Math 300: Midterm

Noah Jackson

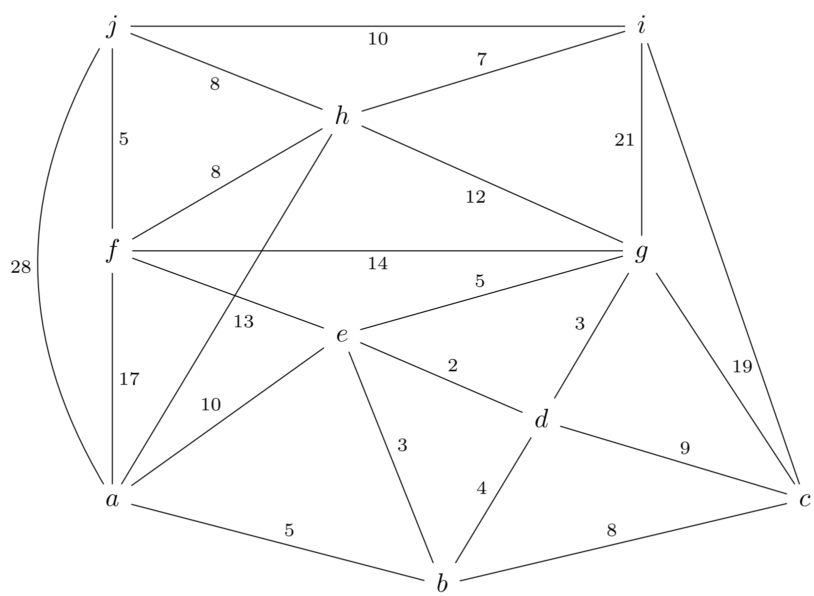
July 13, 2025

Please complete the following problems and typeset your solutions using LaTeX. If AI is used you must provide the model that was used, the time/date it was used, and the query that was entered into the model. Un-cited use of AI will result in a zero for the problem. If AI is used, you must provide a description of how the results work in your own words. You may work with at most 1 other student (and you are encouraged to do so!). If you choose to work with another student you must typeset your own results and use your own words. You must also put the name of your collaborator on your submission. You are welcome to ask me (Jared) questions as well, but I may not answer all questions.

The midterm is out of 105 points with 160 points possible

1. (40 points) Implement Kruskal's algorithm and run your code on a graph of your own design. All vertices must have degree 4 or more, and all edges must have different weights. You also must have at least 10 vertices. Your graph must be included in your writeup (perhaps using quiver?), as well as the subgraph that your algorithm produces. Provide a brief description of why you think this algorithm is able to achieve the task it is designed for.

