Introduction to Mathematical Foundations and Principles of Autonomy - Homework 2

Matthieu J. Capuano

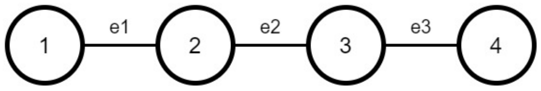
Fall 2019

1 Problem 1

Question: Write down the adjacency and incidence matrices for the following graphs.

Solution:
In order to write the adjacency and incidence matrices for each of these graphs[1], we need to label the vertices and edges first so they correspond to specific rows and columns in the matrices. We re-create and label the given graphs, and then produce their matrices. For the adjacency matrix, each row i corresponds to a vertex v_i and each column j corresponds to vertex v_j, the matrix entry is 0 if v_i and v_j do *not* have an edge between them, and 1 if they *do*. For the incidence matrix, each row i corresponds to a vertex v_i and each column j corresponds to an edge e_j, the entry is 1 if vertex v_i is one of the two endpoints of edge e_j and 0 otherwise. Note that since our graph is undirected, all our adjacency matrices are symmetric. **In all cases, the adjacency matrix is on the left and incidence matrix on the right**. In order:

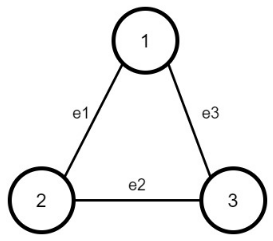
First graph:



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

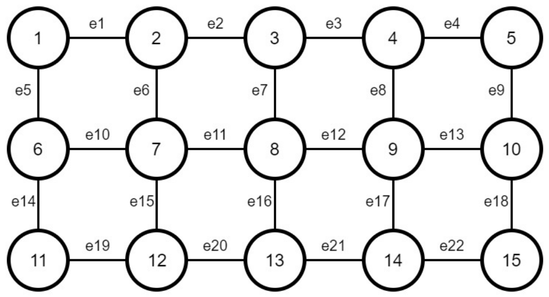
[1]Useful references:
https://en.wikipedia.org/wiki/Adjacency_matrix
https://en.wikipedia.org/wiki/Incidence_matrix

Second graph:



$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

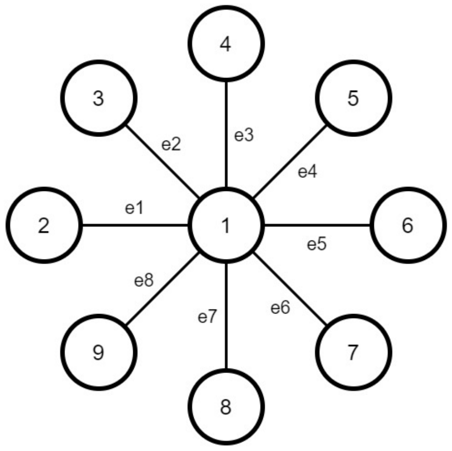
Third graph:



$$\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$$

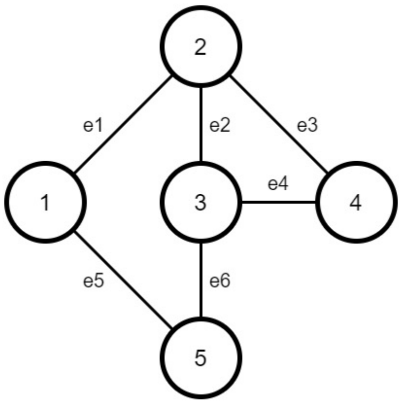
$$\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}$$

Fourth graph:



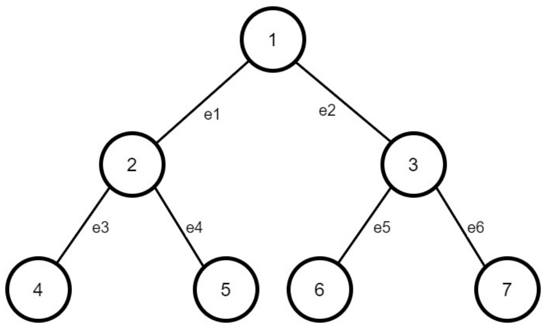
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fifth graph:



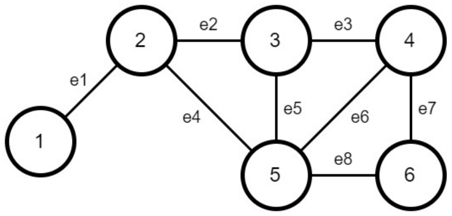
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Sixth graph:



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Seventh graph:



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

2 Problem 2

Question: Write a program to implement the A* algorithm. Use any computer language you wish. Test your algorithm to find the optimal path for a robot navigating in the following grid world, where the start state is denoted by A and the goal state is denoted by B. Black cells denote obstacles that are not feasible positions for the robot. Assume only four-cell connectivity (L-R-U-D).

