# Introduction to Artificial Intelligence

Morgan Benavidez

September 18th, 2022

## 1 Homework 2

1. Please use your own words to define following concepts :

   (a) **State** - A state is a configuration (or representation) of elements in a problem at a given moment.

   (b) **State Space** - The state space is the complete collection (or set) of all the various states that can be achieved by a problem.

   (c) **State Graph** - A state graph is the state space represented by a graph. The nodes are individual states and the edges are actions.

   (d) **Successors** - Successors are the possible states following a given state.

   (e) **Successor Function** - The successor function is what determines the number of states and which states are the successors of a given state. It implicitly represents all the actions that are possible in each state. Only the results of the actions (successor states) and their associated costs are returned by the successor function.

   (f) **Search Node** - A search node is a bookkeeping data structure used to represent the search tree. The same state can be represented multiple times in different search nodes if they can be reached by different paths.

   (g) **Node Expansion** - Node expansion is when you call the successor function for a node and generate a child node for each state returned by the successor function.

   (h) **Fringe** - The fringe is the set of all the nodes that haven't been expanded yet. This is implemented as a Queue. The ordering of the nodes in this data structure defines the search strategy.

2. What is purpose of a search tree? What is relationship between a state vs. a search node?

   The purpose of a search tree is to find a path to take you from the initial state to the goal state whenever one exists. This can also be optimized to find the minimum-cost path and can measure the time and memory it takes to find the solution as well.

   A state is represented in a search node, but a state is unique and search nodes are not. The same state can be represented multiple times in different search nodes if they can be reached by different paths. In fact, a search tree can be infinite, even when state space is finite if states are allowed to be revisited. Search nodes belong to a specific search path, where as a state does not.

3. When moving the empty tile up from S to S' in Figure 1:

   (a) Show Permutation Inversion relationships for tiles x1, x2, x3, and x4, before and after the empty tile movement for the following condition: (1) $x1 > x2 > x3 > x4$; (Tip: check the relationship between $n_{x1}$ and $n\prime_{x1}$)

   (b) Explain why the total Permutation Inversions from S to S' is "N mod 2" invariant (including the row number of the empty tile).