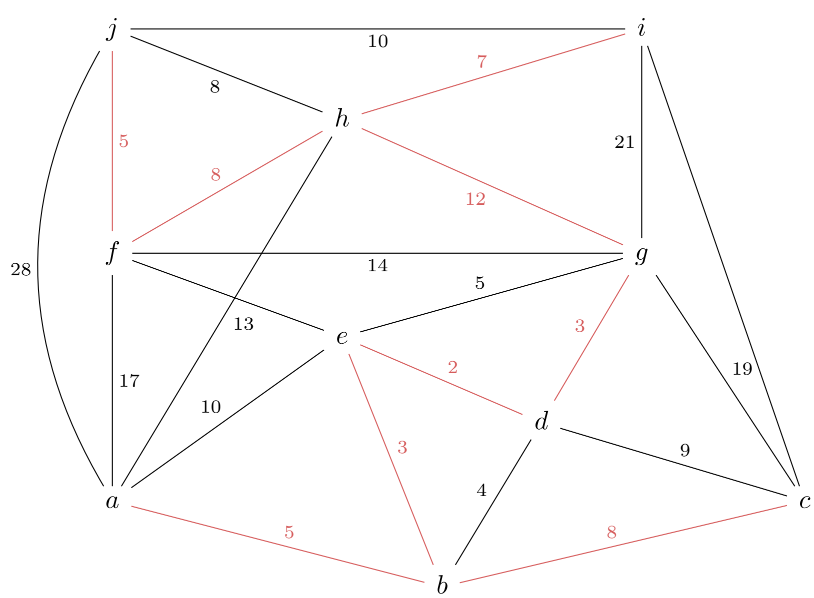
   Accidentally did Prim's (Kruskal's is below this). Here is the graph:




```
    adj1_matrix = [
    # a  b  c  d  e  f  g  h  i  j
    [ 0, 5, 0, 0,10,17, 0, 0, 0,28],  # a
    [ 5, 0, 8, 4, 3, 0, 0, 0, 0, 0],  # b
    [ 0, 8, 0, 9, 0, 0,19, 0, 0, 0],  # c
    [ 0, 4, 9, 0, 2, 0, 3, 0, 0, 0],  # d
    [10, 3, 0, 2, 0,13, 5, 0, 0, 0],  # e
    [17, 0, 0, 0,13, 0,14, 8, 0, 5],  # f
    [ 0, 0,19, 3, 5,14, 0,12,21, 0],  # g
    [ 0, 0, 0, 0, 0, 8,12, 0, 7, 8],  # h
    [ 0, 0, 0, 0, 0, 0,21, 7, 0,10],  # i
    [28, 0, 0, 0, 0, 5, 0, 8,10, 0],  # j
]
```

```


def find_min_tree_prim(adj_matrix, starting_letter):
  T = []
  visited = set()
  letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
  starting_index = letters.index(starting_letter)
  visited.add(starting_index)

  while len(visited) < len(adj_matrix):
    min_weight = float('inf') # highest number possible
    min_edge = None

    for i in visited: # visited will look like something like: (0, 1, 2)
      for j, weight in enumerate(adj_matrix[i]): # looks through all the connections of a node that has been visited
        if j not in visited and  0 < weight < min_weight: # makes sure this new edge doesn't connect to already visited node and sees
                                                          # if it actually has a lower weight than already recorded
          min_weight = weight
          min_edge = (i, j)


    adj_matrix[min_edge[0]][min_edge[1]] = 0 #makes it so that this edge will have no possibility of being looked at again
    adj_matrix[min_edge[1]][min_edge[0]] = 0 #matrix is symmetric

    if min_edge == None: # graph could be disconnected
      break

    if min_edge != None:
      T.append([letters[min_edge[0]], letters[min_edge[1]], min_weight])
      visited.add(min_edge[1])

  return T

print(find_min_tree_prim(adj1_matrix, 'a'))

```

This gave us the Tree:



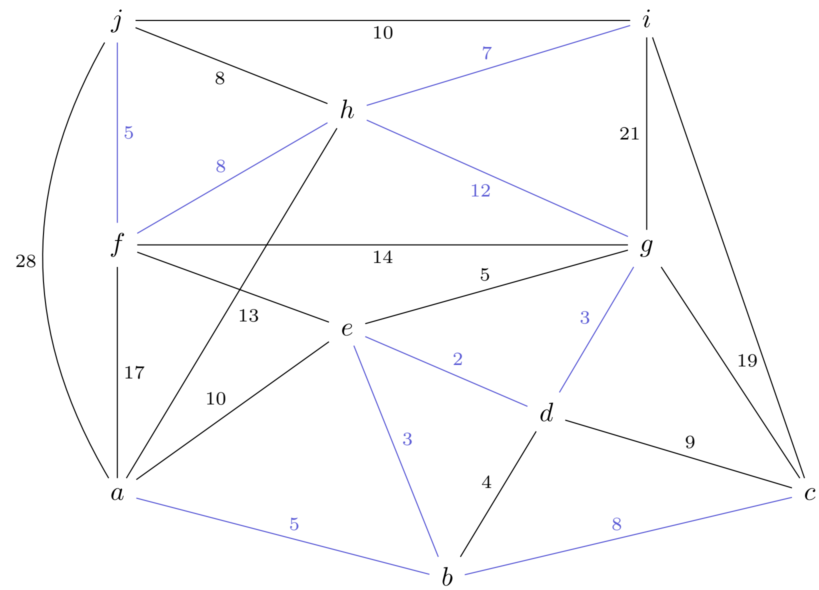
This algorithm is always picking our smallest edge out of the edges that already have a node that is connected. The reason this works for finding the minimal tree is that each iteration is finding the minimal way to add a new node visit without going to a node already visited. The way I like to think about it is imagine each edge is corresponding to the difficulty to getting to get from point i to point j. What our algorithm is doing is trying to find the easiest difficulty to get to a new point given the points we have already gone to.

