18-662 Homework 2

February 14, 2025

Due date: Feb 26, 2025 at 5 pm EST

Problem 1: Maximum Flow for Image Segmentation [15 points]

In this problem, you will implement the Ford-Fulkerson algorithm to compute maximum flow. As part of this algorithm implementation, you will also have to implement an (uninformed) search algorithm, such as breadth-first search (BFS), depth-first search (DFS), or iterative-deepening search as a helper method.

Our goal in this problem is to perform image segmentation, which is the process of identifying and grouping pixels in an image based on which object they belong to. An example of this is shown in Figure 1, however, in your implementation, you will only be segmenting an image into two segments.

Figure 1. Image Segmentation Example

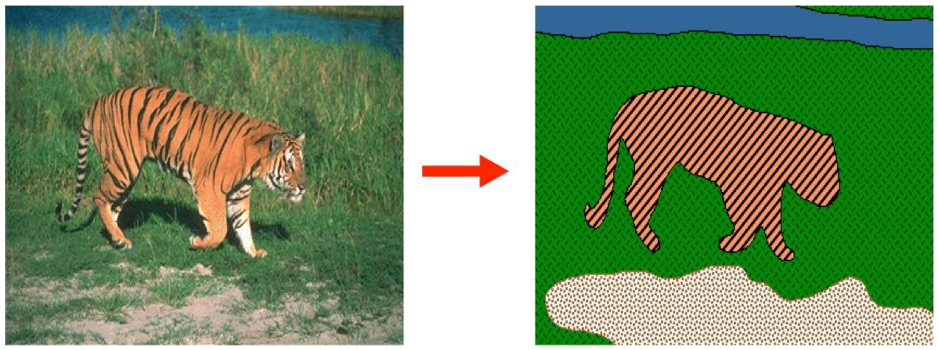


Image Credit: Stanford AI Lab
https://ai.stanford.edu/~syyeung/cvweb/tutorial3.html

Questions to answer before writing any code:

1. How does min-cut relate to this problem? Why is it true that the min-cut solution of a graph is equivalent to the max-flow solution? Draw examples if necessary.

2. Using your knowledge of min-cut, explain why solving the maximum flow problem is a valid method for image segmentation.

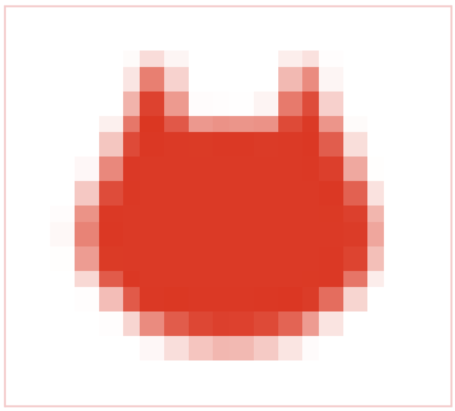
Requirements and Advice (read before implementing)

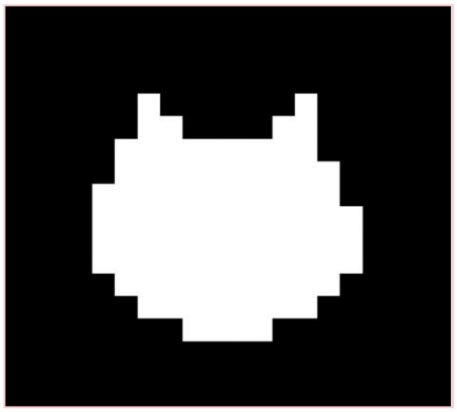
- Your code should include an implementation of the Ford-Fulkerson algorithm as well as an implementation of an uninformed search algorithm (as a helper to the Ford-Fulkerson algorithm).

- You can assume that you will only be segmenting an image into **two** segments (never more and never less). This is to mirror the source and sink in the maximum flow problem.
- Your code should have some method of converting an image to a graph since the Ford-Fulkerson algorithm will require a graph structure.
    - The nodes of your graph should correspond to the pixels of the image. There should be as many nodes as pixels.
    - For every pair of nodes, there should be an edge connecting them if and only if they correspond to neighboring pixels in the image. The weight of the edge should be some measure of pixel similarity, and a greater edge weight should be assigned when the pixels are more similar. You can use the absolute difference of pixel values as edge weight (or another similarity measure you choose that meets the requirements).
- Develop your code on simple graphs (containing only a handful of nodes) and **not** images. This will help you debug as you go.
- You may have to play around with how you determine which pixels are the source and sink. The simplest method is setting your source as some pixel in a corner and the sink as some pixel in the center of the image (i.e. where the object is most likely to be).
- In addition to the implementation, you should submit a segmentation of cat.png. Figure 2. is an example of the desired segmentation.
- You should use the following code to load, view, and store the image as a Numpy array (it will be much simpler if you input binary images to your code):

