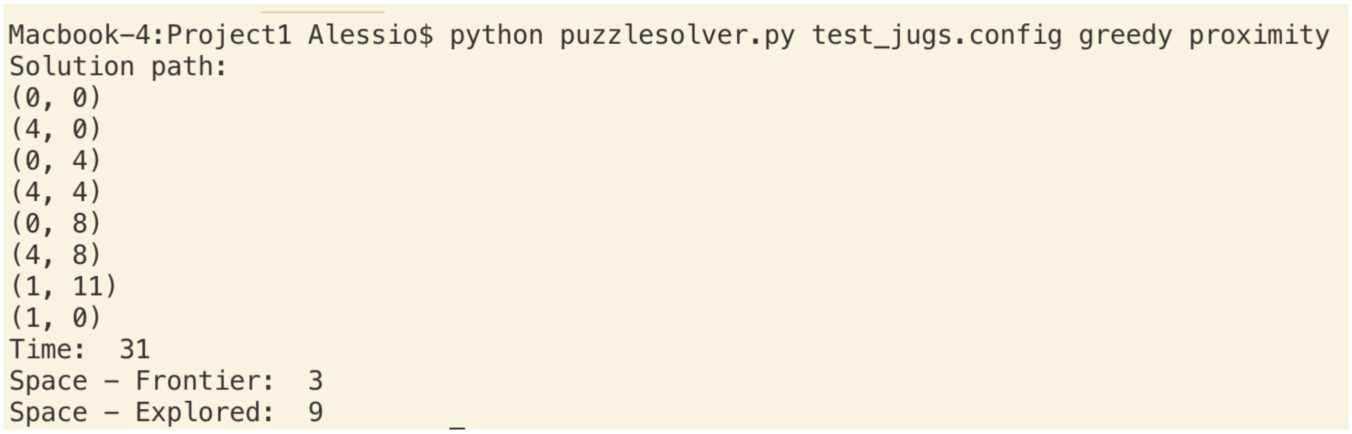
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_jugs.config bfs
Solution path:
(0, 0)
(4, 0)
(0, 4)
(4, 4)
(0, 8)
(4, 8)
(1, 11)
(1, 0)
Time:  48
Space - Frontier:  4
Space - Explored:  14
```

test_jugs.config with dfs:



```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_jugs.config dfs
Solution path:
(0, 0)
(0, 11)
(4, 7)
(4, 0)
(0, 4)
(4, 4)
(0, 8)
(4, 8)
(1, 11)
(1, 0)
Time:  38
Space - Frontier:  6
Space - Explored:  explored list not used
```

test_jugs.config with greedy proximity:



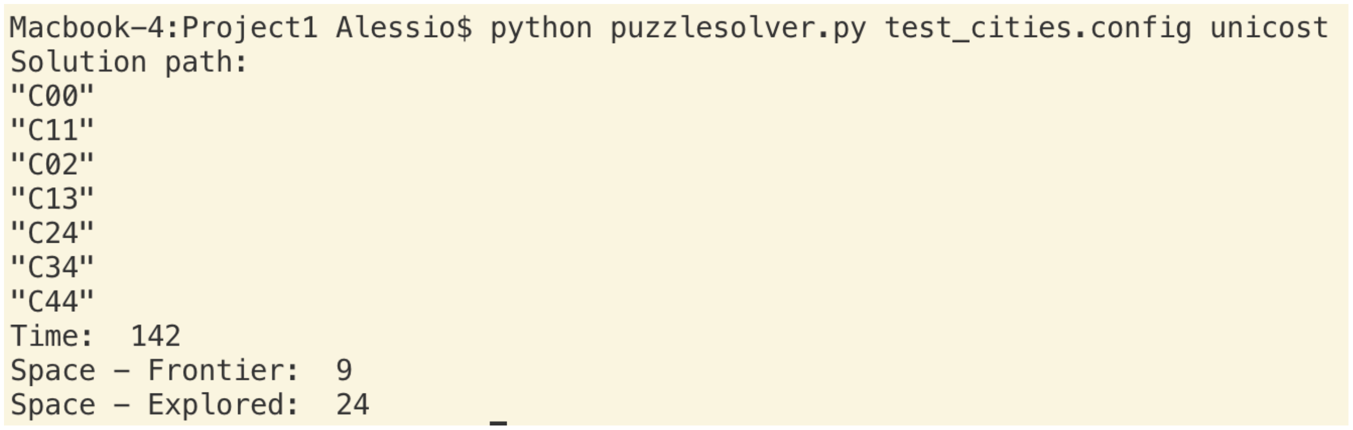
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_jugs.config greedy proximity
Solution path:
(0, 0)
(4, 0)
(0, 4)
(4, 4)
(0, 8)
(4, 8)
(1, 11)
(1, 0)
Time:  31
Space - Frontier:  3
Space - Explored:  9
```