Now for Kruskal's:

```python
    adj1_matrix = [
    # a  b  c  d  e  f  g  h  i  j
    [ 0, 5, 0, 0,10,17, 0, 0, 0,28],  # a
    [ 5, 0, 8, 4, 3, 0, 0, 0, 0, 0],  # b
    [ 0, 8, 0, 9, 0, 0,19, 0, 0, 0],  # c
    [ 0, 4, 9, 0, 2, 0, 3, 0, 0, 0],  # d
    [10, 3, 0, 2, 0,13, 5, 0, 0, 0],  # e
    [17, 0, 0, 0,13, 0,14, 8, 0, 5],  # f
    [ 0, 0,19, 3, 5,14, 0,12,21, 0],  # g
    [ 0, 0, 0, 0, 0, 8,12, 0, 7, 8],  # h
    [ 0, 0, 0, 0, 0, 0,21, 7, 0,10],  # i
    [28, 0, 0, 0, 0, 5, 0, 8,10, 0],  # j
]

def find_root(parent_array, i):
  if parent_array[i] == i:
    return i
  parent_array[i] = find_root(parent_array, parent_array[i])
  return parent_array[i]

def union(parent_array, rank_array, x, y):
  root_x = find_root(parent_array, x)
  root_y = find_root(parent_array, y)
  if root_x != root_y:
        if rank_array[root_x] > rank_array[root_y]:
            parent_array[root_y] = root_x
        else:
            parent_array[root_y] = root_x
            if rank_array[root_x] == rank_array[root_y]:
                rank_array[root_x] += 1

def sort_by_weight(adj_matrix):
  weight_sorted_edges = []
  for i in range(len(adj_matrix)):
    for j in range(len(adj_matrix[i])):
      if adj_matrix[i][j] != 0:
        weight_sorted_edges.append((i, j, adj_matrix[i][j]))
  weight_sorted_edges.sort(key=lambda x: x[2])
  return weight_sorted_edges

def Kruskals(adj_matrix):
  E = sort_by_weight(adj_matrix)
  T = []
  letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

  parent = list(range(len(adj_matrix)))
  rank = [0] * len(adj_matrix)

  for u, v, weight in E:
    if find_root(parent, u) != find_root(parent, v):
      T.append((u, v, weight))
      union(parent, rank, u, v)

  T = [(letters[u], letters[v], weight) for u, v, weight in T]
  return T

print(Kruskals(adj1_matrix))