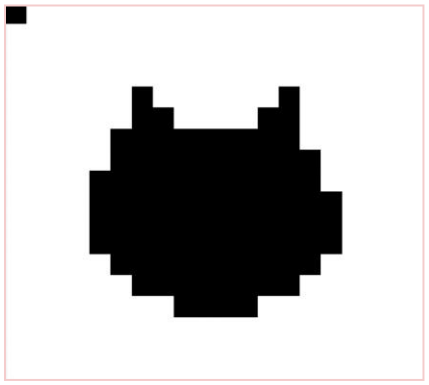
```
from PIL import Image, ImageOps
img_path = cat.png # get this photo from Canvas. file name (or path if
image isn't in the same directory)
img = Image.open(img_path) # define img using PIL
img = ImageOps.grayscale(img) # make the image grayscale
img.show() # view image

img = np.array(img).astype(np.float32) # convert img to ndarray
img = (img < 0.5).astype(int) # binarize the image
print(img.shape)
h, w = img.shape # get height and width of the img
```

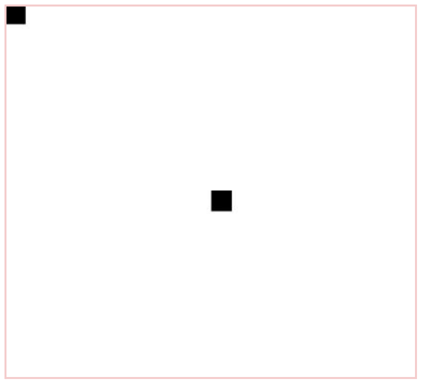
Figure 2. Desired Results


Input (low-resolution) cat.png


Binary input to algorithm

Resulting segmentation:


Nodes reachable from source


Nodes reachable from sink

Note: the resultant segmentations in your code may vary. It is important however that at least one map (either the nodes reachable from the source or the nodes reachable from the sink result in a valid segmentation as above). As an extension, you are welcome to repeat this problem but for non-binary inputs to the algorithm instead.

Questions to answer after implementation is complete:

1. Analyze the time complexity of the code you implemented. What could you change to reduce the runtime? (Note: no need to implement these changes—a few comments will suffice.)

2. What changes could you make to reduce the space (memory) requirement of your code? (Note: no need to implement these changes—a few comments will suffice.)

Problem 2: Constraint Satisfaction for Map Coloring Problem [10 points]

Write a Python script to generate random instances of map-coloring problems as follows: scatter n points on the unit square; select a point *X* at random, connect *X* by a straight line to the nearest point *Y* such that *X* is not already connected to *Y* and the line crosses no other line; repeat the previous step until no more connections are possible. The points represent regions on the map and the lines connect neighbors.

Now try to find *k*-colorings of each map, for both *k* = 3 and *k* = 4, using min-conflicts, backtracking, backtracking with forward checking, and backtracking with MAC (Maintaining arc-consistency). Construct a table of average run times for each algorithm for values of n up to the largest you can manage. Comment on your results.

Note: For Problems 3-6 refer to Jupyter notebook file Homework2_3to6.ipynb

Problem 3: MiniMax and Alpha-Beta pruning for Optimal Decisions [10 points]

You may use this visualization to understand how alpha beta pruning works:
https://pascscha.ch/info2/abTreePractice/

***This visualization is not related to the problem stated in the homework. It is just provided to get an understanding**

Finding the optimal next move in modified Tic-Tac-Toe

Minimax is a Kind of Backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. Alpha-Beta pruning is an optimization technique that can work on top of the Minimax algorithm to decrease computation complexity.

Your task in this section is to build an AI system that plays a perfect game. The agent will be playing a modified version of Tic-Tac-Toe where the size of the grid could be greater than or equal to 3X3. You must use both Minimax and Minimax with Alpha-Beta pruning to solve this problem. The input to the system will be the current state of the game in terms of a grid (list of list of str). The program must be able to produce the following outputs:

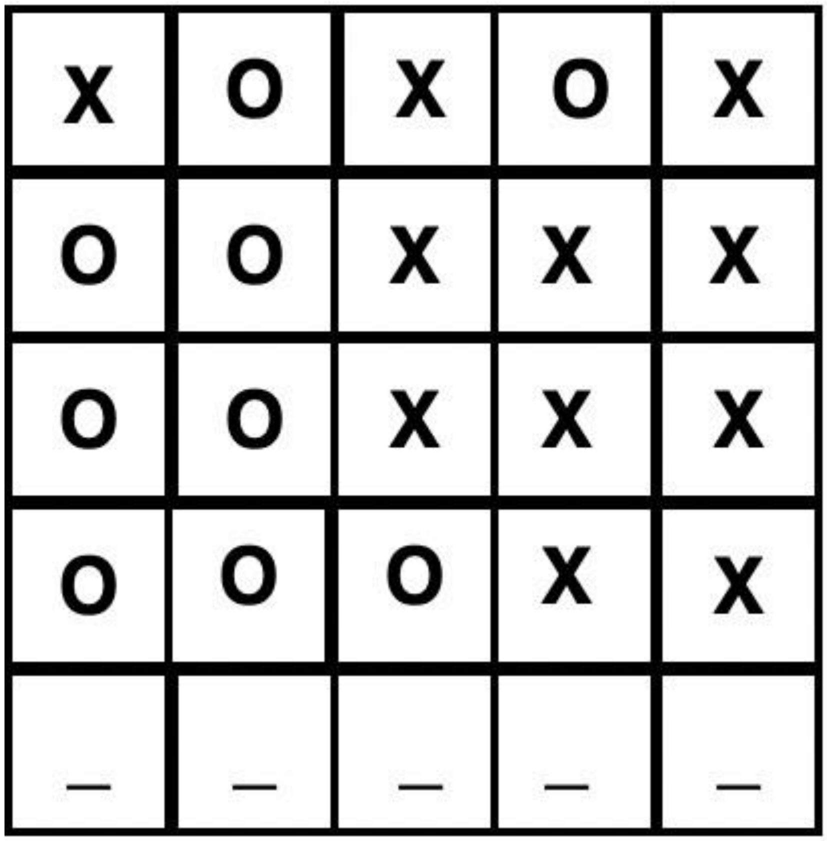
1. The Reward/value of the best move.
2. The position of the best move (the row and the column)
3. The time taken to get the optimal move in both cases (Minimax and Minimax with Alpha-Beta pruning)

Assumptions:

1. The grid will always be a square grid (the number of rows and columns will be equal)
2. 'x' will be the player and 'o' will be the opponent.
3. The player will win if he/she is able to draw a row/column/diagnol from one end of the grid to another(i.e., the maximum length it can be extended).
4. If there are 2 optimal positions, pick one with a lower column id
5. Reward must be +10 in case the player wins, -10 if the opponent wins and 0 if it is a tie. The priority must always be for the player to win.
6. The rows and columns are numbered from 0 to n.

Example:

Input:



```
board = [
        ['x', 'o', 'x', 'o', 'x'],
        ['o', 'o', 'x', 'x', 'x'],
        ['o', 'o', 'x', 'x', 'x'],
        ['o', 'o', 'o', 'x', 'x'],
        ['_', '_', '_', '_', '_']
]
```

Output:

The value of the best Move is : 10

```
The Optimal Move using minimax is :

ROW: 4  COL: 4

time taken by minimax: 0.0014388561248779297

The value of the best Move is : 10


The Optimal Move using alpha beta pruning is :

ROW: 4  COL: 4

time taken by minimax with alpha beta pruning: 0.0009059906005859375
```

The above is just one test case, more test cases will be considered for grading this question.

Problem 4: Dijkstra's and Bellman-Ford algorithms as applications of search [10 points]

Consider a kingdom with n communities. Each community is numbered from 0 to n-1, and they were connected by bonds of friendship and alliances.

The ruler of the kingdom is facing a difficult problem. He wanted to know which communities had the smallest number of friends that were reachable through some bond and he also wanted to keep in mind the distance a friend would have to travel in case of emergency. So, he created a map of these interconnecting communities and calculated a metric called 'sos' that was a combination of the bond that the 2 communities share and their distance.

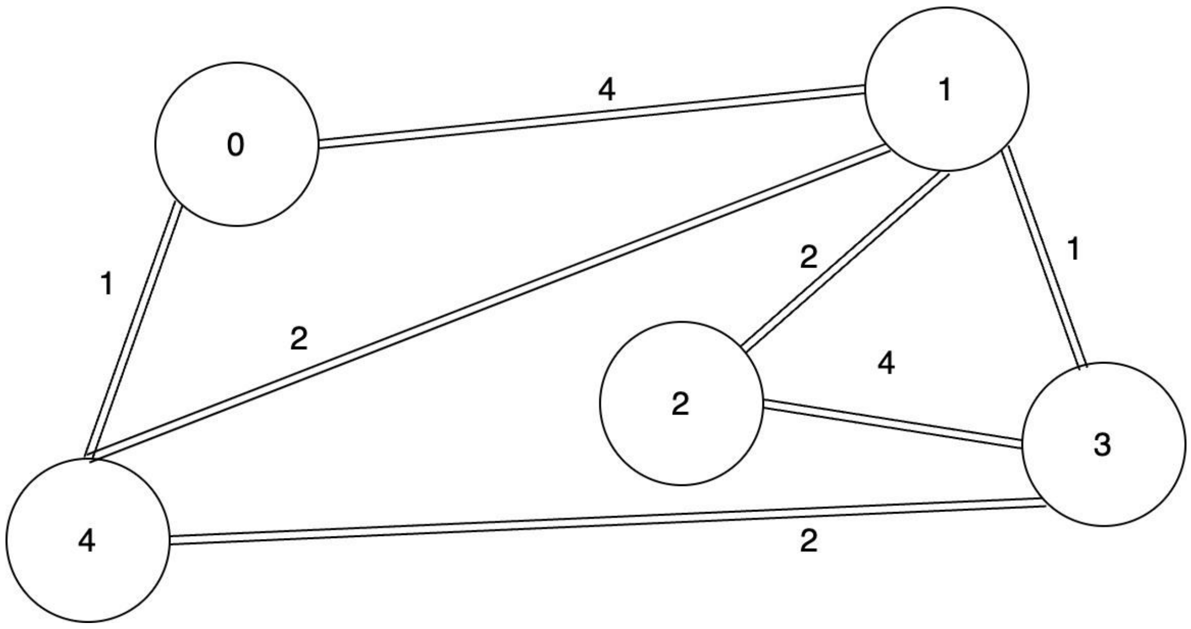
The ruler gathered all the information about the bonds between the communities and their distance, and used this information to find the answer to his problem. He represented this by weighted edges, and the metric 'sos' is determined by the weight of the edges.

