

Kazakh British Technical University

Almaty, Kazakhstan

**SUMMER SEMESTER**

2022

TSIS 4

Teams No: 6

Students Name and ID:

Abdurakhim Kenesbekov 20B030459

Zhanseitov Altair 20B030308

Dilnaz Masakbai 20B030483

Ruslan Kashaev 20B030550

Abdipattaev Azimbek 20B030344

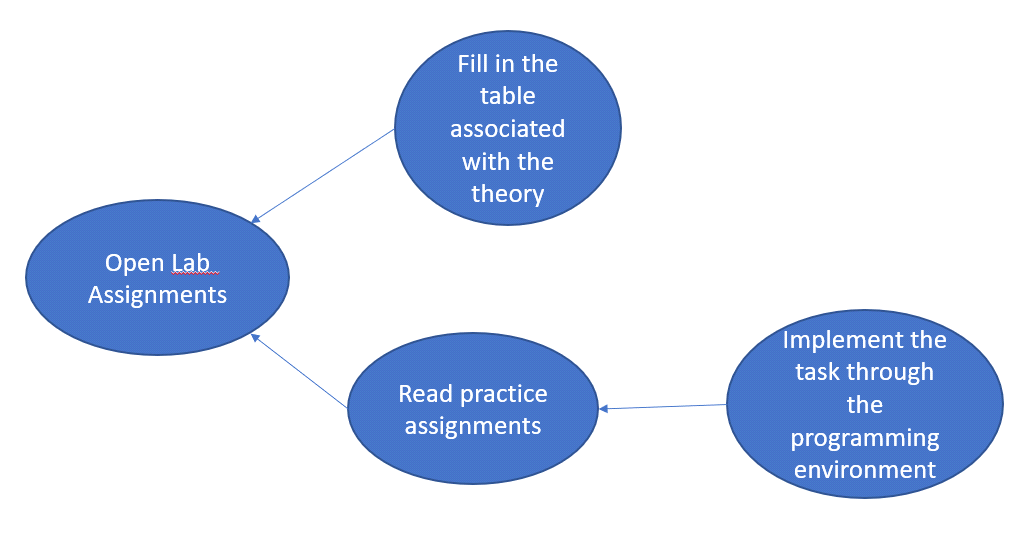
Zeinolla Kamila 18BD120248

Date: 7­­­ Jule 2022

1-8 Fill in the table by answering the questions (each question 4 points)

|  |  |
| --- | --- |
| **Questions** | **Answer** |
| Working time estimates  O(n) O(m) O(m+n) O(n^2) O(mn)  1 A connected acyclic graph is given by adjacency lists. Which of the running time estimates (which are written above) is true for a wide search? | 1. The Time complexity of BFS is O(m + n) when Adjacency List is used. |
| 2 A connected acyclic graph is given by a adjacency matrix. Which of the following running time estimates (which are written above) is true for a wide search? | 2. The Time complexity of BFS is O(n^2) when Adjacency Matrix is used, where n stands for vertices and m stands for edges.  Why is it more in the case of Adjacency Matrix?  This is mainly because every time we want to find what are the edges adjacent to a given vertex 'U', we would have to traverse the whole array AdjacencyMatrix[U], which is of course of length |n|. |
| 3 Four friends live in the same neighbourhood. Arystan and the security guard are older than Sozakbai, Tumanbai and the plumber play chess, and the locksmith is the youngest of the friends. At lunchtime, Ayukhan and the electrician play backgammon against Sozakbai and the locksmith.  What is the profession of each of the friends (define through the graph) | 3. Arystan - electrician  Sozakbai - plumber  Tumanbai - locksmith  Ayukhan – security guard  Diagram  Description automatically generated |
| 4 There are seven trees in the schoolyard: apple, birch, mountain ash, oak, maple, larch and pine. The mountain ash is taller than the larch, the apple is taller than the maple, the oak is lower than the birch but higher than the pine, the pine is higher than the mountain ash and the larch is higher than the apple. Identify the trees from the lowest to the highest by applying an oriented graph. | 4.  Diagram  Description automatically generated |
| 5 Four classmates - Zhambyl, Miras, Ultarak and Elemes - are selected by the class assembly to work in the administrative, cultural, sports and educational sectors of the school.  - Zhambyl and a schoolboy from the cultural sector live in the same house, and the three of them take the bus to school together with Ultarak;  - a schoolboy from the educational sector and Yelemes go to the basketball area together;  - it was decided at a class meeting to elect a boy to the sports section;  - Ultarak sits at the same desk as a schoolboy from the educational sector;  - A schoolboy from the sports sector and Zhambyl are friends.  Which of the children in which of the named sectors are chosen through the graph. Which set should be used? | 5. |
| 6 Using the depth-first search algorithm, find the connectivity components of a graph given by the adjacency lists:  a: b, d, f;  b: a;  c: e, i;  d: a;  e: c, i;  f: a;  g: h, j;  h: g, j, k;  i: c, e;  j: g, h, l;  k: h,l;  l: j, k | 6. There are 3 connectivity components:  a b d f  c e i  g h l j k |
| 7 How many abstract ordinary graphs with 5 vertices and 3 edges are there? | 7. 0 such graphs  2\*(number of edges) = sum of degrees. A graph cannot have a non-integer number of edges such as 7.5, so there is no way for there to be a 3-regular graph on 5 vertices |
| 8 Find the shortest paths from vertex S (teams with an even number) and vertex Y (teams with an odd number) | 8 . Even numbered paths:  From S to T = 8  From S to Y = 10  From S to X = 14  From S to Z = 14    Odd numbered paths:  From Y to S = 9  From Y to T = 3  From Y to X = 9  From Y to Z = 13 |