```

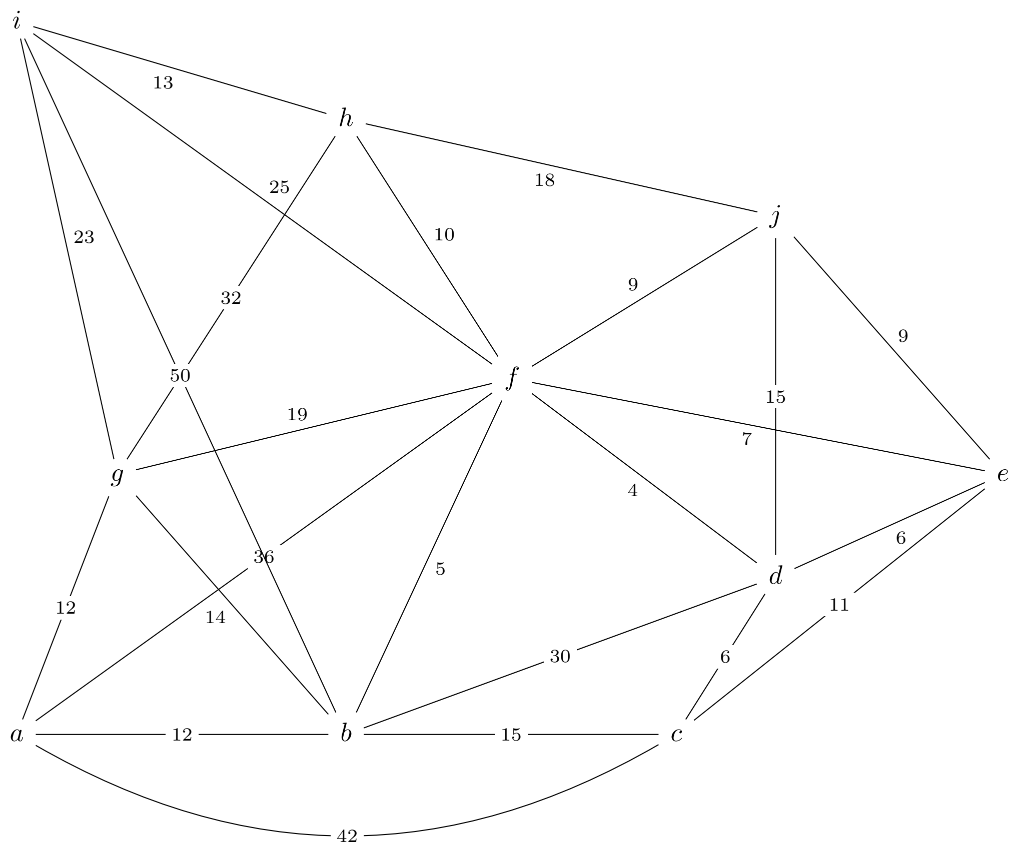
This algorithm goes off the idea that we are making multiple rooted trees in the process of doing

Kruskal's. We update the parent of a node once we have connected them by an edge. If the parents are the same, that means the nodes are already connected by a rooted tree. The rank filters how "parent-y" the node is, so that each tree can have a decisive node to become the "Mother node". It also creates the same tree as Prim's:



2. (40 points) Implement Dijkstra's algorithm, and run your code on a graph of your own design to find the shortest path between two vertices where the shortest path contains more than 1 edge. All vertices must have degree 4 or more, and all edges must have different weights. You also must have at least 10 vertices. Your graph must be included in your writeup (perhaps using quiver?), as well as the subgraph that your algorithm produces. Provide a brief description of why you think this algorithm is able to achieve the task it is designed for.