Path Finding Cities:

4a) My actions are added to the actions array based on the order the current city appears in the list of edges. Starting from the first edge found on configuration file line 5 and moving down, if the current city is one of the two cities listed in the edge, then that edge is added to the actions array.

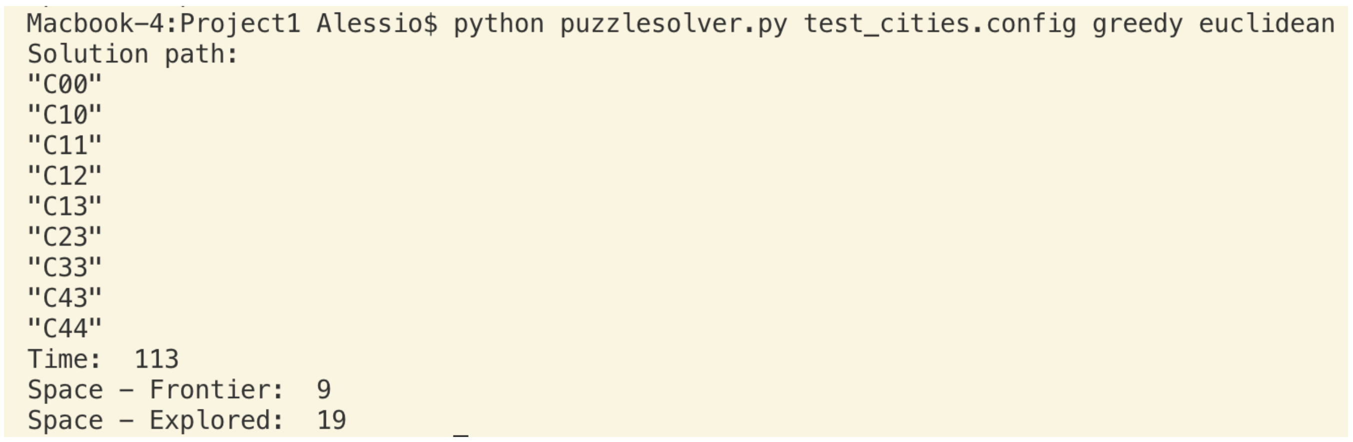
4b) Heuristic function: "euclidean"
We are given the coordinates of the cities on a grid in the configuration file. The Euclidean heuristic function determines the distance between two cities based on the coordinates treated as (x,y). The distance between the cities is just the Pythagorean theorem, or the Euclidean distance.