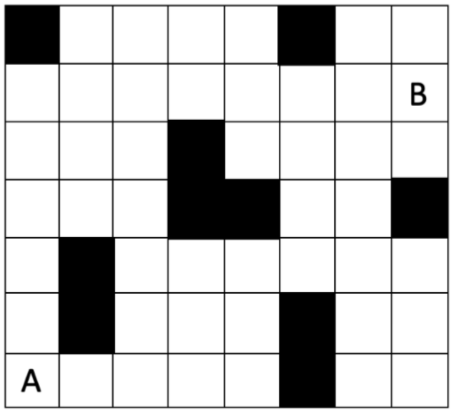


Figure 1: Problem 2 search grid.

Solution:
First, note that all of our implementation details are available at:
https://github.com/capuanomat/AE-8803-Mathematical-Foundations-of-Autonomy
This is the jupyter notebook where we implement all our solutions to this homework set. We strongly recommend you download and try it out a bit as all the classes and functions have been written in a very generic way, such that it is *very* easy to define a new Maze (or n-puzzle for the later problems) and apply the search algorithms to them. For instance, to create a new $n \times m$ maze with a list l_i of obstacle coordinates i of form $i = (x_i, y_i)$, along with start state $(s1, s2)$ and goal state (g_1, g_2), you simply need to run the line:

```
maze = MazeGrid(n, m, [(x1, y1), ..., (xi, yi)], (s1, s2), (g1, g2))
```

Then you can run Bredth-First Search, Depth-First Search, and A* on it, after having defined some heuristic function $heuristic\_function(< params >)$ with:

```
expanded_nodes_dfs = dfs(maze)
expanded_nodes_bfs = bfs(maze)
expanded_nodes_astar = a_star(maze, heuristic_function)
```

The output of which would be the set of states expanded by the algorithm while searching for the goal state.

We have also included our most important code as an Appendix to this submission for reference. Our answer here is more analytical.

The first step in solving this problem is defining a state space for the problem and a function to return neighbors of a specific state. We do so by creating a simple hashMap with (x, y) coordinates of all elements of the grid as value, and a string “_” or “█” if that entry is an obstacle as key. Our find_neighbors function looks at all neighboring states and returns them if they are not obstacles. Check appendix A.1. for this implementation, and accompanying functions.

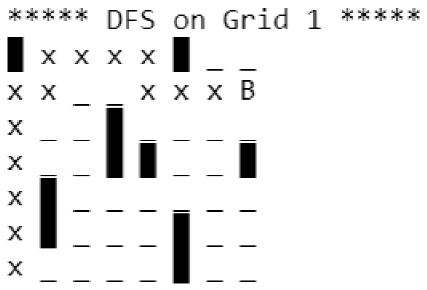
Second, implementing A* requires the definition of a heuristic. A heuristic needs to be both consistent and admissible. The admissible requirement simply states that our heuristic must never overestimate the cost to reach the goal. In this case every action is going to one of the four neighboring states, any of these operations has the same cost (1), if the shortest path to the goal is to take m_i steps when in state s_i at timestep i, the heuristic should never output any value higher than m_i. This makes the heuristic a lower bound on the actual shortest path cost to the nearest goal. This is important so we never block out parts of the search by saying that they cost more than they actually do and then ignoring them. The second requirement about consistency, also known as monotonicity (or just the “triangle inequality”), is that the heuristic is consistent if, for every node n and all its successors, n', the estimated cost of reaching the goal from n is smaller or equal to: $(step\ cost\ from\ n\ to\ n') + (estimated\ cost\ of\ reaching\ goal\ from\ n')$. Basically, if an action has cost c, then taking that action can only cause a drop in heuristic of at most c. Note that any consistent heuristic is also admissible.

For this problem, we try two heuristics: Manhattan distance and Euclidean distance. Both of these are consistent and admissible. For Manhattan distance, the value is the shortest path the agent could travel, if there were no obstacles on the map. This is admissible because it will always underestimate the cost to the goal (adding obstacles can only make the path longer). It is consistent because in this case, all our step costs are 1, and the Manhattan distance only ever changes by 1 when travelling to neighboring states. The Euclidean distance between two points is always smaller than the Manhattan distance between two points, so we know that this latter heuristic is also consistent, and thus admissible. Check Appendix 1.4. for the implementation of these heuristics.

Finally, we can implement A* as shown in Appendix 1.5. We also implement BFS and DFS[2] for comparison. We apply all three algorithms on two different

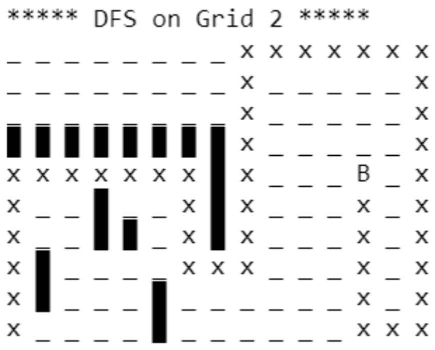
[2]Note that our implementations for all algorithms are such that the algorithm stops once it reaches a valid neighbor of the goal state, instead of when it expands a goal state. This reduces the runtime of BFS from $O(b^d)$ to $O(b^{d-1})$ where b is the average branching factor and d is the depth of the shallowest goal node because we don't have to wait for the goal state to be expanded to return, instead we can return at one shallower level when we see the goal as a neighbor.