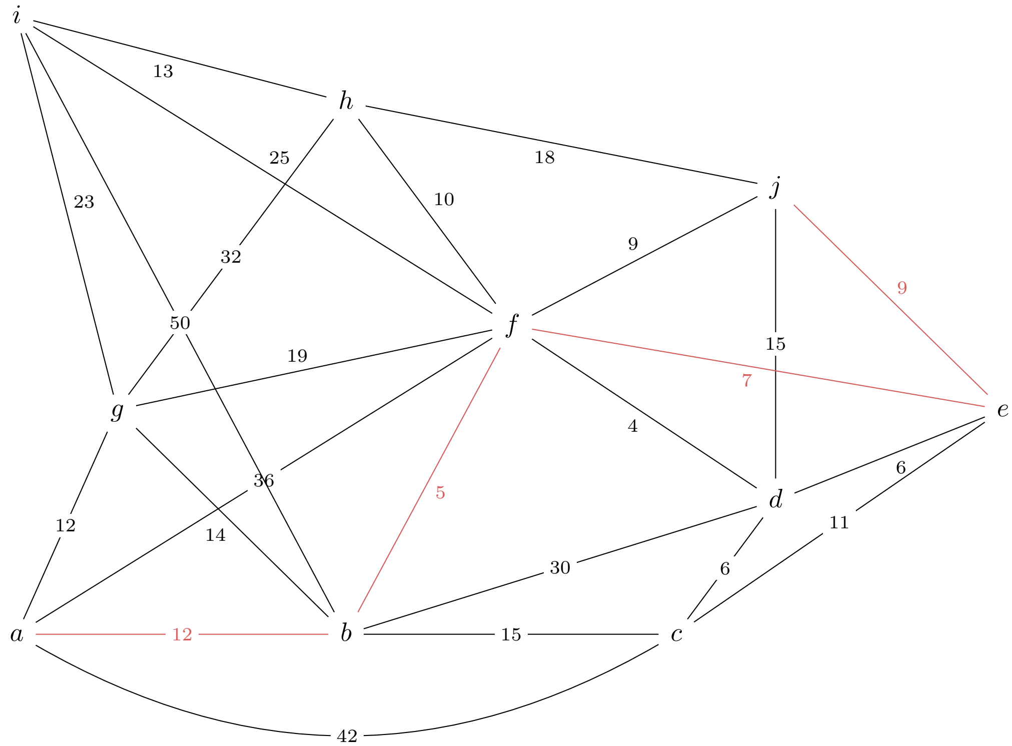
   Here is the graph I am going to be testing Dijkstra's on:




```
    adj2_matrix = [
    # a   b   c   d   e   f   g   h   i   j
    [ 0, 12,  0,  0,  0,  0, 12,  0,  0,  0],  # a
    [12,  0, 15, 30,  0,  5, 14,  0,  0, 42],  # b
    [ 0, 15,  0,  6, 11,  0,  0,  0,  0,  0],  # c
    [ 0, 30,  6,  0,  6,  4,  0,  0,  0, 15],  # d
    [ 0,  0, 11,  6,  0,  9,  0,  0,  0,  9],  # e
    [ 0,  5,  0,  4,  9,  0, 19, 10, 18,  0],  # f
    [12, 14,  0,  0,  0, 19,  0, 32, 23, 50],  # g
    [ 0,  0,  0,  0,  0, 10, 32,  0, 13,  0],  # h
    [ 0,  0,  0,  0,  0, 18, 23, 13,  0, 25],  # i
    [ 0, 42,  0, 15,  9,  0, 50,  0, 25,  0],  # j
]


def Dijkstra(adj_matrix, start, end):
  letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
  starting_index = letters.index(start)
  ending_index = letters.index(end)

  n = len(adj_matrix)
```

```python
    labels = [['', None] for _ in range(n)]
    visited = []                                    #     a          b
    labels[starting_index] = [start, 0]             # [['a', 0], ['a', 2], ...]

    while len(visited) < n:


      min_distance = float('inf')
      min_index = None
      # finds what node is best to visit
      for i in range(n):
        if i not in visited and labels[i][1] is not None and labels[i][1] < min_distance:
          min_distance = labels[i][1]
          node = i


      if node is None:
        break

      visited.append(node)
      # for the current visited node, look at its distance, compare to the labels distance and see if it is better than what's currently there
      for i, weight, in enumerate(adj_matrix[node]):
        if weight > 0:
          new_distance = labels[node][1] + weight
          if labels[i][1] is None or new_distance < labels[i][1]:
            labels[i] = [letters[node], new_distance]

    best_path = []
    current_node = ending_index
    best_path.append(letters[current_node])
    while current_node != starting_index:
      best_path.append(labels[current_node][0])
      current_node = letters.index(labels[current_node][0])
    best_path.reverse()


    return best_path, labels[ending_index][1]

print(Dijkstra(adj2_matrix, 'a', 'j'))
```

This created this path:



This algorithm will eventually find all of the optimal ways to get to each of the nodes. We update each node only if the total magnitude for the given path is a better deal then the total magnitude already found by another path. In Dijkstra's, once we have visited a node, our path is "finalized", meaning that we have found the optimal way to get to the node from the starting point in the given path. So, once we have visited all of the nodes, we know the most optimal path for each of them.