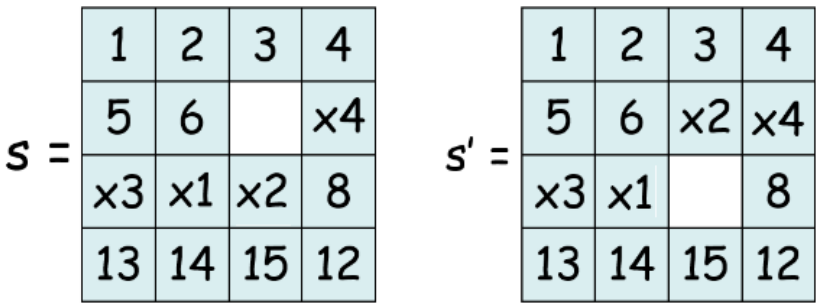
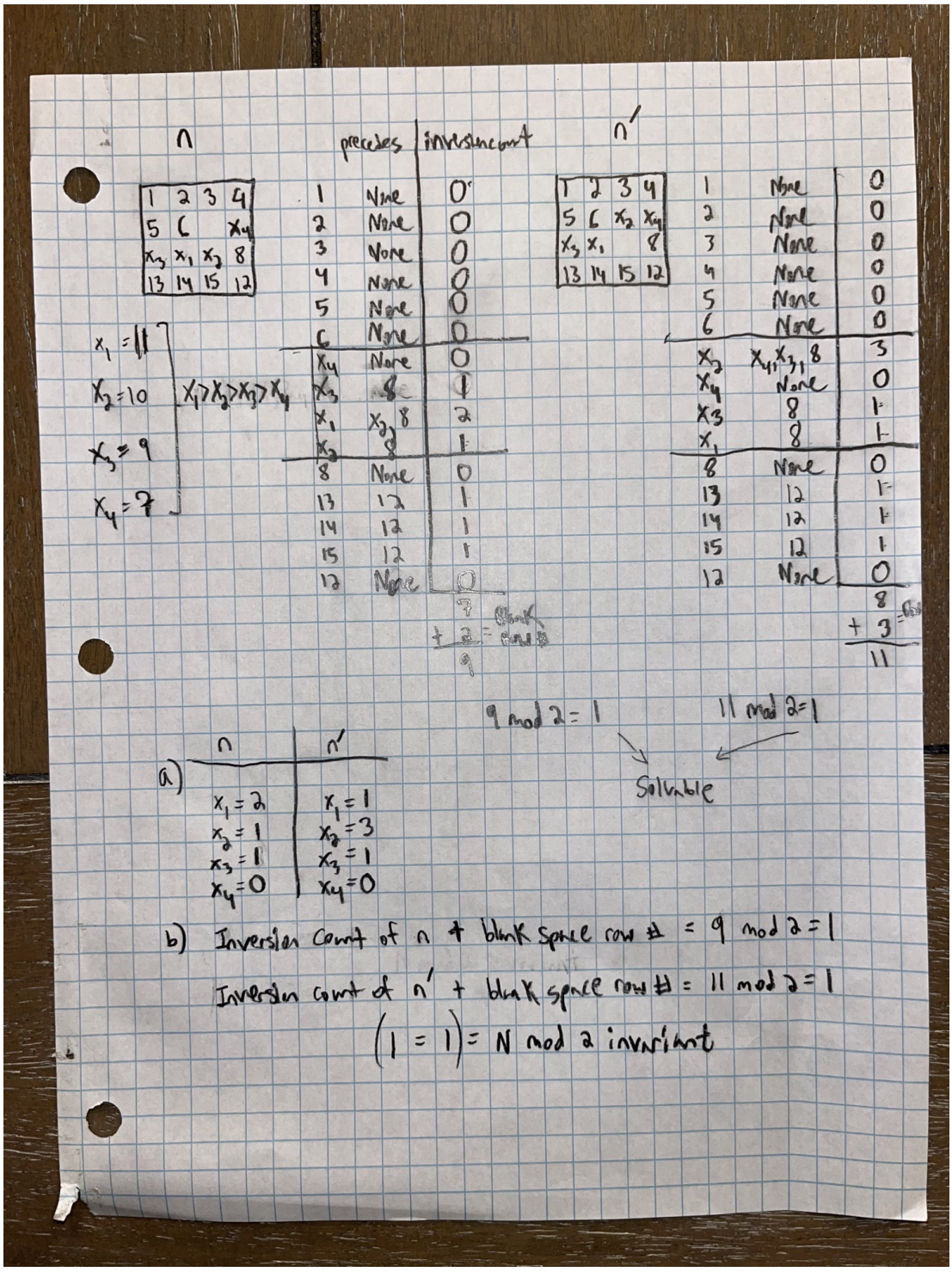



Figure 1


n
n'
precedes
inversion count
a)
b) Inversion count of n + blank space row # = 9 mod 2 = 1
Inversion count of n' + blank space row # = 11 mod 2 = 1
(1 = 1) = N mod 2 invariant
Solvable

4. In Figure 2, the left panel shows the Initial State, and the right panel shows the Goal State. Is the goal state reach from the given initial state? Explain why.