He discovered that the strength of a bond connecting two communities x and y was equal to the sum of the edges' weights along that bond.

The ruler is concerned about the communities that might fall short of help in Emergency situations. Your task is to help the ruler find the community with the smallest number of friends that can help. To ensure that the friend is not too far, only the friends with a sos score 4 or below must be considered.

Your program should return a list of communities that might need help.

Input:



In the above graph, the nodes are the communities and the edges(bidirectional) show the sos score

The input to the program would be following:

n = 5

edges = [[0, 1, 4], [1, 2, 2], [1, 3, 1], [

    2, 3, 4], [3, 4, 2], [1, 4, 2], [0, 4, 1]]

max_sos = 4

where the convention for edges is: [src_node, dest_node, sos_score]. So the first element in the edges list above would mean that the sos score between community 0 and community 1 is 4.

Output:

[0,2]

Explanation: The communities and their corresponding friends that can help will be:

community0 = [1,3,4]   ==> 3 friends

community1 = [0,2,3,4] ==> 4 friends

community2 = [1,3,4]   ==> 3 friends

community3 = [0,1,2,4] ==> 4 friends

community4 = [0,1,2,3] ==> 4 friends

Thus community 0 and community 2 will need help as they have the minimum number of friends.

The above is just one test case, more test cases will be considered for grading this question.

Problem 5: Informed Search (A*, Heuristics)

[10 points]

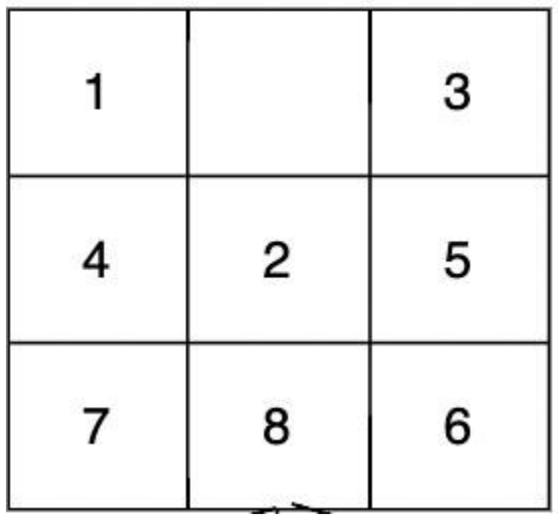
The N-puzzle, also known as the sliding puzzle, is a classic puzzle game where the player is tasked with sliding tiles on a grid to get them back into a specific order. The grid is typically an NxN square with numbers or images on each tile, and one empty tile that allows the other tiles to be moved. The goal is to rearrange the tiles back into their original order, which is usually numbered in ascending order or with a specific image arrangement. N-puzzles can range in size from a 3x3 grid to much larger grids (The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns), and can be played with either numbers or pictures. The puzzle is considered to be challenging because the player must carefully think about each move to ensure that they can ultimately reach the goal.

Part 1

In this task we will be considering the puzzle with numbers and you will solve it using A* and a heuristic of your choice (EXCEPT number of misplaced tiles since that was used in the live code example in recitation). The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration. Your program should be able to solve this puzzle and print all intermediate states leading up to the final state.

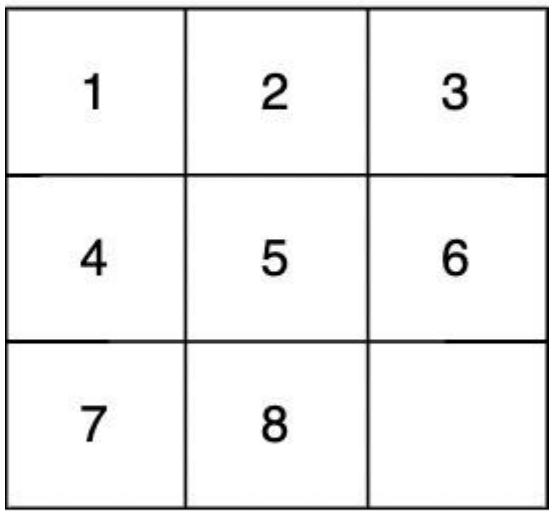
Here is and example of this puzzle when N =8:

Start state:



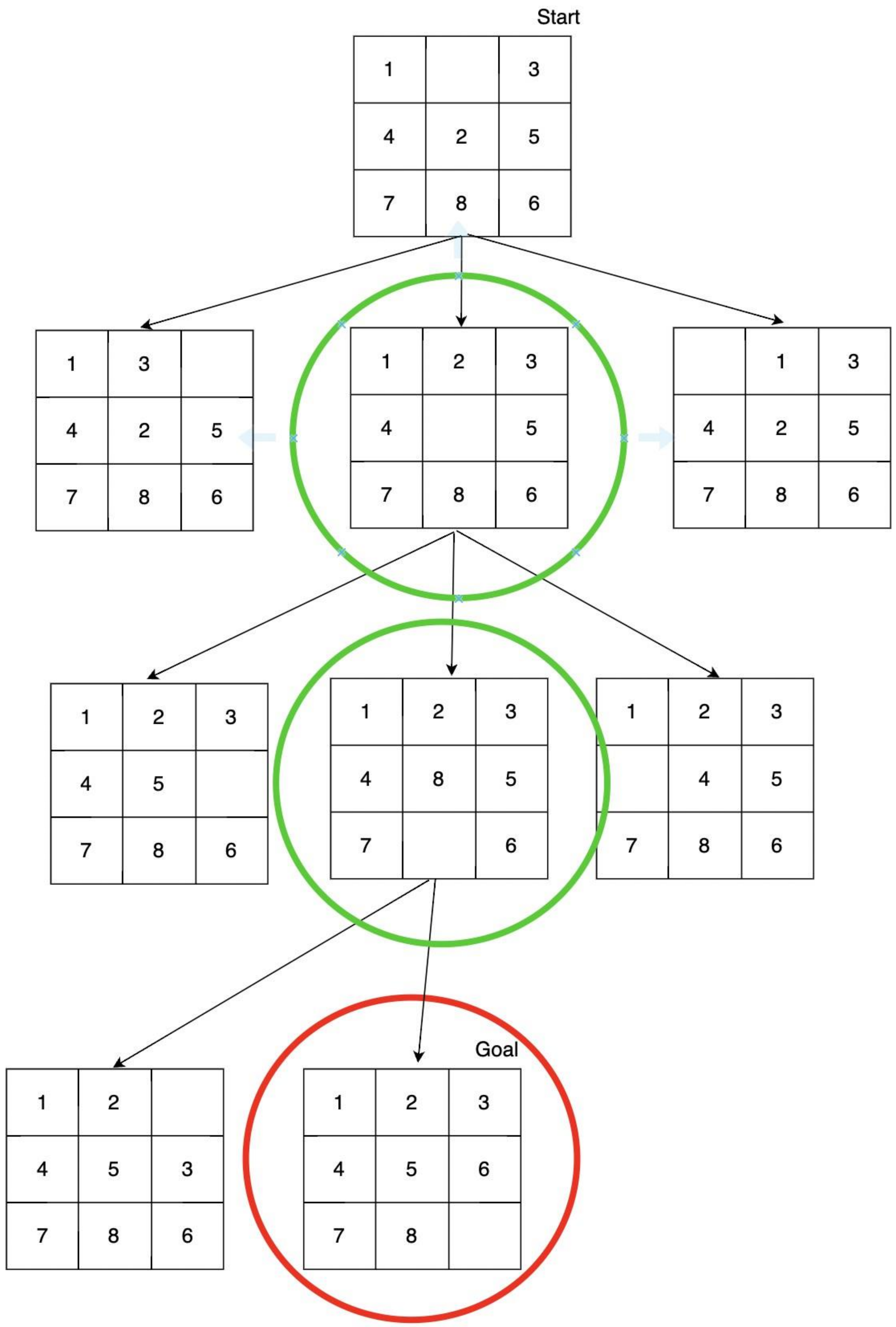
1		3
4	2	5
7	8	6

Goal state:



1	2	3
4	5	6
7	8	

Here is the transformations it would ideally go through in order to reach the goal state:


Start
1 3 / 4 2 5 / 7 8 6
1 3 / 4 2 5 / 7 8 6
1 2 3 / 4 5 / 7 8 6
1 3 / 4 2 5 / 7 8 6
1 2 3 / 4 5 / 7 8 6
1 2 3 / 4 8 5 / 7 6
1 2 3 / 4 5 / 7 8 6
1 2 / 4 5 3 / 7 8 6
Goal
1 2 3 / 4 5 6 / 7 8

The parts annotated with green are the selected intermediate states and the one in red is the goal. Your program should generate the following output:

Input:

```
Enter the start state matrix

1 _ 3
4 2 5
7 8 6
Enter the goal state matrix

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

Output:

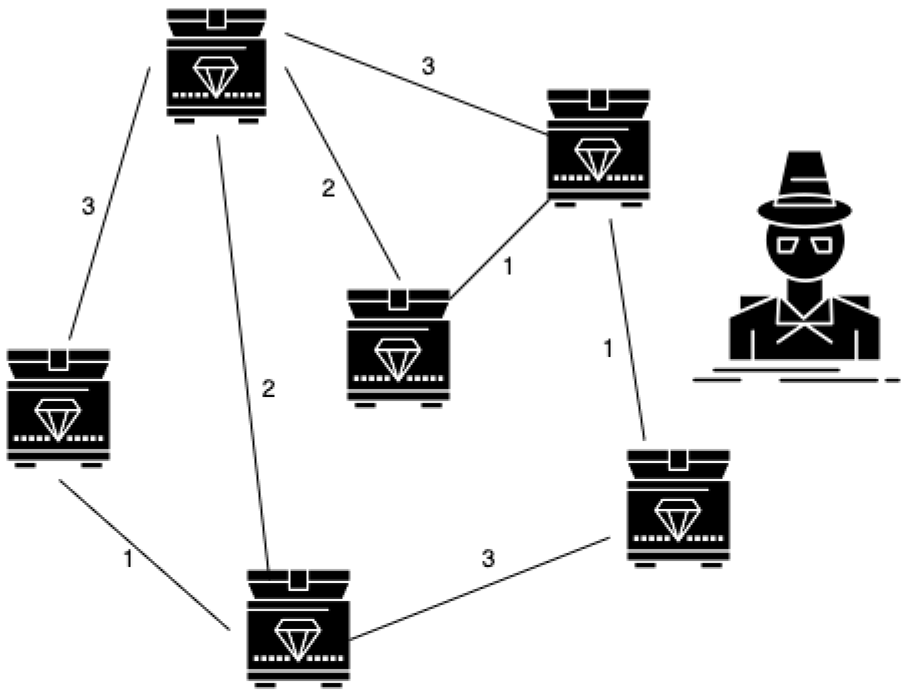
```
  |
  |
\'/

1 _ 3
4 2 5
7 8 6

  |
  |
\'/

1 2 3
4 _ 5
7 8 6

  |
  |
\'/

1 2 3
4 5 _
7 8 6

  |
  |
\'/

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

A starter code for input and output has been provided in jupyter notebook. You may choose to use or not use the code but the input and output format must remain consistent with the one shown here.

The example shows the puzzle for N=8 size=3X3, your code must implement a program such that it is able to solve the puzzle for any value of N (15,24...)

Problem 6: Minimum Spanning Tree, Kruskal's Algorithm, Prim's Algorithm [10 points]

Suppose you are an archaeologist who is trying to plan a trip to visit a set of ancient ruins located in different parts of a country. You want to visit each of the ruins at least once, but you also want to minimize the total distance that you need to travel in order to see them all. Each ruin has its own unique features and historical significance, and you are excited to learn as much as you can about each one. You can visualize this scenario in the form of a graph like this:



The above graph is just for visualization, it is not a test case to consider.

The nodes are the ruins and the weight on the edges(bi-directional) correspond to the distance (in thousands of km) you need to travel in order to move from 1 ruin to the other. To solve this problem, you must find the optimal route that will allow you to visit each ruin while minimizing the total distance traveled.

Your program should take as input a list of lists defining the adjacency matrix of a graph. The edge weights of 2 connecting nodes must be stored at indices row 0 and col 1 i.e., if the distance between ruin 0 and ruin 1 is 3000 km, graph[0][1] = 3

example input:

```
graph = [[0, 9, 0, 1, 0],
        [2, 0, 3, 7, 5],
        [0, 3, 0, 0, 7],
        [4, 8, 1, 0, 9],
        [0, 3, 6, 9, 0]]
```

The program must be able to output the edges and the weights selected by the algorithm in the form (the order of the node in edges while printing does not matter i.e., 1 - 2 is equal to 2 - 1:

```
Edge   Weight
2 - 1    3
3 - 2    0
0 - 3    4
1 - 4    3
```

A zero edge weight typically means there's no direct path between those two nodes. If you see a 0 in the adjacency matrix (other than on the diagonal), it implies there's no direct connection between those two ruins, and you cannot travel directly between them.

Another test case:

```
graph = [[0, 2, 0, 1, 0],
         [2, 0, 3, 8, 5],
         [0, 3, 0, 0, 1],
         [1, 8, 0, 0, 9],
         [0, 5, 1, 9, 0]]
```

Note: For Problems 3-6 refer to Jupyter notebook file Homework2_3to6.ipynb

Problem 7: Two-Player Game [15 points]

In this problem we will extend the previous problem. Your task is to design an *adversarial* agent that will compete against you in N-puzzle by trying to place the tiles in the opposite order.

Desired Outcomes of the Game

8	**7**	**6**
5	**4**	**3**
2	**1**	**0**

The opponent's desired end state

1	**2**	**3**
4	**5**	**6**
7	**8**	**0**

The agent's desired end state

Game description:

- This is a zero-sum game (i.e. the players take turns and the net gain and loss from each round is zero). You will define you own zero-sum reward system (i.e. when and how many points are assigned at each step).
- The problem will be simplest if you assume that the agent is trying to minimize a heuristic and that the opponent is trying to maxize the same heuristic or vice versa. You can choose any (appropriate) heuristic. This heuristic should be used to determine the value of each final position (i.e. the leaves of the game tree).
- The opponent is adversarial (not stochastic) and the opponent should view the agent as an adversarial (not stochastic) opponent.
- Define an `isEnd()` function that you use to determine whether the game is over.