grids: the given one and a slightly more complex grid for analytical purposes.



```
***** DFS on Grid 1 *****
█ x x x x █ _ _
x x _ _ x x x B
x _ _ █ _ _ _ _
x _ _ █ █ _ _ █
x █ _ _ _ _ _ _
x █ _ _ _ █ _ _
x _ _ _ _ █ _ _
```

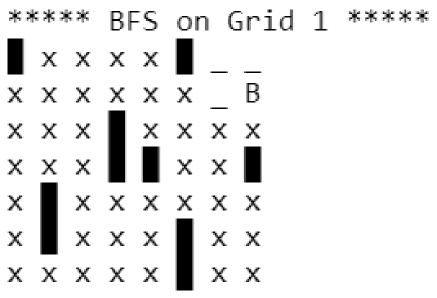
Figure 2: DFS Results on Given Grid.



```
***** DFS on Grid 2 *****
_ _ _ _ _ _ _ _ x x x x x x x
_ _ _ _ _ _ _ _ x _ _ _ _ _ x
_ _ _ _ _ _ _ _ x _ _ _ _ _ x
█ █ █ █ █ █ █ █ x _ _ _ _ _ x
x x x x x x x █ x _ _ _ B _ x
x _ _ █ _ _ x █ x _ _ _ x _ x
x _ _ █ █ _ x █ x _ _ _ x _ x
x █ _ _ _ _ x x x _ _ _ x _ x
x █ _ _ _ █ _ _ _ _ _ _ x _ x
x _ _ _ _ █ _ _ _ _ _ _ x x x
```

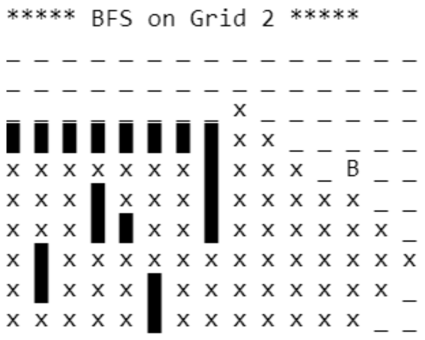
Figure 3: DFS Results on Second Grid.

The above results make sense, DFS expands a single path to it's greatest depth before backtracking one step and exploring the next path. In this case, the expansion first expands upwards then, when there are no more upward neighbors, it expands to the right. Once there are no unexplored up or right neighbors, it goes down, and then left. This is why our agent first went upwards (instead of right), then right, down, right, and up again. It also didn't turn left once it saw an opening but just continued straight to the top, before wrapping around. Note that because we check if a neighbor is already in the frontier before pushing it onto the stack, we did not push the upwards neighbor when in state (14, 0), one left of the bottom right, because it was already pushed into the frontier when in state (15, 1), so the agent continued left to (13, 0) instead. By contrast DFS expanded the states by "layers", as expected:



```
***** BFS on Grid 1 *****
█ x x x x █ _ _
x x x x x x _ B
x x x █ x x x x
x x x █ █ x x █
x █ x x x x x x
x █ x x x █ x x
x x x x x █ x x
```

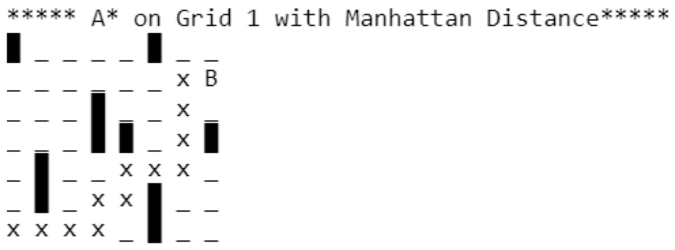
Figure 4: BFS Results on Given Grid.



```
***** BFS on Grid 2 *****
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ x _ _ _ _ _ _
█ █ █ █ █ █ █ █ x x _ _ _ _ _
x x x x x x x █ x x x _ B _ _
x x x █ x x x █ x x x x x _ _
x x x █ █ x x █ x x x x x x _
x █ x x x x x x x x x x x x x
x █ x x x █ x x x x x x x x _
x x x x x █ x x x x x x x _ _
```

Figure 5: BFS Results on Second Grid.

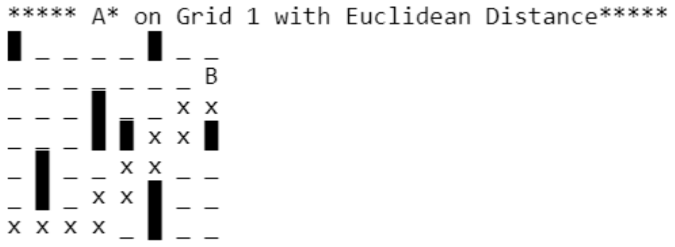
Finally, we apply our A* search algorithms with our two heuristics:



```
***** A* on Grid 1 with Manhattan Distance*****
█ _ _ _ _ █ _ _
_ _ _ _ _ _ x B
_ _ _ █ _ _ x _
_ _ _ █ █ _ x █
_ █ _ _ x x x _
_ █ _ x x █ _ _
x x x x _ █ _ _
```

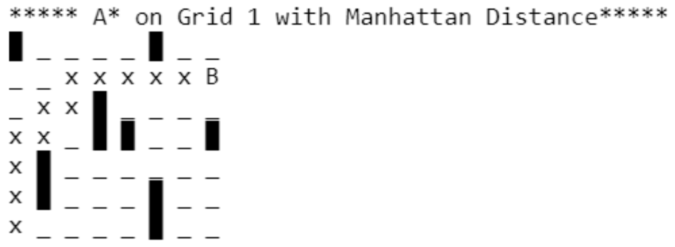
Figure 6: A* Results on Given Grid with Manhattan Distance.