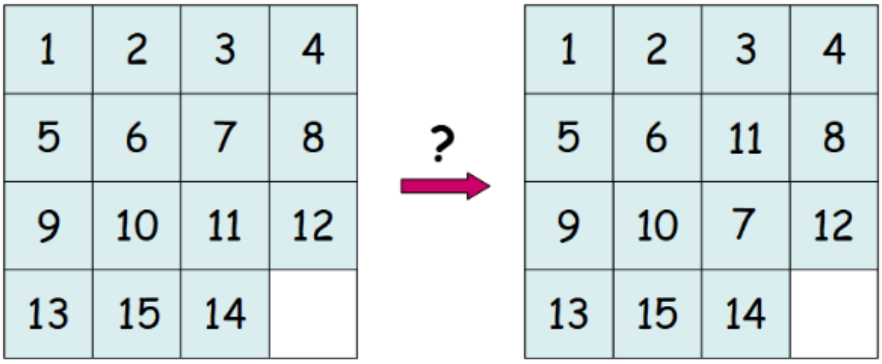
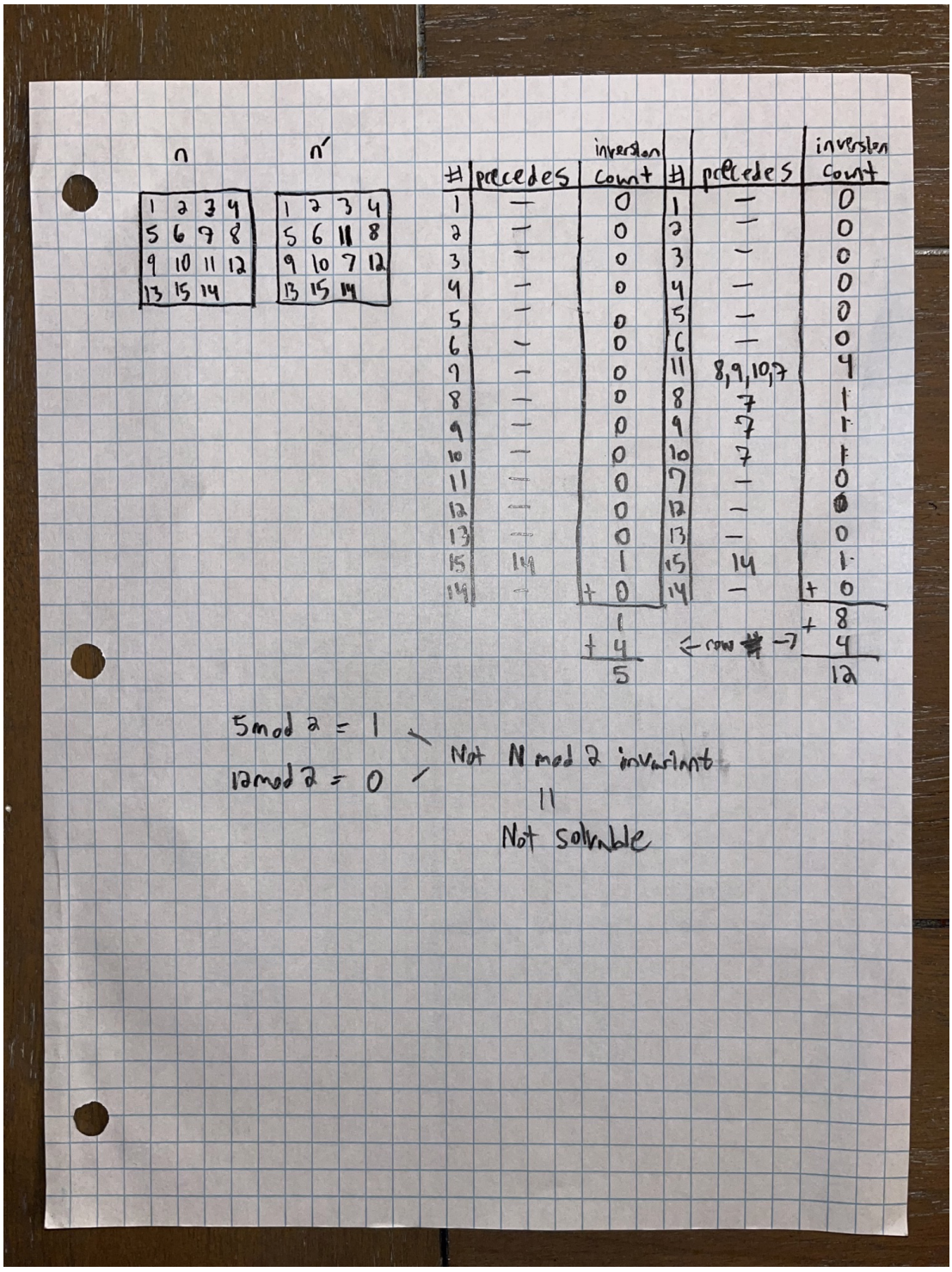



Figure 2

No, the goal state can't be reached from the initial state because they are not "N mod 2 invariant". I've included the proof for this on the next page.


n
1 2 3 4
5 6 7 8
9 10 11 12
13 15 14
n'
1 2 3 4
5 6 11 8
9 10 7 12
13 15 14
# | precedes | inversion count
1 — 0
2 — 0
3 — 0
4 — 0
5 — 0
6 — 0
7 — 0
8 — 0
9 — 0
10 — 0
11 — 0
12 — 0
13 — 0
15 14 1
14 — + 0
1
+ 4
5
# | precedes | inversion count
1 — 0
2 — 0
3 — 0
4 — 0
5 — 0
6 — 0
11 8,9,10,7 4
8 7 1
9 7 1
10 7 1
7 — 0
12 — 0
13 — 0
15 14 1
14 — + 0
+ 8
⇐ row # ⇒ 4
12
5 mod 2 = 1
12 mod 2 = 0
Not N mod 2 invariant
||
Not solvable

5. The goal of the 4-queens problem is to place 4 queens on a 4 rows 4 columns chessboard, such that no two queens attach each other. One solution to the 4-queens problem is shown in Figure 3. Please design a search based solution to solve the 4-queens problem. Your solution must include following four components

   (a) Define State.

