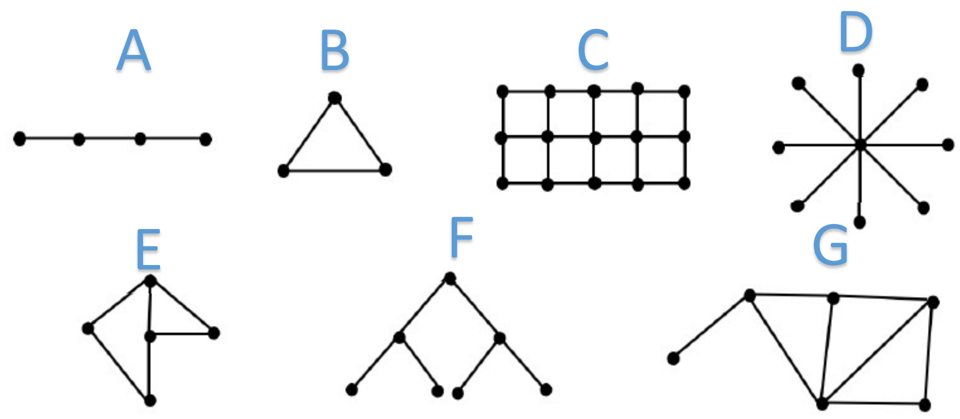
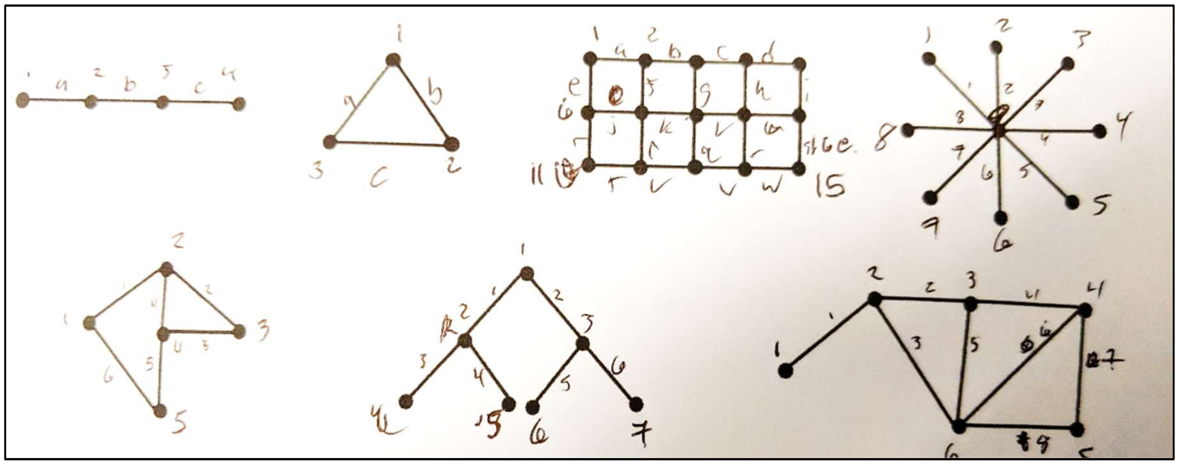
AE for AI – Homework #2

Question 1

Adjacency and incidence matrices for:





Adjacency Matrix: The adjacency matrix of G is the n × n matrix A = (aij), where aij = 1 if there is an edge between vertex i and vertex j and aij = 0 otherwise

Incidence Matrix: The incidence matrix of G is an n × m matrix B = (bik), where each row corresponds to a vertex and each column corresponds to an edge such that if ek is an edge between i and j, then all elements of column k are 0 except bik = bjk = 1.

A:

$$\text{Adjacency: } \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{Incidence: } \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

B:

$$\text{Adjacency: } \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \text{Incidence: } \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

C:

$$\text{Adjacency: }\begin{bmatrix}
0&1&0&0&0&1&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&0&1&0&0&0&0&0&0&0&0\\
0&1&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&0&1&0&1&0&0&0&1&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&1&0&0&0&0&0\\
1&0&0&0&0&0&1&0&0&0&1&0&0&0&0\\
0&1&0&0&0&1&0&1&0&0&0&1&0&0&0\\
0&0&1&0&0&0&1&0&1&0&0&0&1&0&0\\
0&0&0&1&0&0&0&1&0&1&0&0&0&1&0\\
0&0&0&0&1&0&0&0&1&0&0&0&0&0&1\\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&1&0\\
0&0&0&0&0&0&0&0&1&0&0&0&1&0&1\\
0&0&0&0&0&0&0&0&0&1&0&0&0&1&0
\end{bmatrix}$$

$$\text{Incidence: }\begin{bmatrix}
1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1
\end{bmatrix}$$