4c)
test_cities.config with unicost



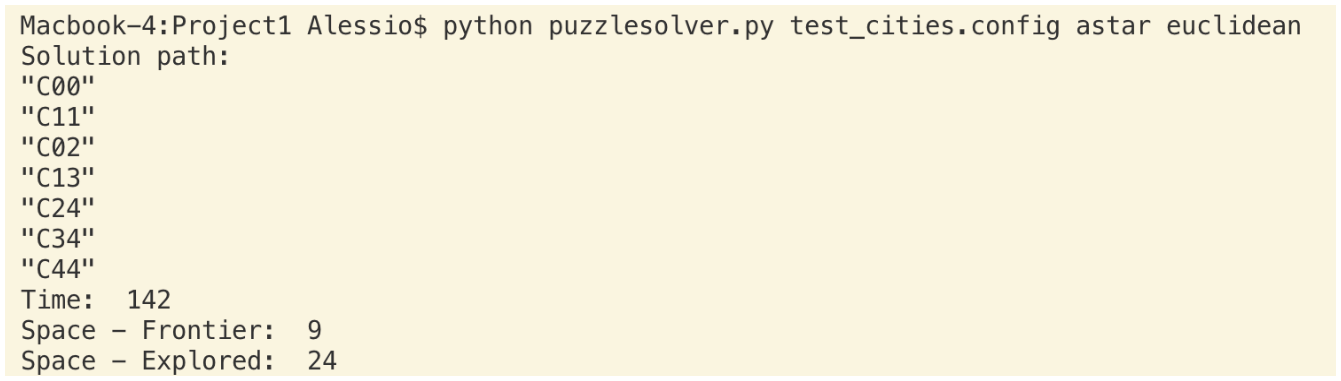
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_cities.config unicost
Solution path:
"C00"
"C11"
"C02"
"C13"
"C24"
"C34"
"C44"
Time:  142
Space - Frontier:  9
Space - Explored:  24
```

test_cities.config with greedy euclidean:



```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_cities.config greedy euclidean
Solution path:
"C00"
"C10"
"C11"
"C12"
"C13"
"C23"
"C33"
"C43"
"C44"
Time:  113
Space - Frontier:  9
Space - Explored:  19
```

test_cities.config with astar euclidean:



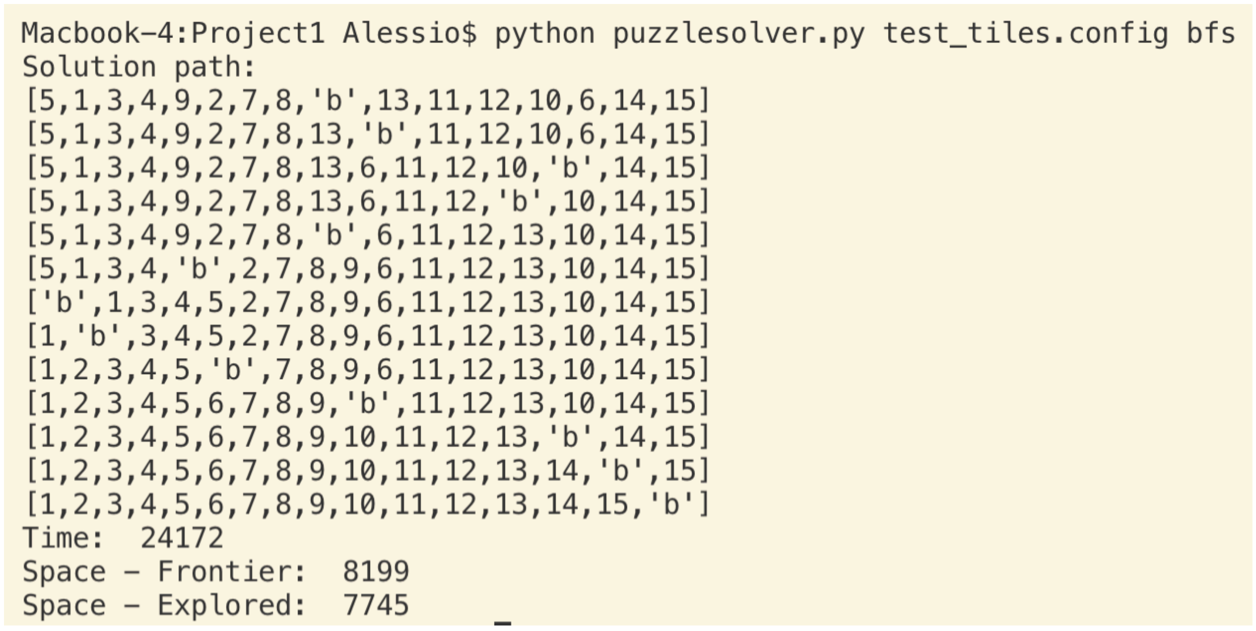
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_cities.config astar euclidean
Solution path:
"C00"
"C11"
"C02"
"C13"
"C24"
"C34"
"C44"
Time:  142
Space - Frontier:  9
Space - Explored:  24
```

Tiles

4a) With respect to the blank, actions are added to the actions array by swapping blank with tile to its right, up, left, and down.