3. (80 points) Consider the game chomp from class on an $n \times m$ board drawn in the plain. The lower left square is the sad square. Write a python function that takes as input the current state of the board, and returns the index of the square which is a winning move for that board state. You may assume that the board state input is a position that would be handed to player 1 (after all, player 1 always wins with optimal moves.

   Your code will be tested against 4 kinds of boards of increasing difficulty, and the following points will be assigned based on your algorithms ability to win:

   (a) (10 points) the board at the begining of the game is a linear board $1 \times n$

   (b) (15 points) the board at the beginning of the game is a square board $n \times n$.

   (c) (20 points) the board at the beginning of the game is a $2 \times n$ board

   (d) (35 points) the board at the beginning of the game is of arbitrary dimensions $n \times m$

   The first 3 cases have simple winning strategies, while the last case is more difficult. I will not test your code against boards larger than 100×100. If your code takes more than a few seconds to provide a move, I will treat it as your code failing to produce an answer.

```
        import numpy as np
import math
import matplotlib.pyplot as plt

def play_chomp():

  m, n = input(f'The board will be m columns and n rows, please enter the values for m and n: ').split()
  m = int(m) - 1
  n = int(n) - 1

  board = initialize_board(m, n)
  print('Board is now initialized: ')
  print_board(board)

  i = 0

  while any(0 in row for row in board):

    i += 1

    if i % 2 == 1:
      print("Player 1's turn: ")

      if n == 0: #trivial cases

        move_row = 0
        move_column = 1
        board = turn(board, move_row, move_column)
        print_board(board)

      if m == 0:
        move_row = n - 1
        move_column = 0
        board = turn(board, move_row, move_column)
        print_board(board)

      if m == n: # square board

        if i == 1: #first move

          move_row = n - 1
          move_column = 1


        else:

```

```
          move_row = n
          move_column = n - p_row

        board = turn(board, move_row, move_column)
        print_board(board)

      if n == 1: #2xn

        if i == 1:

          move_row = 0
          move_column = m
          board = turn(board, move_row, move_column)
          print_board(board)
        else:
          if p_row == 0:
            move_row = 1
            move_column = p_column + 1
            board = turn(board, move_row, move_column)
            print_board(board)

          if p_row == 1:
            move_row = 0
            move_column = p_column -1
            board = turn(board, move_row, move_column)
            print_board(board)

      if m == 1: #nx2

        if i == 1:

          move_row = 0
          move_column = 1
          board = turn(board, move_row, move_column)
          print_board(board)
        else:
          if p_column == 0:
            move_column = 1
            move_row = p_row + 1
            board = turn(board, move_row, move_column)
            print_board(board)
          if p_column == 1:
            move_column = 0
            move_row = p_row - 1
            board = turn(board, move_row, move_column)
            print_board(board)

      #else: #mxn

    else:
      invalid_input = True
      while invalid_input == True:
        print("Player 2's turn: ")
        p_row, p_column = input('Choose your row and column: ').split()
        p_row = int(p_row) - 1
        p_column = int(p_column) - 1
        if board[p_row][p_column] == 'x' or ((0 > p_row < n) or (0 > p_column < m)):
          if p_row == n and p_column == 0:
            print('You ate the poison and lost!')
            return
          else:
            print('Invalid move, please try again')
        if board[p_row][p_column] != 'x':
          invalid_input = False
          move_row = p_row
          move_column = p_column
          board = turn(board, move_row, move_column)
          print_board(board)
```

```python

  print('Player 1 wins!')

def initialize_board(m, n):
  board = [[ 0 for _ in range(m + 1)] for _ in range(n + 1)]
  board[n][0] = 'x'
  return board

def print_board(board):
  for i in range(len(board)):
    for j in range(len(board[0])):
      print(board[i][j], end=' ')
    print()

def turn(board, move_row, move_column):
  n = len(board)
  m = len(board[0])

  for i in range(move_row + 1):
    for j in range(move_column, m):
      board[i][j] = 'x'
  return board

play_chomp()