   The states in this scenario are the various versions of the chess board every time a Queen is added to the board.

   (b) Define successor function.

   The successor function will determine the number of states and which states are the successors of a given state. It represents all the actions that are possible in each state. My successor function will weed out future Queen placements that violate the rules of being in the same column, row or diagonal of a previously placed Queen.

   (c) Calculate total number of states, based on the defined successor function.

   In my Python implementation, I check to see if there are any Queens in the column, positive diagonal and negative diagonals before placing a Queen. This immediately limits the number of states in my implementation. The total number of states in my implementation is 17 (this includes the state of having an empty board). You can see this demonstrated in the next two figures.

   (d) Show state graph (if state graph is too large, you can just show a portion of the graph with minimum 10 nodes).

   Please see the next figure I drew on graph paper to depict my state graph.

   (e) Show how to find solutions from state graph.

   Please see the next following figure that shows my Python code for N-Queens (it can solve for other values of N as well) and how I found my solutions by only placing Queens where no Queen had previously been placed in the same column, row or diagonal.

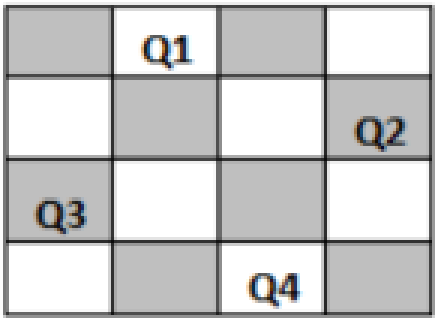
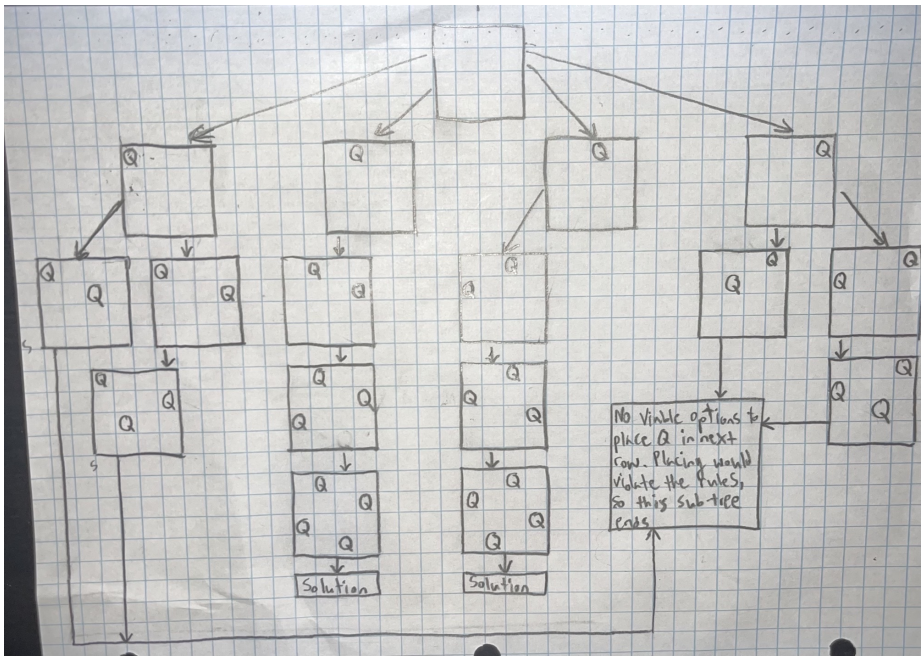