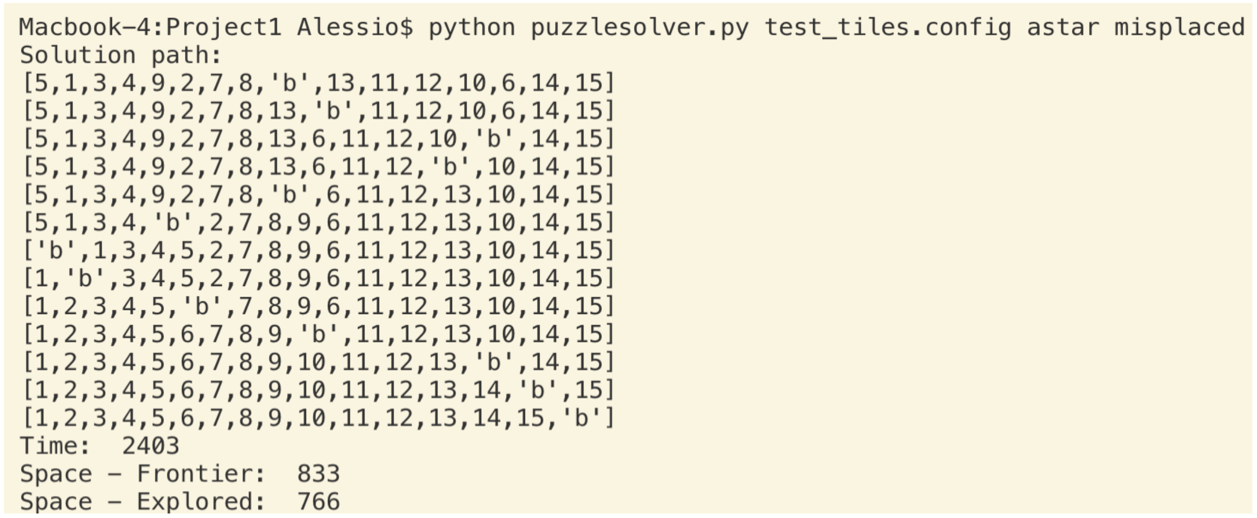
4b) Two heuristic functions used: “misplaced” and “manhattan”
Both heuristics were implemented as explained in class. Misplaced heuristic counts the number of tiles that are not in the correct place with respect to goal. Manhattan heuristic counts how many slides away from where a tile is on goal a tile is currently and sums them together.