```

This implementation works for boards in cases:

- $1 \times n$ or $m \times 1$: This is a trivial case because since we are the player to make the first move, we can just go to the right by 1 or up by 1 and this forces player 2 to eat the poison
- The $2 \times n$ or $m \times 2$ case uses the strategy of always having the board turn into a rectangle with 1 less piece on the upper right. If the strategy keeps going and going, we will eventually get to when player 2 has to eat the chocolate
- The $m = n$ case is unique in its strategy. The strategy goes that player 1 will condense the board down into an L-shaped board. After this, player 1 matches player 2's moves by whatever player 2 does, player 1 does the equivalent version of that move flipped across the 45-degree axis. This makes the board always symmetric at the start of player 2's turn, gauranteeing that an extra move after their turn is possible.

AI Model Used: GPT-4o, OpenAI ChatGPT Date Used: July 6-7, 2025

- "Make an adjacency matrix of this" (uploaded an image of a graph showing vertices and weighted edges)
- "Put it in Python"
- "Can you go through a couple runs to show me how this works"
- "Can you keep going through this process"
- (Uploaded code image) "Does it matter in the last argument: `if rank_array[root_x] == rank_array[root_y]: rank_array[root_x] += 1` vs. `rank_array[root_y] += 1`"
- "Show me what would happen if we reversed this then"
- `parent_array[root_x] = root_y`
  `if rank_array[root_x] == rank_array[root_y]:  rank_array[root_y] += 1`
- (Uploaded graph image) "Wouldn't there be a problem with the change in this graph though—`f{j` would result in `j` rank going up, then `f{h` would result in `h` rank going up, and `j` and `h` would connect, creating a cycle"
- `if rank_array[root_x] < rank_array[root_y]:  parent_array[root_x] = root_y`
  "Wouldn't this have to flip then too?"
- "In my homework it says: 'If AI is used you must provide the model that was used, the time/date it was used, and the query that was entered into the model. Un-cited use of AI will result in a zero for the problem.' So cite all that I did for this"
- "So go through the proof by induction step with the graph I showed you"
- "So when can we finalize a node then?"
- "This graph is a counterexample, correct? Because `a{b{e` is not as small as `a{f{e`"
- "`a{f = 6` and `f{e` is 1, is it now a counterexample?"
- "Why is this an infinite loop?"
- `def Dijkstra(adj_matrix, start, end):` (code debugging and algorithm questions)
- "How to enumerate again?"
- "How to initialize an array of same size of adjacency matrix but each entry is a list of size 2"
- "Make adjacency matrix in Python of this"
- "Does this work for Prim's?"
- "Is this the correct way to implement Kruskal's algorithm?"
- "Fix the adjacency matrix then"
- "Why is `aj = 28` but `ja = 10`?"
- "Convert this into adjacency matrix"
- "Make an adjacency matrix of this"
- "Is a vertex finalized when we pick it?"
- "This is a counterexample — look at `e`, we would pick `e` but it can't be finalized yet, right?"
- "So this is a counterexample to saying that we know it's finalized when we pick it"
- "Is there a general strategy for any board size?"
- "I want it to make all the entries above and to the right of it `x`, not this stuff"
- "What is the relationship between the row of player and the column of compute?"
- "In Dijkstra's algorithm, once a vertex is visited, does that mean that we have found the shortest path from the starting point to the vertex?"