Q1
Q2
Q3
Q4

Figure 3


Solution
Solution
No viable options to place Q in next row. Placing would violate the rules, so this sub-tree ends

```python
def solveNQueens(n):
    # Initialize the sets for columns, positive diagonals, negative diagonals, list
    # for solutions
    col = set()
    posDiag = set() # (r+c)
    negDiag = set() # (r-c)
    solutions = []
    # Create the starting board of n x n matrix for "chess board"
    board = [["."]*n for i in range(n)]
    print(board)

    def successorFunction(r):
      # This is for when you have completed all rows in chess board and the row
      # index now equals n
      if r == n:
        copy = ["".join(row) for row in board]
        # Appends a copy of the solution we've reached.
        solutions.append(copy)
        print('Solution Found!')
        # Returns in order to go back up the branch to the root.
        return
      # This is iterating over 0 through 1 less than n (every column) and checking
      # if the column has already been used, if the positive diagonal has already
      # been used, or the negative diagonal has already been used. If any of them
      # have been used, it will jump to the next column value in this row.
      # If none of them have been used, it will jump to the next comment.
      for c in range(n):
        if ((c in col) or ((r+c) in posDiag) or ((r-c) in negDiag)):
          continue

        # If you jumped to this comment, then the current column values haven't
        # been assigned to any of the 3 sets and we will now add them.
        col.add(c)
        posDiag.add(r+c)
        negDiag.add(r-c)
        # After we've added these values, we will now assign Q to this column
        # in this row.
        board[r][c] = "Q"
        print('edited Board')
        print(board)

        # We will now recursively call the successor function and increment the row
        successorFunction(r+1)

        # When the previous recursive call completes, these will clear the sets of
        # values related to that specific branch.  When it hits the root, it will begin
```

```
        # the iterative process of the next branch.
        print('begin removing')
        col.remove(c)
        posDiag.remove(r+c)
        negDiag.remove(r-c)
        board[r][c] = "."

    # Initial call to the successorFunction
    successorFunction(0)
    # Returns the complete list of solutions
    return solutions

# Change the value being passed into solveNQueens in order to change the size of
# "chess board" and number of Queens
x = solveNQueens(4)
count = 0
for i in x:
  count+=1
  print(i)

print('Number of Solutions: ' + str(count))
```

```
[['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['Q', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['Q', '.', '.', '.'], ['.', '.', 'Q', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
begin removing
edited Board
[['Q', '.', '.', '.'], ['.', '.', '.', 'Q'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['Q', '.', '.', '.'], ['.', '.', '.', 'Q'], ['.', 'Q', '.', '.'], ['.', '.', '.', '.']]
begin removing
begin removing
begin removing
edited Board
[['.', 'Q', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', 'Q', '.', '.'], ['.', '.', '.', 'Q'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', 'Q', '.', '.'], ['.', '.', '.', 'Q'], ['Q', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', 'Q', '.', '.'], ['.', '.', '.', 'Q'], ['Q', '.', '.', '.'], ['.', '.', 'Q', '.']]
Solution Found!
```

```
begin removing
begin removing
begin removing
begin removing
edited Board
[['.', '.', 'Q', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', '.', 'Q', '.'], ['Q', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', '.', 'Q', '.'], ['Q', '.', '.', '.'], ['.', '.', '.', 'Q'], ['.', '.', '.', '.']]
edited Board
[['.', '.', 'Q', '.'], ['Q', '.', '.', '.'], ['.', '.', '.', 'Q'], ['.', 'Q', '.', '.']]
Solution Found!
begin removing
begin removing
begin removing
begin removing
edited Board
[['.', '.', '.', 'Q'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', '.', '.', 'Q'], ['Q', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
edited Board
[['.', '.', '.', 'Q'], ['Q', '.', '.', '.'], ['.', '.', 'Q', '.'], ['.', '.', '.', '.']]
begin removing
begin removing
edited Board
[['.', '.', '.', 'Q'], ['.', 'Q', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]
begin removing
begin removing
['.Q..', '...Q', 'Q...', '..Q.']
['..Q.', 'Q...', '...Q', '.Q..']
Number of Solutions: 2
```

6. Design a search based solution to help a robot navigate from starting location (red dot) to its destination (green dot), where the object in the middle is the obstacle (assume environment is static).

   (a) Define State.

   I made a graph implementation of this problem in Python. I included a drawing of the graph that I built along with the code. I didn't include weights for the graph, so every move costs the same. Therefore I didn't implement an ability to choose paths based off of the cost of an action. The states in this model are every point in the graph that the robot can visit.

   (b) Define successor function.

   The successor function breaks the problem into a grid pattern overlay of the map. I then use depth first search to find all the possible paths. I select the minimum from the list of all paths and calculate for its length. To represent the cost of each action (move) I subtract 1 from the total length which means the cost for each action is 1. If a goal is passed to the function, it will attempt to find it and if not present in graph or can't be reached, the program will print(Goal cannot be reached).

   (c) Calculate total number of states, based on the defined successor function.

   The total number of my states is 23 as demonstrated in my state graph. J cannot be reached and so is not included as a possible state in my calculation.

   (d) Show state graph.

   My image has a different goal state drawn on it, but my code will work for any initial and goal state you give it. I tested it multiple times for different goal states, I just happened to use this one, that's why the goal state is different than the drawing. I could have just as easily made it R or S on my graph to make it look like the picture we were given, but had already made the drawing. But the functionality is all the same.