10-12 In front of you is a graph for timetable for doing any TSIS. (each question 3 points)



The graph shows that it is possible to implement the task through the programming environment by reading the condition of the practical task. Thus, the node " Implement the task through the programming environment" depends on the node "Read practice assignments".

"Fill in the table associated to the theory" is independent of " Read practice assignments", because I can first "Fill in the table related to the theory" and then " Read practice assignments".

Based on the graph, it is possible to formulate the order in which I act in the morning:

1. Open Lab Assignments.

2. Fill in the table associated with the theory

3. Read practice assignments

4. Implement the task through a programming environment.

Note that the "Fill in the table related to theory" action can be moved around in the list, so the next list is also valid:

1. Open Lab Assignments

2. Read practice assignments

3. Fill in the table associated with the theory

4. Implement the task through the programming environment

For each of the following three lists, indicate whether it is valid or not valid.

|  |  |  |
| --- | --- | --- |
| List 1  1. Open Lab Assignments.  2. Fill in the table associated with the theory  3. Implement the task through the programming environment  4. Read practice assignments | List 2  1. Open Lab Assignments.  2. Read practice assignments  3. Implement the task through the programming environment  4. Fill in the table associated with the theory | List 3  1. Fill in the table associated with the theory  2. Open Lab Assignments  3. Read practice assignments  4. Implement the task through the programming environment |
| 10 Valid/non-valid | 11 Valid/not valid | 12 Valid/not valid |

10. Non-valid

11. Valid

12. Non-valid

13 Construct a valid list for this graph. (each question 3 points)



13:

1. Open lab assignments

2. Read terms related to theory

3. Fill in the table associated with the theory

4. Read practice assignments

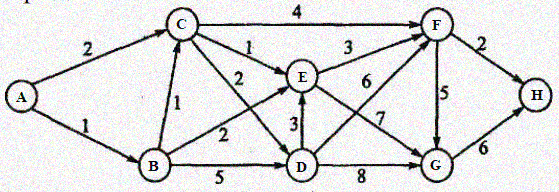
5. Fill in the number and team members

6. Implement the task through the programming environment

7. Add code and screenshot to the report

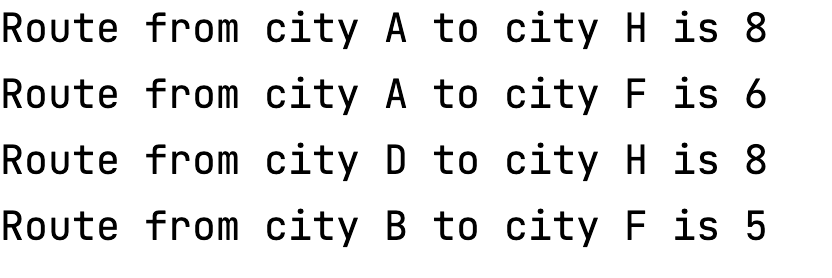
2) Practical task

1Dijkstra's Algorithm transport network connecting eight cities and the distances between them is shown.

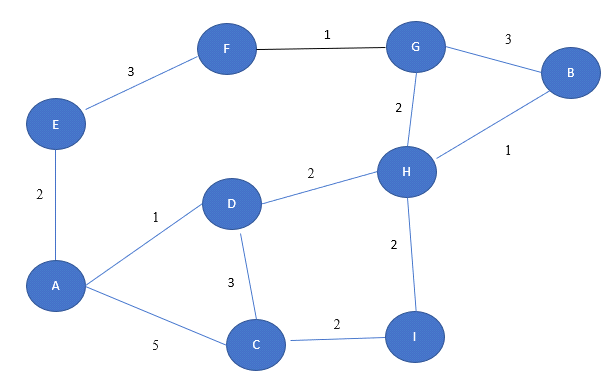


Find the shortest routes between the following cities (16 points):