Note that the Manhattan distance did not always take that exact path, since there are multiple optimal paths to the goal state. For instance, on some occasions the path take was:



```
***** A* on Grid 1 with Euclidean Distance*****
█ _ _ _ _ █ _ _
_ _ _ _ _ _ _ B
_ _ _ █ _ _ x x
_ _ _ █ █ x x █
_ █ _ _ x x _ _
_ █ _ x x █ _ _
x x x x _ █ _ _
```

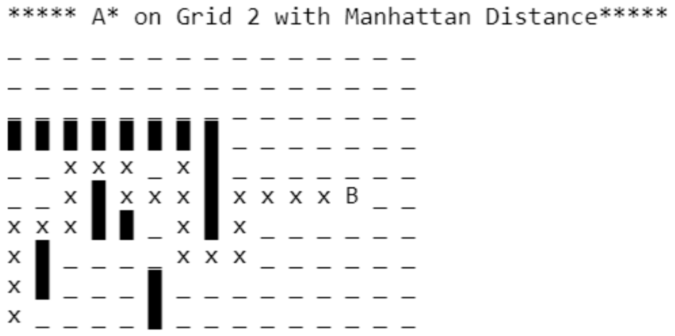
Figure 7: A* Results on Given Grid with Euclidean Distance



```
***** A* on Grid 1 with Manhattan Distance*****
█ _ _ _ _ █ _ _
_ _ x x x x x B
_ x x █ _ _ _ _
x x _ █ █ _ _ █
x █ _ _ _ _ _ _
x █ _ _ _ █ _ _
x _ _ _ _ █ _ _
```

Figure 8: A* Results on Given Grid with Manhattan Distance.

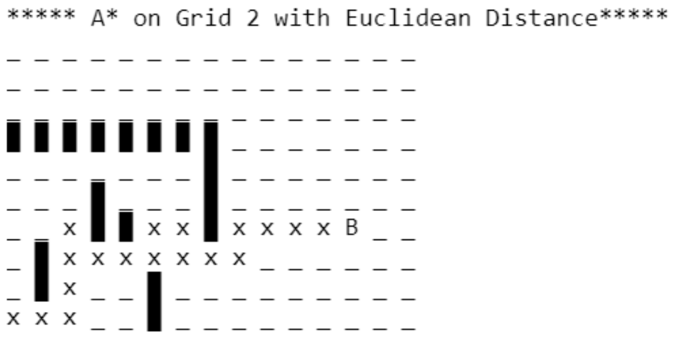
Which is equivalently optimal, since travelling between neighboring states does not have varying costs, so any two paths that take the same number of steps to the goal are equivalent in value. For the larger grid:



```
***** A* on Grid 2 with Manhattan Distance*****
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
█ █ █ █ █ █ █ █ _ _ _ _ _ _ _
_ _ x x x _ x █ _ _ _ _ _ _ _
_ _ x █ x x x █ x x x x B _ _
x x x █ █ _ x █ x _ _ _ _ _ _
x █ _ _ _ _ x x x _ _ _ _ _ _
x █ _ _ _ █ _ _ _ _ _ _ _ _ _
x _ _ _ _ █ _ _ _ _ _ _ _ _ _
```

Figure 9: A* Results on Second Grid with Manhattan Distance.

Again, these results make complete sense. In the Manhattan distance case, the agent happened to begin by travelling upwards (sometimes it begins by travelling right, both have the same associated decrease in estimated cost to the goal, 1) and only took steps that brought it closer to the goal based on our



```
***** A* on Grid 2 with Euclidean Distance*****
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
█ █ █ █ █ █ █ █ _ _ _ _ _ _ _
_ _ _ _ _ _ _ █ _ _ _ _ _ _ _
_ _ _ █ _ _ _ █ _ _ _ _ _ _ _
_ _ x █ █ x x █ x x x x B _ _
_ █ x x x x x x x _ _ _ _ _ _
_ █ x _ _ █ _ _ _ _ _ _ _ _ _
x x x _ _ █ _ _ _ _ _ _ _ _ _
```

Figure 10: A* Results on Second Grid with Euclidean Distance

heuristic unless there was no such option. At no point did it expand a node farther back in the priority queue that did not have a neighboring state closer to the goal. For instance, once the agent went up, to the right, and hit a wall, it did not go back to the start state and try to expand right, because the right neighbor of the start state is not closer to the goal than the neighbors of the state where it hit a wall.