4c)
test_tiles.config with bfs



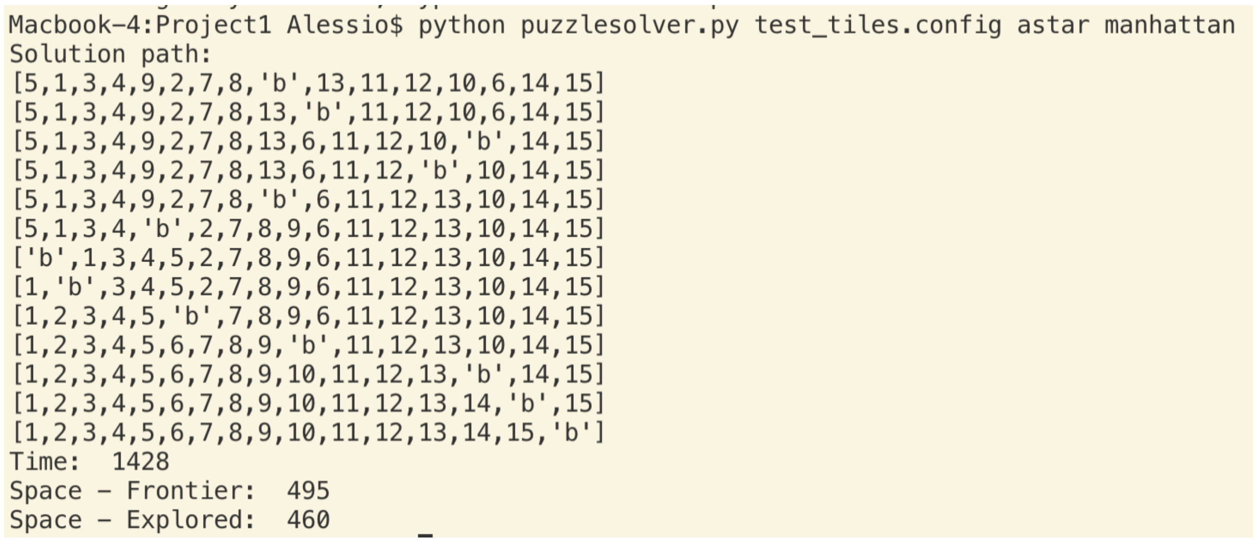
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_tiles.config bfs
Solution path:
[5,1,3,4,9,2,7,8,'b',13,11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,'b',11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,10,'b',14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,'b',10,14,15]
[5,1,3,4,9,2,7,8,'b',6,11,12,13,10,14,15]
[5,1,3,4,'b',2,7,8,9,6,11,12,13,10,14,15]
['b',1,3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,'b',3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,'b',7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,'b',11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,'b',14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,'b',15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,'b']
Time:  24172
Space - Frontier:  8199
Space - Explored:  7745
```

test_tiles.config with astar misplaced:



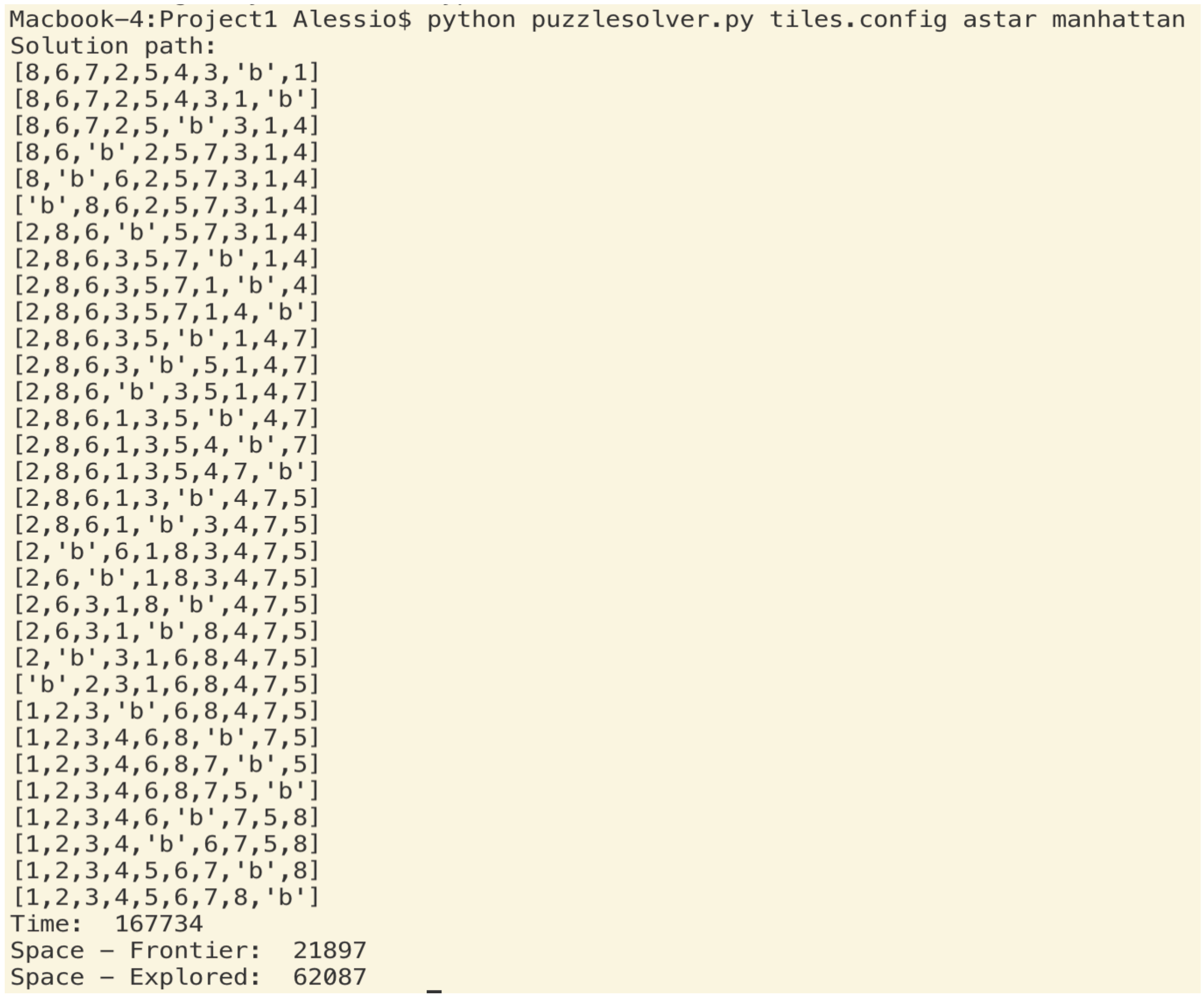
```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_tiles.config astar misplaced
Solution path:
[5,1,3,4,9,2,7,8,'b',13,11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,'b',11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,10,'b',14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,'b',10,14,15]
[5,1,3,4,9,2,7,8,'b',6,11,12,13,10,14,15]
[5,1,3,4,'b',2,7,8,9,6,11,12,13,10,14,15]
['b',1,3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,'b',3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,'b',7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,'b',11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,'b',14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,'b',15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,'b']
Time:  2403
Space - Frontier:  833
Space - Explored:  766
```

test_tiles.config with astar manhattan:



```
Macbook-4:Project1 Alessio$ python puzzlesolver.py test_tiles.config astar manhattan
Solution path:
[5,1,3,4,9,2,7,8,'b',13,11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,'b',11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,10,'b',14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,'b',10,14,15]
[5,1,3,4,9,2,7,8,'b',6,11,12,13,10,14,15]
[5,1,3,4,'b',2,7,8,9,6,11,12,13,10,14,15]
['b',1,3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,'b',3,4,5,2,7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,'b',7,8,9,6,11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,'b',11,12,13,10,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,'b',14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,'b',15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,'b']
Time:  1428
Space - Frontier:  495
Space - Explored:  460
```

BFS and astar with misplaced tiles heuristic did not work for tiles.config file within the 30 minute time frame. Only astar with manhattan heuristic was able to finish in under 30 minutes. Results are found below:



```
Macbook-4:Project1 Alessio$ python puzzlesolver.py tiles.config astar manhattan
Solution path:
[8,6,7,2,5,4,3,'b',1]
[8,6,7,2,5,4,3,1,'b']
[8,6,7,2,5,'b',3,1,4]
[8,6,'b',2,5,7,3,1,4]
[8,'b',6,2,5,7,3,1,4]
['b',8,6,2,5,7,3,1,4]
[2,8,6,'b',5,7,3,1,4]
[2,8,6,3,5,7,'b',1,4]
[2,8,6,3,5,7,1,'b',4]
[2,8,6,3,5,7,1,4,'b']
[2,8,6,3,5,'b',1,4,7]
[2,8,6,3,'b',5,1,4,7]
[2,8,6,'b',3,5,1,4,7]
[2,8,6,1,3,5,'b',4,7]
[2,8,6,1,3,5,4,'b',7]
[2,8,6,1,3,5,4,7,'b']
[2,8,6,1,3,'b',4,7,5]
[2,8,6,1,'b',3,4,7,5]
[2,'b',6,1,8,3,4,7,5]
[2,6,'b',1,8,3,4,7,5]
[2,6,3,1,8,'b',4,7,5]
[2,6,3,1,'b',8,4,7,5]
[2,'b',3,1,6,8,4,7,5]
['b',2,3,1,6,8,4,7,5]
[1,2,3,'b',6,8,4,7,5]
[1,2,3,4,6,8,'b',7,5]
[1,2,3,4,6,8,7,'b',5]
[1,2,3,4,6,8,7,5,'b']
[1,2,3,4,6,'b',7,5,8]
[1,2,3,4,'b',6,7,5,8]
[1,2,3,4,5,6,7,'b',8]
[1,2,3,4,5,6,7,8,'b']
Time:  167734
Space - Frontier:  21897