Things to keep in mind:

- You can use either minimax or minimax with alpha-beta pruning to search the game tree.
- If you make any other assumptions or design choices, include an explanation along with your code submission.

Problem 8: A Bargaining Game Using Expectiminimax [10 points]

In this problem, you will implement an AI negotiator that engages in a simple buyer-seller price negotiation. The buyer aims to get the lowest possible price, while the seller tries to maximize the deal value. However, the seller may sometimes act randomly, meaning the negotiation is not purely deterministic. The buyer must decide whether to accept the seller's price, counter with a slightly higher offer, or walk away from the deal. The seller either accepts, counters with a lower price, or makes an unpredictable counteroffer
Your task is to use the Expectiminimax algorithm to model the buyer's and seller's decision-making while accounting for possible uncertainty. The game lasts for a maximum of 5 rounds, and if no deal is reached, the negotiation fails. See skeleton code: hw2_Bargaining.py
Game Rules

1. The buyer and seller negotiate using a series of offers and counteroffers.
2. The buyer always starts with a random offer in a predefined range.
3. The seller starts with an initial asking price and may adjust it based on the buyer's move.
4. The seller behaves stochastically, meaning there is a fixed probability (α) that they will act randomly instead of optimally.
   - With probability $(1 - \alpha)$, the seller makes the best possible counteroffer based on Expectiminimax.
   - With probability α, the seller chooses a random counteroffer (either increasing or decreasing their price unpredictably).
   - α is a fixed parameter (e.g., $\alpha = 0.3$ means the seller behaves randomly 30% of the time).
5. If the buyer's offer meets or exceeds the seller's price, the deal is immediately accepted.
6. If 5 rounds pass without an agreement, the deal fails.
7. The buyer may choose to walk away, ending the negotiation early.

Scenario 1 (Quick Deal)
Round 1: Buyer offers \$50, Seller asks for \$80.
Buyer counters with \$55.
Seller counters with \$75.
Buyer accepts the \$75 deal.
Negotiation successful!

Scenario 2 (Failed Deal)
Round 1: Buyer offers \$40, Seller asks for \$90.
Buyer counters with \$45.
Seller randomly asks for \$95 (unexpected move).
Buyer walks away.
Negotiation failed!

Tasks:
expectiminimax(player, buyer_offer, seller_price, depth) is the core AI function for decision-making. You need to implement:

1. How the buyer and seller make decisions.
2. How the algorithm handles randomness in the seller's behavior -The seller follows Expectiminimax $(1 - \alpha)$% of the time but acts randomly α% of the time.
3. The logic for exploring possible future moves.

Problem 9 – Connect Four [10 points]

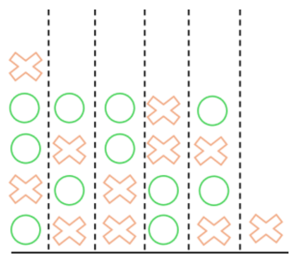


Figure 1. A typical 6 x 6 Connect Four Game Layout

In this problem, we will examine the classic game of Connect Four. A standard Connect Four board consists of 6 rows and 7 columns. Players take turns dropping pieces into one of the 7 columns. Due to gravity, the pieces stack from the bottom up within each column, rather than being placed freely on the board like in tic-tac-toe or Go. This means that players must carefully consider not only where they drop their piece but also how it might influence future moves. As the game progresses, the board fills up from the bottom, creating strategic opportunities to block an opponent or set up a winning sequence. If you haven't played it before, you can get familiar with the game through the following link: https://www.cbc.ca/kids/games/all/connect-4.

To limit the state space slightly, let's limit the board size to by 6 x 6 for this problem.

(1) Now, to get you started. In a few sentences, justify whether the game is a minimax game?