D:

$$\text{Adjacency: }\begin{bmatrix}
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
1&1&1&1&1&1&1&1&0
\end{bmatrix}\quad\text{Incidence: }\begin{bmatrix}
1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0\\
0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1\\
1&1&1&1&1&1&1&1
\end{bmatrix}$$

E:

$$\text{Adjacency: }\begin{bmatrix}
0&1&0&0&1\\
1&0&1&1&0\\
0&1&0&1&0\\
0&1&1&0&1\\
1&0&0&1&0
\end{bmatrix}\quad\text{Incidence: }\begin{bmatrix}
1&0&0&0&0&1\\
1&1&0&1&0&0\\
0&1&1&0&0&0\\
0&0&1&1&1&0\\
0&0&0&0&1&1
\end{bmatrix}$$

F:

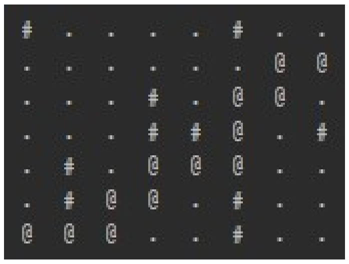
$$\text{Adjacency: }\begin{bmatrix}
0&1&1&0&0&0&0\\
1&0&0&1&1&0&0\\
1&0&0&0&0&1&1\\
0&1&0&0&0&0&0\\
0&1&0&0&0&0&0\\
0&0&1&0&0&0&0\\
0&0&1&0&0&0&0
\end{bmatrix}\quad\text{Incidence: }\begin{bmatrix}
1&1&0&0&0&0\\
1&0&1&1&0&0\\
0&1&0&0&1&1\\
0&0&1&0&0&0\\
0&0&0&1&0&0\\
0&0&0&0&1&0\\
0&0&0&0&0&1
\end{bmatrix}$$

G:

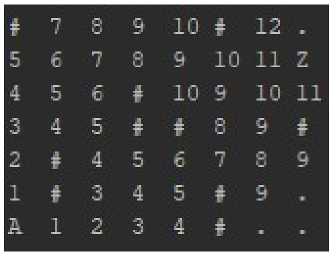
$$\text{Adjacency: } \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \quad \text{Incidence: } \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Question 2

The A* program was developed with insight from RedBlobGames (https://www.redblobgames.com/pathfinding/a-star/implementation.html). This was chosen to after investigating efficient data storage methods in Python. This implementation of A* utilizes a data structure called PriorityQueue which utilizes binary heaps for speed.



```
#  .  .  .  .  #  .  .
.  .  .  .  .  .  @  @
.  .  .  #  .  @  @  .
.  .  .  #  #  @  .  #
.  #  .  @  @  @  .  .
.  #  @  @  .  #  .  .
@  @  @  .  .  #  .  .
```



```
#  7  8  9  10 #  12 .
5  6  7  8  9  10 11 Z
4  5  6  #  10 9  10 11
3  4  5  #  #  8  9  #
2  #  4  5  6  7  8  9
1  #  3  4  5  #  9  .
A  1  2  3  4  #  .  .
```

Figure 1: Resulting Path Traced by '@' Symbol with '#' denoting walls and Cost Map

```
from algorithms import *

# Define
grid = SquareGrid(width=8, height=7)
WALLS = [from_id_width(id, width=8) for id in
[0,5,19,27,28,31,33,41,45,53]]

grid.walls = WALLS

start = (0,6)  # Column, Row (indexing from 0)
goal = (7,1)
wtd_grid = GridWithWeights(width=8, height=7)
wtd_grid.walls = [(0, 0), (5, 0), (3, 2), (3, 3), (4, 3), (7, 3), (1,
4), (1, 5), (5, 5), (5, 6)]
path = a_star_search(wtd_grid, start, goal, debug=True)
```

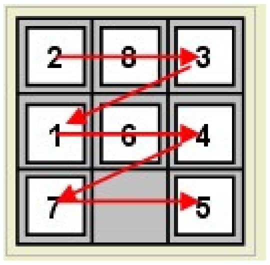
In addition, the number of search steps and search time are calculated when the debug option is set to true. The problem resulted in results similar to the following:

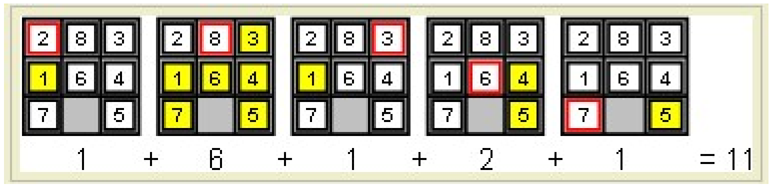
Solution after 33 search iterations and 290.1820 microseconds!