For the Euclidean distance heuristic, it is interesting to note that the agent generally tends to go right as opposed to upwards. This is because the goal state is farther to the right than it is upwards, so moving one step right decreases the estimated cost to the goal more than moving upwarsd. I.e. in this specific case the start state (1, 1) is distance $\sqrt{(13-1)^2+(4-1)^2} \approx 12.36$ from the goal, the state to the right of it (2, 1) is $\sqrt{(13-2)^2+(4-1)^2} \approx 11.40$, and the state above it (1, 2) is $\sqrt{(13-1)^2+(4-2)^2} \approx 12.17$, so it makes sense for the agent to move right. This behavior was even stronger in cases where the goal state was moved lower and more rightwards.

3 Problem 3

Question: Show that the 8-puzzle states are divided into two disjoint sets, such that any state is reachable from any other state in the same set, while no state is reachable from any state in the other set. Devise a procedure to decide which set a given state is in, and explain why this is useful for generating random states (Hint: See "Winning Ways, For Your Mathematical Plays," by Berlekamp, E. R., Conway, J. H., and Guy, R. K. 1982, Academic Press).

Solution:
We propose approaching this problem by assigning a numerical integer value to each state based on the configuration of the board. We will show that the numerical value assigned to each possible configuration split into two distinct groups of numbers, which are a direct mapping to the disjoint sets. Let the value assigned to each state be equivalent to the total number of *displacements* of the pieces on the puzzle. What we refer to as a *displacement* is an inversion between two pieces that should be in opposite order. For example, consider the goal state below:

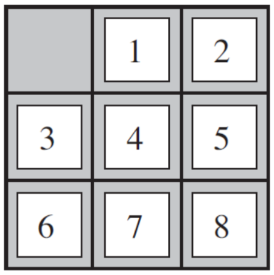


Figure 11: a goal state for the 8-puzzle

We define this as being the goal state because the empty square is on the top left, and the remaining 8 numbers are in ascending number from left to right and top to bottom. This has 0 total displacements. If we were to switch pieces 1 and 2 around, then there would be 1 total displacement because there would be one case of two numbers being in descending order instead of ascending . If we switched pieces 1 and 3 around instead, there would be 3 total displacements because we now have that 1 3 are in an incorrect position relative to each other, as would be 3 2, and 2 1. By contrast, note that if we slid pieces 1 and 2 to the left, and then piece 5 upwards, we would have two displacements only, for numbers (5, 3), (5, 4). For a more complicated example, consider starting state:

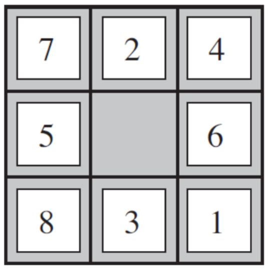


Figure 12: A starting state for the 8-puzzle

This has displacements (7, 2), (7, 4), (7, 5), (7, 6), (7, 3), (7, 1), (2, 1), (4, 3), (4, 1), (5, 3), (5, 1), (6, 3), (6, 1), (8, 3), (8, 1), and (3, 1), which gives a total of $d = 16$ total displacements. Our claim is that the parity (whether the total number of displacements d is odd or even) determines which of the two disjoint sets the state belongs in.

First, observe that moving a piece left or right cannot change the value of d (e.g. moving 5 to the right or 6 to the left above does nothing to d). Now consider vertical moves, note that such a move will only ever change the relative position of three numbers: if we move a piece down, the position of that piece and the two to the right of it are shifted (if the piece is in the middle or right column, the two "right pieces" are the two immediately after it from right to left and top to bottom, so in the case above for 2 we look at 4 and 5), if we move a piece up, the position of that piece and the two to the left(/up) are shifted. In both cases we have three possibilities, if we none of the three numbers are displaced (like in the case above for numbers (2, 4, 5), then moving the piece down will increase d by 2. E.g. Here moving 2 down would add inversions (4, 2) and (5, 2). If only one of the pieces relative to the moving piece is displaced, then moving the piece down will not change d, because it will fix the first displacement, but cause a displacement with the other, originally non-displaced number. If both of the pieces are displaced relative to the moving piece (such as if the 2 was a 7 instead), then moving that piece down would fix the two displacements and decrease d by 2. The opposite is true for moving a piece up.

So we have shown that moving pieces horizontally does not change d, but moving them vertically changes d by either +2, 0, or -2. This means that whatever our state is, we can only maintain the parity of d. Thus, the two disjoint sets correspond to all the states with d having odd parity, and those with even parity. We have shown that one cannot change the parity, so we cannot go from one set to another. This means that given a start state, we can find out right away if we can reach a given goal state by making sure their d-parities match. When generating random states, they will only be useful if they have same d-parity.

4 Problem 4

Question: Test your algorithm to the 8-puzzle. The initial condition is shown in the figure below. Use both the h1 and h2 heuristics mentioned in class.

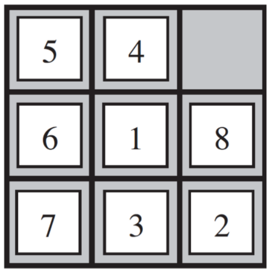


Figure 13: The given starting state for the 8-puzzle.