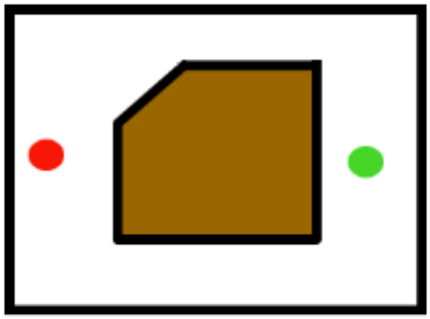
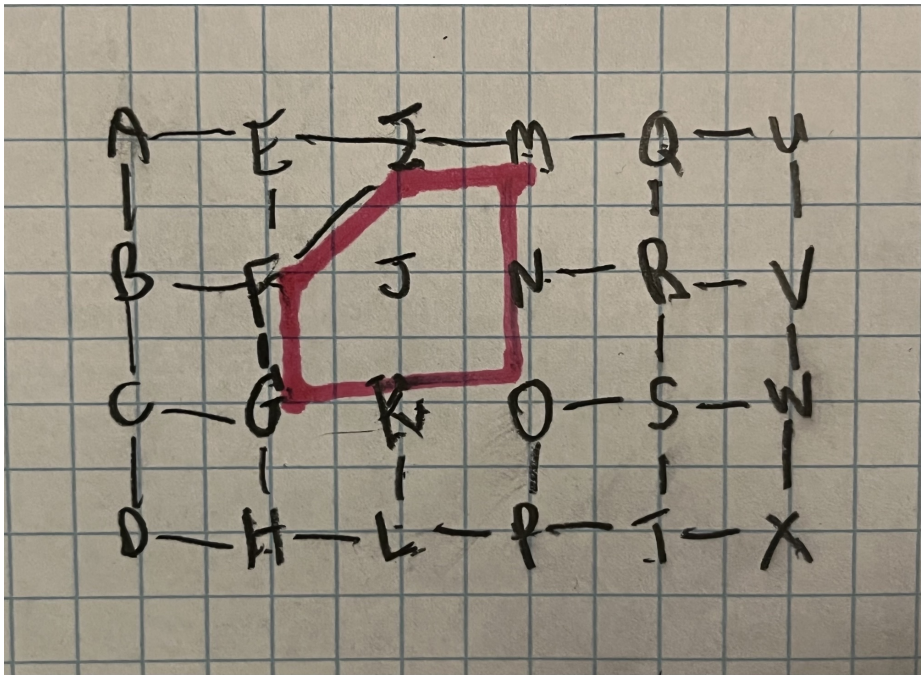
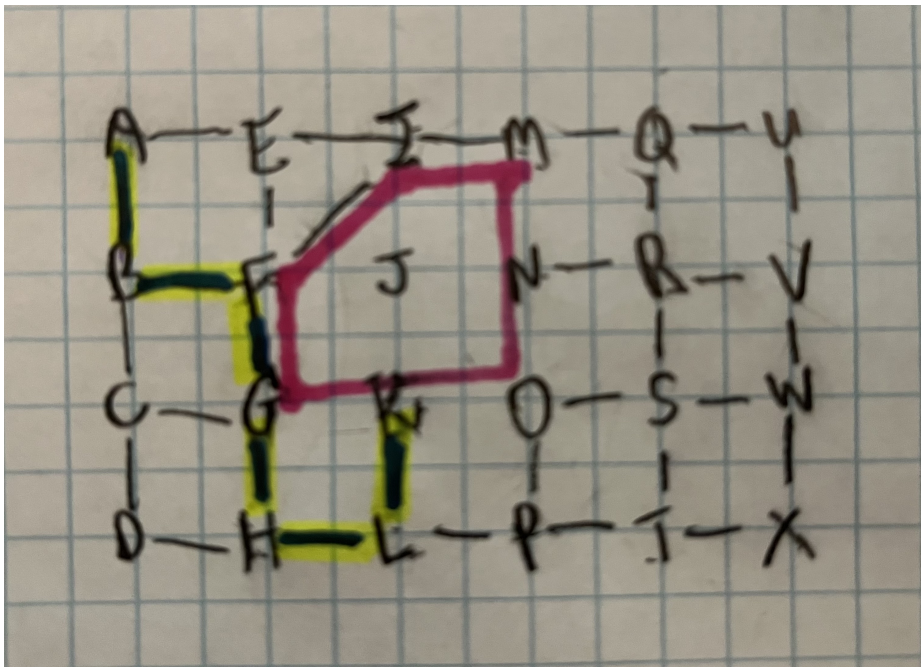