Question 3

The 8-puzzle, otherwise known as Loyd's Eight, is a version of Sam Loyd's Fifteen puzzle. The graph of the puzzle can be shown to be a bipartite, meaning there exists two disjoint sets of vertices that can only be achieved if starting in the respective set. Therefore, there exists two potential solutions to the 8-puzzle, but the potential to achieve these goal states depend on the starting position.

The proof involves defining an inversion at each grid cell as the number of following grid cells which are lower in value. An example is shown, using graphics from (http://jinchengchen.blogspot.com/2009/07/8-puzzle-algorithm.html):





Then, two disjoint sets are seen by showing that the N Mod 2 of this resulting value is invariant under any legal move. Further investigation results in two goal states which can be achieved, correlating to the result of N Mod 2 of either 0 (goal state B) or 1 (goal state A)

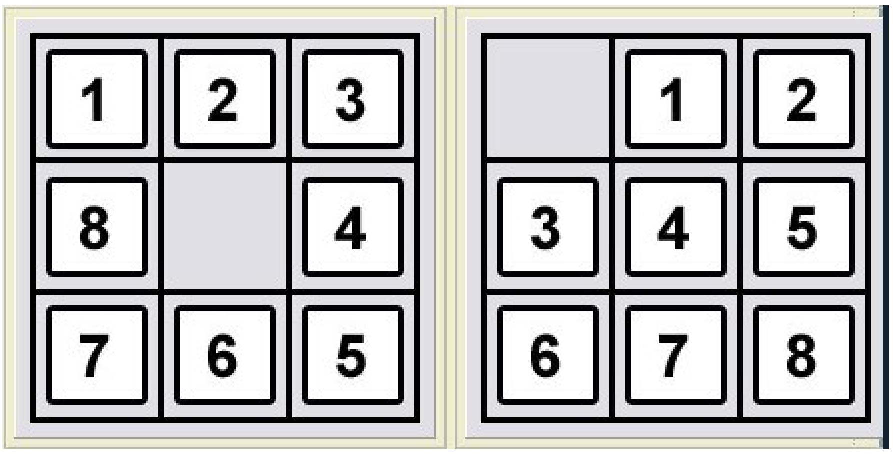


Figure 2: Goal States A and B

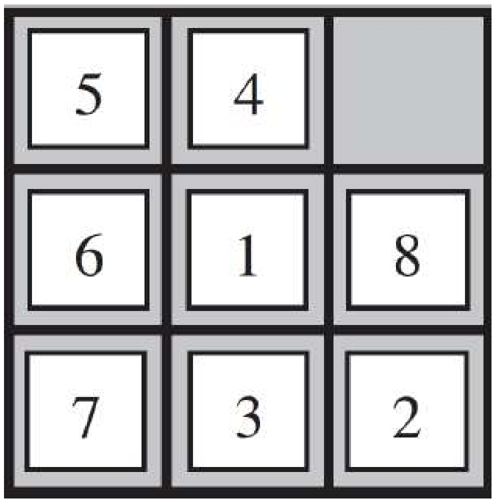
A programming procedure was developed to determine the goal set (A or B) that can be achieved from a given start state. The procedure involves vectorising the matrix in the counting order as shown in previous figures. Then, for each index in the array, the number of inversions are calculated by looping through the larger indices of the vector. Lastly, the inversions are summed and N Mod 2 determines whether goal state A or B is achievable. An example is shown below:

```
>>> test_puzzle = [[1,2,3],[8,'_',4],[7,6,5]] # GOAL STATE a

Count = 7, Goal state = A

>>> test_puzzle = [[1,2,3],[4,5,6],[7,8,0]] # GOAL STATE B

Count = 0, Goal state = B
```

This method is useful for generating random states and for integrating into a graph-based solver because it prevents cases of unachievable states and never-ending searches.

Question 4

The example puzzle is given:



The program from Question 3 results in the following results:

Inversions = 16, Goal State = B

In order to solve the problem, graph search code for a similar puzzle problem was modified to use the eight-puzzle states and the appropriate heuristics.

Two heuristics were used which have proven appropriate and successful in the past. Heuristic H1 is the count of how many values are in the incorrect spot. The maximum being a value of eight. Heurist H2 is the sum of each incorrect value's manhattan-distance to the correct spot.