Solution:
We have already implemented A* in a generic way, so we now simply need to program a state space representation that has the ability to store the current state and fetch its neighbors. We do so as shown in Appendix A.1.5[3] We also implement the two heuristics h1 and h2 as shown in Appendix A.1.6. We then apply A* to this new state space with its start state, and the given goal state shown in problem 3.

Note that we know ahead of time that h2 should perform better. A quality measure for heuristics is the **effective branching factor**, b^*, which is the branching factor that a uniform tree of depth d would need to be able to contain the N nodes expanded by A^* plus one. In "Introduction to Artifical Intelligence" (Peter Norvig), they show that heuristic h2 expands far fewer nodes and has a much better (lower) branching factor than h1.

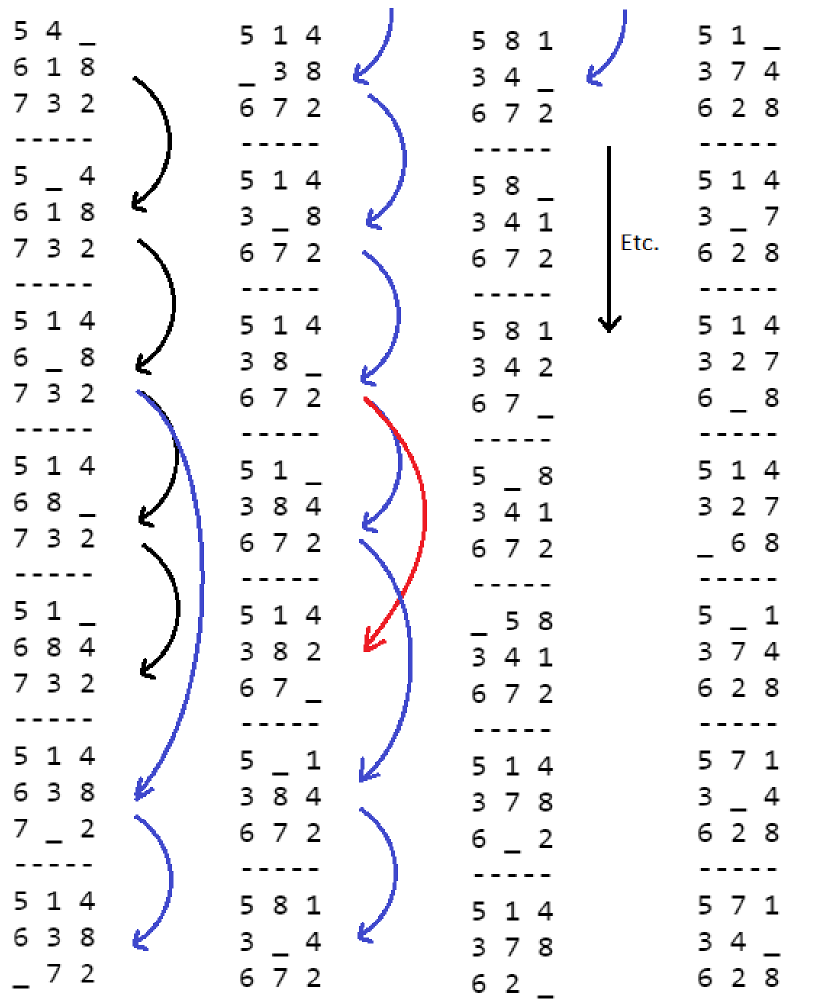
This was consistent with our results. We ran A* with both heuristics 100 times to compute the average number of expanded state with both heuristics. We obtained the following results:

- **Average number of states expanded by A* with h1:** 820.28
- **Average number of states expanded by A* with h2:** 315.71 [4]

[3]Again, we recommend you visit the jupyter notebook we produced at https://github.com/capuanomat/AE-8803-Mathematical-Foundations-of-Autonomy, if not to test it, at least to view the results more easily.

[4]A keen observer might note that the A* algorithm, given the same starting and goal states for this problem, as well as the same policy for breaking ties, should be deterministic, but that our solutions and number of states expanded vary. The reason for this is that our policy to break ties is not always the same. Python priority queues take in tuples of the priority an element should have and the element to be added to the queue, it uses the priority to determine where to push the element onto the priority queue (heap), but if two elements have the same priority, it then looks at the next element in the tuple to determine which one should go higher up. So one needs to have a consistently differentiating number as the second