Figure 4




A
E
I
M
Q
U
F
J
N
R
V
C
G
K
O
S
W
D
H
L
P
T
X

```python
graph = {'A': set(['B', 'E']), 'B': set(['A', 'C', 'F']), 'C': set(['B', 'D', 'G']),
         'D': set(['C', 'H']), 'E': set(['A', 'F', 'I']), 'F': set(['B', 'E', 'I', 'G']),
         'G': set(['C', 'F', 'H']), 'H': set(['D', 'G', 'L']), 'I': set(['E', 'F', 'M']),
         'J': set([]), 'K': set(['L']), 'L': set(['H', 'K', 'P']), 'M': set(['I', 'Q']),
         'N': set(['R']), 'O': set(['P', 'S']), 'P': set(['L', 'T']), 'Q': set(['M', 'U', 'R']),
         'R': set(['N', 'Q', 'V', 'S']), 'S': set(['O', 'R', 'W', 'T']), 'T': set(['P', 'S', 'X']),
         'U': set(['Q', 'V']), 'V': set(['R', 'U', 'W']), 'W': set(['S', 'V', 'X']), 'X': set(['T', 'W'])
         }

# Search that will search all the paths
def dfs_paths(graph, start, goal):
    totalPaths = []
    stack = [(start, [start])]

    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                #print(path + [next])
                totalPaths.append(path + [next])
                #yield path + [next]
            else:
                stack.append((next, path + [next]))

    return totalPaths

# Create a variable to hold the list of paths generated by dfs_paths
# Pass in 3 parameters to dfs_paths
# Graph to traverse, initial state and goal state.
totalPaths = dfs_paths(graph, 'A', 'K')
print('Every Path: ')
print(totalPaths)
print('Total number of paths: ' + str(len(totalPaths)))
shortestPath = min(totalPaths, key=len)
print('Shortest Path: ' + "-".join(shortestPath))
print('Shortest Path Length: ' + str(len(shortestPath)))
print('Shortest Path Cost: ' + str(len(shortestPath)-1))

```

```
Every Path:
[['A', 'B', 'F', 'G', 'H', 'L', 'K'], ['A', 'B', 'F', 'G', 'C', 'D', 'H', 'L', 'K'], ['A', 'B', 'F', 'I', 'M', 'Q'
Total number of paths: 124
Shortest Path: A-B-F-G-H-L-K
Shortest Path Length: 7
Shortest Path Cost: 6
```

**For all programming tasks, please submit the Notebook as html or pdf files for grading (your submission must include scrips/code and the results of the script).** If you are not familiar with Python programming (and want to use Python for the coding tasks), please check Python Plotting notebook and Python Simple Analysis notebook posted in the Canvas, before working the coding tasks.

For each sub-task, please use task description (requirement) as comments, and report your coding and results in following format:

```
# List as FIFO (first in, first out)
stack = ["a", "b", "c"]

# add an element to the end of the list
stack.append("e")
stack.append("f")
print(stack)
```

```
['a', 'b', 'c', 'e', 'f']
```

7. Coding problem.

   (a) Define a "node" class (using any programming language). Each node should have a parent and a state, the node should also have one associate variable to record the path cost from the root node to the current node.

   (b) The node class should support a function to compare whether two nodes are the same or not (i.e., whether they correspond to the same state).

   (c) Create a FIFO fringe, to insert at least five nodes, and pop out all nodes until the fringe is empty. Clearly show that the nodes were popped out in a FIFO order.

   (d) Create a LIFO fringe, to insert at least five nodes, and pop out all nodes until the fringe is empty. Clearly show that the nodes were popped out in a FIFO order.