H1: Total numbers in incorrect spot

The result is shown below:

```
_ 1 2

3 4 5

6 7 8

The current cost to go is: 0

The current depth of tree is: 24

Total nodes expanded: 23628

Time taken: 188.44 seconds
```

H2: Sum of manhattan-distances of number locations

The result is shown below:

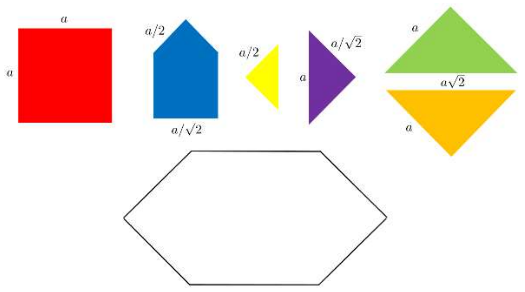
```
_ 1 2

3 4 5

6 7 8

The current cost to go is: 0

The current depth of tree is: 24

Total nodes expanded: 2413

Time taken: 1.75 seconds
```

Question 5

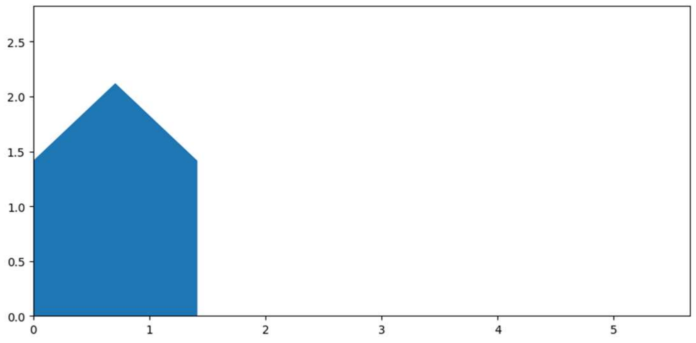
The polygon puzzle problem requires abstraction to be able to solve via a graph-based solver. The problem formulation is as follows:

Initial State and State Space

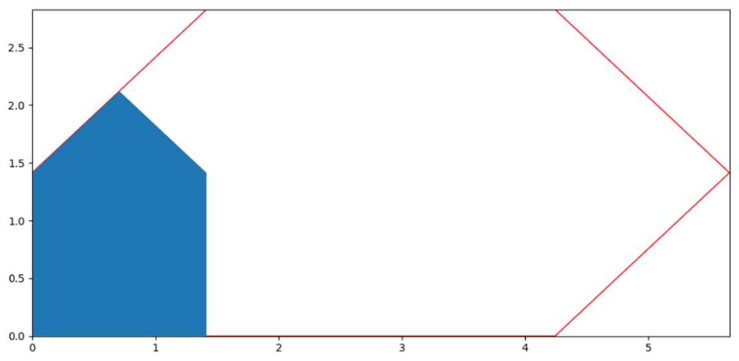
There exists six polygon pieces. Each piece has dimensions as seen in the following figure.



Each piece can be defined in a local coordinate axes by the location of the vertices.



Similarly, the target shape can be defined as a grid:



In order to achieve the target state and be able to scale with the sizing parameter, *a*, the grid is defined as starting from (0,0), ending at $\left(\frac{8}{2\sqrt{2}},\frac{4}{2\sqrt{2}}\right)*a$ with steps of $\left(\frac{1}{2\sqrt{2}}\right)*a$.

A node of the graph representing a state can be defined by a tuple with each polygon, the vectorized grid index the corner is attached to, and normalized angle of the piece. For example:

[{*Piece, index, angle*},

{1,15,0.25}

{6,42,0.75}

Piece: (1,2,3,4,5,6)

Index: (1,2, …, 44,45)

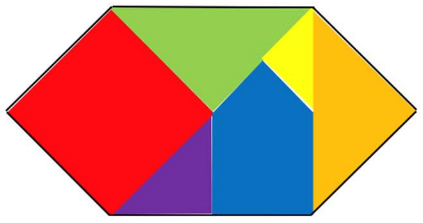
Angle: (0,0.125, …, 0.875,1.0)

Transition Model and Path Cost

The transition from one state to another can be expanded as children in a graph. The transformations include choosing a piece, moving to another index, and rotating by an angle. A valid state is one in which all the pieces are within the bounds of the polygon grid. The cost for each placement will be one level of the graph search, however a heuristic is needed for fast exploration. For instance, Astar can be used with the heuristic being the amount of area in the polygon that is not covered.

Goal Test

The goal state is achieved when there is a feasible state where the are not covered is zero. The goal state can be seen in the following figure.



Appendix

The code for this work can be found at: https://github.com/calebh94/path-planning.

Any questions can be sent to Caleb Harris at caleb.harris94@gatech.edu.