For the following two questions complete the *connectfourminimaxquestions.py.*

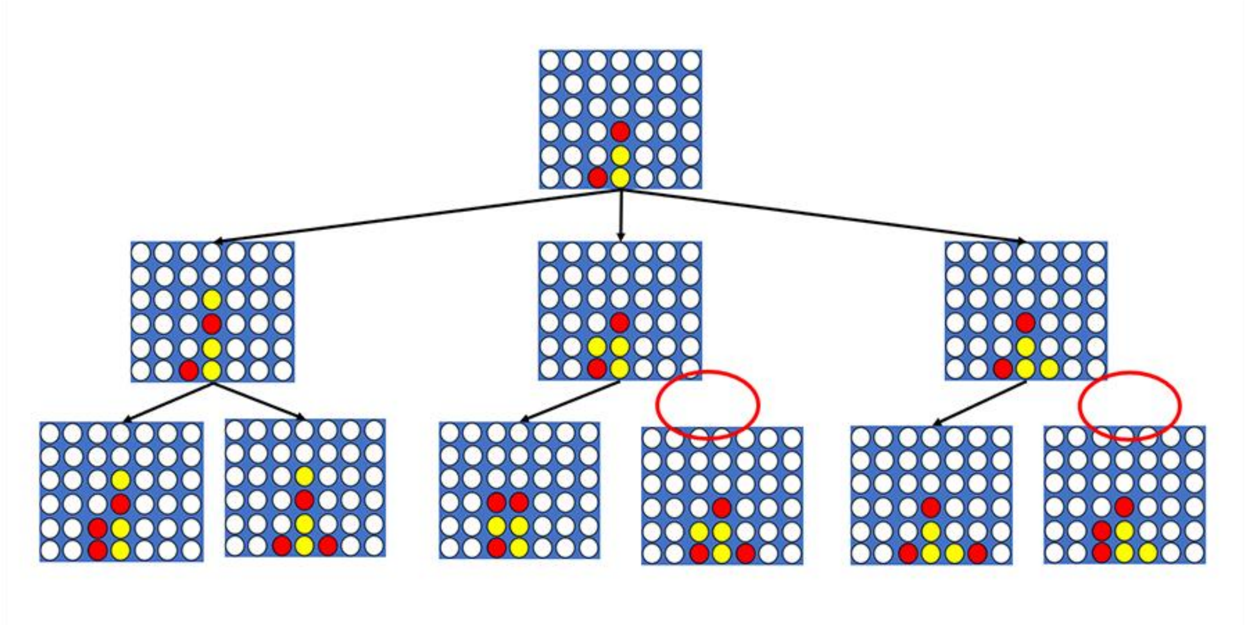
(2) Write a minimax solver (without any pruning) for connect four? I have written some helper functions in the connectfourminimax.py helper code. For this question, what is the very next piece 'x' player wants to play? Also, give us the time you take to compute the next move.

(3) Now apply alpha-beta pruning to your solver. Give us the time you take to compute the next move.

Although for (2) and (3), we ask specifically for the next move of the specific test case. We have some unreleased testcases, which we will run to check your solver for grading.

(4) The game isn't an expectimax problem. In a few sentences, how can we convert the problem to an expectimax problem. (Any rational modification will suffice).

Extra credit [10 points] (optional)



In this assignment, you will implement an AI agent for a Connect Four game using the Minimax algorithm. The goal is to create an unbeatable AI that either wins or ties every game. You will start with a partially completed Connect Four game implementation and add the AI component. The Minimax algorithm should be used to evaluate potential moves and ensure optimal play. You will need to implement the AI's decision-making process, including the evaluation function and the Minimax algorithm itself. The assignment will help you understand game theory, recursion, and algorithmic thinking.

Connect Four Game Overview

Connect Four is a two-player connection game played on a vertically suspended grid with **6 rows** and **7 columns**. Players take turns dropping discs into a column, and the disc falls to the lowest unoccupied row in that column. The objective is to **connect four of your discs in a row**—horizontally, vertically, or diagonally—before your opponent does. If the grid fills completely with no player achieving four in a row, the game ends in a tie.

Key Rules:

1. Players alternate turns, starting with Player 1 (human) in this implementation.
2. A valid move requires selecting a column that is not already full.
3. The game terminates immediately when a player connects four discs or the grid is full.

This game is a classic example of a **zero-sum, perfect-information adversarial game**, making it an excellent case study for algorithms like Minimax. The turn-based nature and finite branching factor align well with adversarial search strategies discussed in lectures.

Below are some resources to help understand more how the game is played:

- How to Play Connect Four
- https://www.unco.edu/hewit/pdf/giant-map/connect-4-instructions.pdf
- https://www.wikihow.com/Play-Connect-4

Task to be Done

Your task is to complete the implementation of the AI player in the Connect Four game. You will use the Minimax algorithm to enable the AI to make optimal moves. The Minimax algorithm should consider all possible moves and choose the one that maximizes the AI's chances of winning while minimizing the opponent's chances. You will need to implement the following components:

Evaluation Function: This function should evaluate the board state and return a score indicating how favorable the position is for the AI.
Minimax Algorithm: Implement the Minimax algorithm to simulate future game states and choose the best move for the AI.
Alpha-Beta Pruning: Optionally, you can implement alpha-beta pruning to optimize the Minimax algorithm by reducing the number of nodes that are evaluated.

Starting Code

code.py is the starting code for the Connect Four game. You will need to complete the sections marked with TODO comments to implement the AI player.

Instructions

Implement the Minimax Algorithm: Complete the minimax function to evaluate all possible moves and choose the best one for the AI.
Pick the Best Move: Implement the pick_best_move function to determine the optimal move for the AI based on the Minimax algorithm.
Test Your Implementation: Play multiple games against the AI to ensure it performs optimally and never loses.

By completing this assignment, you will gain a deeper understanding of game theory and games as a subset of search problems (including two-agent or multi-agent zero-sum adversarial games). Good luck!