Space - Explored:  62087
```

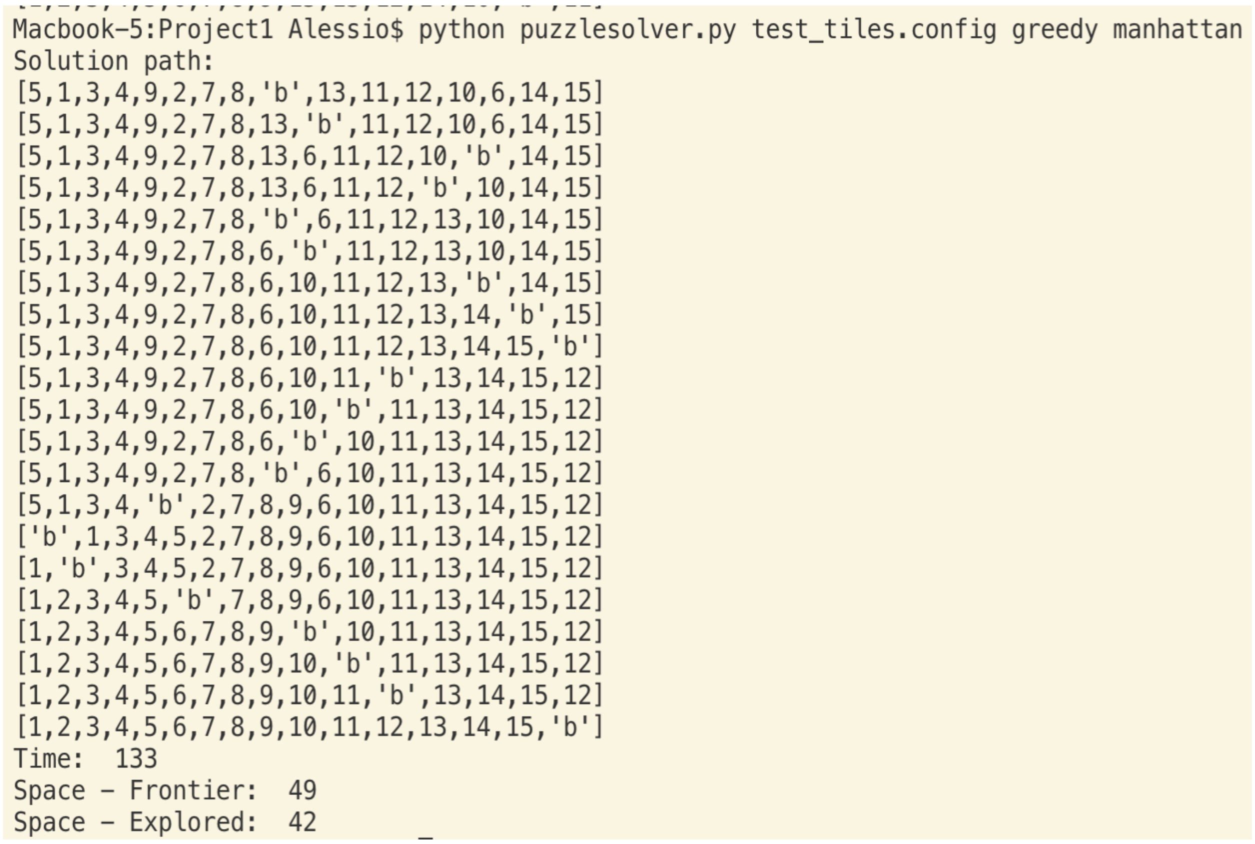
5) Additional discussion points: Here, you may want to run some test cases on more search strategies than those required for the transcript. Address the following issues:

a. **For each of the three puzzle types, what do you think is the best search strategy, and why? You should take all four factors into considerations (completeness, optimality, time complexity, space complexity).**
b. **For the water jugs problem, is your heuristic admissible and consistent? Explain.**
c. **You may also include additional discussions on any observations that you find surprising and/or interesting.**

5a) For the water jug problem using test_jugs.config, Based on the data alone, DFS probably performed best. DFS is complete in this case because we have a finite state space, but it is not optimal. The path found by DFS was indeed longer than the all the algorithms by a few nodes, but it saved on space because it did not use an explored list, while the other algorithms did. Further, based on the data, the time it took was almost as good as the Greedy algorithm, and, considering again that it didn't use an explored list, DFS is probably better still. The space complexity of DFS here is better than all the other algorithms.

For the city planning problem using test_cities.config, the greedy algorithm using the Euclidean heuristic was the best performer according to the data. Greedy performed better than the other algorithms in terms of time and space. But, in terms of shortest path, astar and unicost had shorter paths than greedy. The greedy algorithm is complete in finite state spaces. Greedy is optimal since the Euclidean herustic is admissible.

For the tile puzzle, using test_tiles.config, the greedy algorithm with the manhattan heuristic again performed best, according to the data. A screenshot of the performance is included below:



```
Macbook-5:Project1 Alessio$ python puzzlesolver.py test_tiles.config greedy manhattan
Solution path:
[5,1,3,4,9,2,7,8,'b',13,11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,'b',11,12,10,6,14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,10,'b',14,15]
[5,1,3,4,9,2,7,8,13,6,11,12,'b',10,14,15]
[5,1,3,4,9,2,7,8,'b',6,11,12,13,10,14,15]
[5,1,3,4,9,2,7,8,6,'b',11,12,13,10,14,15]
[5,1,3,4,9,2,7,8,6,10,11,12,13,'b',14,15]
[5,1,3,4,9,2,7,8,6,10,11,12,13,14,'b',15]
[5,1,3,4,9,2,7,8,6,10,11,12,13,14,15,'b']
[5,1,3,4,9,2,7,8,6,10,11,'b',13,14,15,12]
[5,1,3,4,9,2,7,8,6,10,'b',11,13,14,15,12]
[5,1,3,4,9,2,7,8,6,'b',10,11,13,14,15,12]
[5,1,3,4,9,2,7,8,'b',6,10,11,13,14,15,12]
[5,1,3,4,'b',2,7,8,9,6,10,11,13,14,15,12]
['b',1,3,4,5,2,7,8,9,6,10,11,13,14,15,12]
[1,'b',3,4,5,2,7,8,9,6,10,11,13,14,15,12]
[1,2,3,4,5,'b',7,8,9,6,10,11,13,14,15,12]
[1,2,3,4,5,6,7,8,9,'b',10,11,13,14,15,12]
[1,2,3,4,5,6,7,8,9,10,'b',11,13,14,15,12]
[1,2,3,4,5,6,7,8,9,10,11,'b',13,14,15,12]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,'b']
Time:  133
Space - Frontier:  49
Space - Explored:  42
```

In terms of space and time, it far surpasses all other algorithms, including astar with both heuristics. The path found by greedy is indeed much larger than the path found by astar, but the time and complexity were far, far superior in the greedy manhattan algorithm. Greedy is optimal since the Euclidean heuristic is admissible.

b) The heuristic I used for the water jug problem was something I call the proximity heuristic. The proximity heuristic checks the state (looks at all the jugs) and checks to see how close the current state of the jugs is to the goal state. It checks how far away jug1 is from the goal and how far away jug2 is from the goal. For three jugs problem, jug3 is also checked. To calculate h(n), I used the following formulas for 2 jugs and 3 jugs:
h(n) = abs(curr_jug1 - gloal_jug1) + abs(curr_jug2 - goal_jug2)
h(n) = abs(curr_jug1 - gloal_jug1) + abs(curr_jug2 - goal_jug2) + abs(curr_jug3 – goal_jug3)
The proximity heuristic is admissible, optimistic, and consistent because it can never overestimate how far away from the goal the jugs are, and so the heuristic value is always less than or equal to the actual distance. The closer the jugs are to the goal states, the lower h(n) is. The value of h(n) is 0 when both jugs are at their goal states.

c) As a final thought, for some reason I thought that the astar algorithm, no matter what the heuristic function, would always provide the best performance, as it is the most sophisticated algorithm of the five we implemented. But that was actually not the case for any of the runs. Although I will add that astar often had the shortest paths to the solutions, especially for the test_tiles file.