   I created class for Node, FIFO and LIFO and had them all interact in the main function. There is also a function in the Node Class that can be passed self and another node to compare if they are the same state or not. I also wrote the code to clearly display the FIFO and LIFO ordering for input and outputs.

```python
from queue import Queue
# Node Class

class Node:
  # This function will instantiate the node object and assign the values passed
  # into the class to the node object
  def __init__(self, state, parent=None, pathCost=1):
    self.state = state
    self.parent = parent
    self.pathCost = pathCost

  # This function can have another node passed to it and will compare the
  # node that is calling it with the node being passed in.
  def compareNodes(self, comparisonNode):
    # This line will reference the state variables in both nodes and return True
    # if they are the same.
    if (self.state == comparisonNode.state): return True
    # If they are not the same, the function will return False.
    else: return False

# Class for FIFO (Queue)
class FIFO:

  def __init__(self, Queue):
    self.Queue = Queue

  # This functionality will remove from the Queue
  def removeFromQueue(self):
    if (len(self.Queue) > 1):
      ObjectOut = self.Queue[0]
      self.Queue[1].parent = None
      for i in range(0, len(self.Queue)-1):
        self.Queue[i] = self.Queue[i+1]
    else:
      ObjectOut = self.Queue[0]
      for i in range(0, len(self.Queue)-1):
        self.Queue[i] = self.Queue[i+1]

    del self.Queue[-1]
    return ObjectOut, self.Queue

# Class for LIFO (Stack)
```

```
43 class LIFO:
44
45   def __init__(self, Stack):
46     self.Stack = Stack
47
48   def removeFromStack(self):
49     ObjectOut = self.Stack[-1]
50     del self.Stack[-1]
51     return ObjectOut, self.Stack
52
53 # Function for calling the compareNodes function inside the Node class
54 def testCompareNode(ListOfNodeObjects):
55   compare = ListOfNodeObjects[0].compareNodes(ListOfNodeObjects[3])
56   print('Compare ' + ListOfNodeObjects[0].state + ' with ' + ListOfNodeObjects[3].state + '.')
57   print('Are they the same? ')
58   print(str(compare))
59
60 # Main function - This function will build nodes, create stack, queue and test all of them.
61 def main(NodeValues):
62
63   # These will be passed to the Queue and Stack classes respectively
64   ListOfNodeObjects = []
65   ListOfNodeObjects2 = []
66
67   # This is where the nodes are initially created
68   print('Order that everything is initially put into the Queue and also into the Stack')
69   for i in range(0, len(NodeValues)):
70     if (i==0):
71       node = Node(NodeValues[i], None, 2)
72     else:
73       node = Node(NodeValues[i], ListOfNodeObjects[i-1], 2)
74     print('State = ' + node.state + ', Order IN: ' + str(i+1))
75     ListOfNodeObjects.append(node)
76     ListOfNodeObjects2.append(node)
77
78   # This will test the node comparison function inside of the node class
79   print('\n')
80   print('Test the comparison Function')
81   testCompareNode(ListOfNodeObjects)
82
83   # Creates an instance of the Queue and Stack classes
84   Queue = FIFO(ListOfNodeObjects).Queue
```

```
85   Stack = LIFO(ListOfNodeObjects2).Stack
86   print('\n')
87   print('Converted into FIFO object (Queue) and one by one removed.')
88
89   # FIFO remove from and display order
90   for i in range(0, len(Queue)):
91     PoppedFromQueue, QueueObjects = FIFO(Queue).removeFromQueue()
92     print('State = ' + PoppedFromQueue.state + ', Order OUT: ' + str(i+1))
93     if (len(QueueObjects) == 0):
94       print('Queue is empty.')
95
96   # LIFO remove from and display order
97   print('\n')
98   print('Converted into LIFO object (Stack) and one by one removed.')
99   for i in range(0, len(Stack)):
100    PoppedFromStack, StackObjects = LIFO(Stack).removeFromStack()
101    print('State = ' + PoppedFromStack.state + ', Order OUT: ' + str(i+1))
102    if (len(StackObjects) == 0):
103      print('Stack is empty.')
104
105 # Any size of node array can be passed into main function
106 NodeValues = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
107 main(NodeValues)
```

```
Order that everything is initially put into the Queue and also into the Stack
State = A, Order IN: 1
State = B, Order IN: 2
State = C, Order IN: 3
State = D, Order IN: 4
State = E, Order IN: 5
State = F, Order IN: 6
State = G, Order IN: 7


Test the comparison Function
Compare A with D.
Are they the same?
False


Converted into FIFO object (Queue) and one by one removed.
State = A, Order OUT: 1
State = B, Order OUT: 2
State = C, Order OUT: 3
State = D, Order OUT: 4
State = E, Order OUT: 5
State = F, Order OUT: 6
State = G, Order OUT: 7
Queue is empty.


Converted into LIFO object (Stack) and one by one removed.
State = G, Order OUT: 1
State = F, Order OUT: 2
State = E, Order OUT: 3
State = D, Order OUT: 4
State = C, Order OUT: 5
State = B, Order OUT: 6
State = A, Order OUT: 7
Stack is empty.
```