Though we added functionality to display the expanded states in order, these solutions are too lengthy to show (as just explained, hundreds of states are usually expanded). Below are the first 28 states expanded during a run with h1. Note how A* will sometimes backtrack to a previous state and expand that one if it has a lower estimated cost to the goal:



```
5 4 _
6 1 8
7 3 2
-----
5 _ 4
6 1 8
7 3 2
-----
5 1 4
6 _ 8
7 3 2
-----
5 1 4
6 8 _
7 3 2
-----
5 1 _
6 8 4
7 3 2
-----
5 1 4
6 3 8
7 _ 2
-----
5 1 4
6 3 8
_ 7 2
```

```
5 1 4
_ 3 8
6 7 2
-----
5 1 4
3 _ 8
6 7 2
-----
5 1 4
3 8 _
6 7 2
-----
5 1 _
3 8 4
6 7 2
-----
5 1 4
3 8 2
6 7 _
-----
5 _ 1
3 8 4
6 7 2
-----
5 8 1
3 _ 4
6 7 2
```

```
5 8 1
3 4 _
6 7 2
-----
5 8 _
3 4 1
6 7 2
-----
5 8 1
3 4 2
6 7 _
-----
5 _ 8
3 4 1
6 7 2
-----
_ 5 8
3 4 1
6 7 2
-----
5 1 4
3 7 8
6 _ 2
-----
5 1 4
3 7 8
6 2 _
```

```
5 1 _
3 7 4
6 2 8
-----
5 1 4
3 _ 7
6 2 8
-----
5 1 4
3 2 7
6 _ 8
-----
5 1 4
3 2 7
_ 6 8
-----
5 _ 1
3 7 4
6 2 8
-----
5 7 1
3 _ 4
6 2 8
-----
5 7 1
3 4 _
6 2 8
```

Figure 14: First 28 expanded states during a run with A* using h1 heuristic.

Which I personally find quite beautiful.

element of the tuple in case of ties. In our case, we used the ID of the state we were adding (a dictionary), which is not always the same between runs. This was actually very convenient as it allowed us to test the runtime of A* without even having to vary the start and goal state!

5 Problem 5

Question: We have the following puzzle consisting of six polygonal pieces of different color. Can you use your program to put the pieces together in order to construct the shape shown at the bottom of the figure?

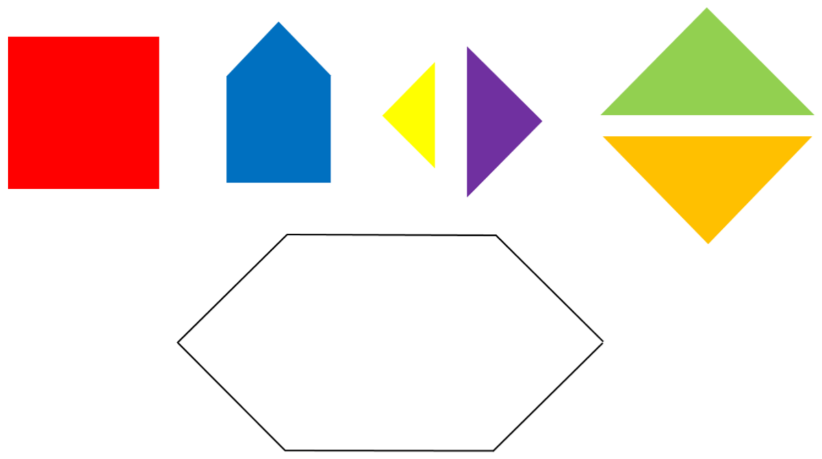


Figure 15: Problem 5 pieces and goal state.

Solution:
Since we already have generic implementations of DFS, BFS, and A*, the key challenge with this problem is defining a single state, the state space, a function to find neighbors of a given state, and a method of checking if a state is equivalent to a goal state, all of which are necessary for these algorithms. We address all of these in our implementation of the ColorGeometry class, as can be seen Appendix A.1.7. but we dive into some detail about some of these functions here, in particularly the get_neighbors function, which was the biggest challenge.

First, we choose to define a state as being a tuple of our current polygon, and the set of shapes that are still available to be added (shapes not currently part of the geometry). That way, whenever we produce a neighbor, we fetch (and remove) a shape from the set of available shapes, see how we can append it to our current geometry, and return the combined geometry if there are valid alignment. This process is more complex that it sounds. For any two polygons, if we focus on methods of combining them that only involve aligning any two edges against one another, and sliding them along that edge, then there are infinitely many ways of putting the polygons together. To simplify this process such that our state space becomes finite and manageable in such a way that we can still construct the shape of interest, we choose to focus on solely on the vertices of the polygons, and how we can align those together. Note that a polygon can be completely defined by its vertices, so this still allows us to consider the entirety of the structures. Th only limitations using this method are that two polygons can only be joined by aligning two of their vertices, and then rotating one or both shapes such that the edges neighboring those vertices are aligned. For example, we cannot place the yellow triangle at some random point along an edge of the red square, it has to be connected by a vertex. However, note that our approach always rotates one of the shapes such that two edges of the two shapes are aligned, so we do not have "dangling" pieces, i.e. where a vertex touches the edge of another object but nothing else is in contact.

Now, whenever combining two geometries, if we focus on the vertices alone, there is a **finite** number of ways that vertices can be placed in contact. If shape s_1 has M vertices, and shape s_2 has N vertices, then there are $M \times N$ possible ways of putting any two vertices from the two polygons together. Our algorithm for finding neighbors takes advantage of this by looking at every pair of edges, and checking how the adjacent faces can be aligned. At any given point, we are focusing on two vertices v_1 and v_2. We align them, then look at the their neighboring vertices v_{1p}, v_{1n}, v_{2p}, v_{2n}, where p and n stand for "previous" and "next" vertices of v_1 and v_2 in the ordered list of vertices, this just corresponds to either of the neighboring vertices. We then attempt to align the neighboring faces by computing the angles between all pairs of vertices $\{v_{1p}, v_{1n}, v_{2p}, v_{2n}\}$ and the now superposed vertices $v_1 = v_2$. We do so by using the angles to compute the 2D rotation matrix, and applying it to the piece we are adding to our current state (polygon). This gives four different possible ways of aligning the four edges (only rotation is allowed, we do not reflect the pieces), but note that two of these alignments are not valid, as they will cause the interior of the two polygons to intersect. We thus need an additional functionality (in our Python implementation, it is provided by Shapely) that can take any two polygons and check if their interiors intersect. We can use this to remove the two overlapping geometries, and only keep the two valid ones. Finally, we combine the geometries by merging their two lists of vertices while preserving order, so that neighboring vertices can still easily be found. Algorithmically:

Algorithm 1: getNeighbors

```
   Data: current state $state$
1  state_vertices = $state[0]$
2  available_shapes = $state[1]$
3  neighbors = []
4  for All shapes $s_1 \in available\_shapes$ do
5      for All vertices $v_2 \in state\_vertices$ do
6          for All vertices $v_1 \in s_1$ do
                  1. Align vertex $v_2$ of shape state_vertices (our current polygon) with vertex
                     $v_1$ of shape $s_1$ by computing their Euclidean distance and performing the
                     corresponding transformation on shape $s_1$ (the shape we are currently
                     trying to add to our polygon)
                  2. Look at both neighboring vertices (edges) of $v_1$: $v_{1p}$ and $v_{1n}$
                  3. Look at both neighboring vertices (edges) of $v_2$: $v_{2p}$ and $v_{2n}$
                  4. For each of the four possible combination of neighboring vertices (edges),
                     find the angle that would align those edges
                  5. Compute the rotation matrix from this angle
                  6. Rotate the shape $s_1$ about vertex $v_1 = v_2$ (which are the same now) to
                     produce shape $s_1^{'}$. For the four possible combinations of edges/faces,
                     there should now be four shapes $v_1^{'}$ that are aligned along a face with
                     state_vertices
              for $s_1^{'} \in$ four newly rotated shapes do
                  if $s_1^{'}$ interior intersects with state_vertices interior then
                      Reject $s_1^{'}$
                  else
                      combined_geometry = combine shapes $s_1^{'}$ and
                      state_vertices to form a new polygon by combining
                      their lists of vertices in the correct order, removing
                      any that is completely surrounded by the geometry

                      neighbors.append((combined_geometry,
                      available_shapes.remove($s_1$)
                  end
              end
7          end
8      end
9  end
10 return neighbors
```

The runtime of this algorithm is dominated by the three for loops. Note that the most complicated geometry we could produce with this method (in terms of vertices, which we focus on) would have as many vertices as all the available shapes combined. I.e. If the total number of shapes available at the beginning of the problem is S, and each shape $s_i \in S$ has m_i vertices, then the most complex polygon could have has $\sum_{i \in |S|} m_i$ vertices. Done this number by M_{sum}. We use $M_{something}$ to denote the total number of vertices in polygon *something* or set of polygons *something*. Then the runtime of our get neighbors algorithm, the most time intensive part of our implementation (with a slight abuse of notation), is $O(|S| \times M_{current\_polygon} \times M_{available\_shapes})$. An upper bound for this would be when $M_{current\_polygon} = M_{sum}$ and $M_{available\_shapes} = M_{sum}$, which is never quite possible since $M_{available\_shapes} + M_{current_polygon} = M_{all}$ at all times, so there may be a tighter upper bound, but this remains valid. So the final runtime complexity of our get_neighbors function is

$$O(|S| \times M_{all}^2)$$

In terms of defining our goal state, we do so by defining the coordinates of our goal geometry, and using Python Shapely's .equals tool for checking if two polygons are the same within a certain tolerance.

In terms of heuristics for the A* algorithm, one of our options was to compute the -to-height ratio of our current polygon + an added shape, and give it higher priority if it is closer to the -to-height ratio of the goal polygon. That way geometries with a width-to-height ratio closer to that of the goal state would be prioritized, and A* would avoid searching through too long and narrow, or too square-ish, and irrelevant polygons. This would simply be done by finding the two most distant vertices within the geometry, defining that as the width, and then finding the two most distant vertices among the remaining vertices *projected onto a line perpendicular to a line through the original two vertices.*[5]

When implementing the get_neighbors algorithm in practice, as shown in Appendix A.1.7. some challenges were encountered. Our function was able to produce some neighbors, but not the ones we were interested in, specifically, it did not always align the edges correctly (most likely a bug in the computation of the angles between vertices). So we do not have any pretty results to show, but concretely went through every aspect of the learning process and feel much more confident in our understanding of these topics. Which is the important part.

Thank you for coming to my TED Talk.

[5]We did not get the chance to implement this, unfortunately, as our get_neighbors function ran into a few issues and seemed to be priority :(