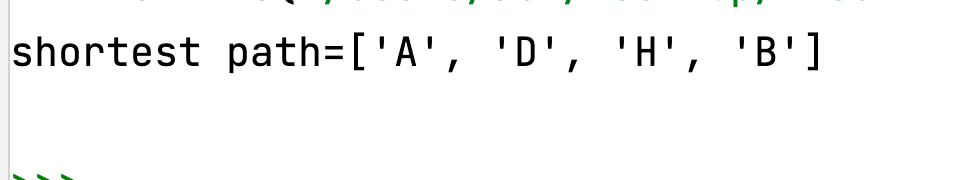
* Between cities A and H.
* Between cities A and F.
* Between cities D and H.
* Between cities B and F.
* from queue import PriorityQueue  
    
    
  class Graph:  
   def \_\_init\_\_(self, num\_of\_vertices):  
   self.v = num\_of\_vertices  
   self.edges = [[-1 for i in range(num\_of\_vertices)] for j in range(num\_of\_vertices)]  
   self.visited = []  
    
   def add\_edge(self, u, v, weight):  
   self.edges[u][v] = weight  
   self.edges[v][u] = weight  
    
    
  def dijkstra(graph, start\_vertex):  
   D = {v: float('inf') for v in range(graph.v)}  
   D[start\_vertex] = 0  
    
   pq = PriorityQueue()  
   pq.put((0, start\_vertex))  
    
   while not pq.empty():  
   (dist, current\_vertex) = pq.get()  
   graph.visited.append(current\_vertex)  
    
   for neighbor in range(graph.v):  
   if graph.edges[current\_vertex][neighbor] != -1:  
   distance = graph.edges[current\_vertex][neighbor]  
   if neighbor not in graph.visited:  
   old\_cost = D[neighbor]  
   new\_cost = D[current\_vertex] + distance  
   if new\_cost < old\_cost:  
   pq.put((new\_cost, neighbor))  
   D[neighbor] = new\_cost  
   return D  
    
    
  def as\_index(input\_char):  
   return ord(input\_char) - ord("A")  
    
    
  adj\_list = {  
   'A': {'B': 1, 'C': 2},  
   'B': {'C': 1, 'E': 2, 'D': 5},  
   'C': {'D': 2, 'E': 1, 'F': 4},  
   'D': {'E': 3, 'F': 6, 'G': 8},  
   'E': {'F': 3, 'G': 7},  
   'F': {'G': 5, 'H': 2},  
   'G': {'H': 6},  
  }  
  routes = [  
   ["A", "H"],  
   ["A", "F"],  
   ["D", "H"],  
   ["B", "F"]  
  ]  
    
  for start\_V, end\_V in routes:  
   g = Graph(8)  
   for fromV, toVs in adj\_list.items():  
   for toV, cost in toVs.items():  
   g.add\_edge(as\_index(fromV), as\_index(toV), cost)  
    
   D = dijkstra(g, as\_index(start\_V))  
   print(f"Route from city {start\_V} to city {end\_V} is {D[as\_index(end\_V)]}")



2 Dijkstra's algorithm Find the shortest and longest path from point A to point B (12 points)

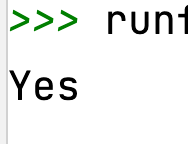


def search(fromV, toV, graph, costs, parents):  
 next\_node = fromV  
  
 while next\_node != toV:  
 for neighbor in graph[next\_node]:  
 *# if graph[next\_node][neighbor] + costs[next\_node] > costs[neighbor]:* if graph[next\_node][neighbor] + costs[next\_node] < costs[neighbor]:  
 costs[neighbor] = graph[next\_node][neighbor] + costs[next\_node]  
 parents[neighbor] = next\_node  
 del graph[neighbor][next\_node]  
 del costs[next\_node]  
  
 *# next\_node = max(costs, key=costs.get)* next\_node = min(costs, key=costs.get)  
  
 return parents  
  
  
def backpedal(fromV, toV, search\_result):  
 current\_node = toV  
 path = [toV]  
 reverse\_path = []  
  
 while current\_node != fromV:  
 path.append(search\_result[current\_node])  
 current\_node = search\_result[current\_node]  
  
 for i in range(len(path)):  
 reverse\_path.append(path[-i - 1])  
  
 return reverse\_path  
  
  
adj\_list = {  
 'A': {'C': 5, 'D': 1, 'E': 2},  
 'B': {'H': 1, 'G': 3},  
 'C': {'A': 5, 'D': 3, 'I': 2,},  
 'D': {'A': 1, 'C': 3, 'H': 2},  
 'E': {'A': 2, 'F': 3},  
 'F': {'E': 3, 'G': 1},  
 'G': {'B': 3, 'F': 1, 'H': 2},  
 'H': {'B': 1, 'D': 2, 'G': 2, 'I': 2},  
 'I': {'C': 2, 'H': 2}  
}  
  
inf = float('inf')  
costs = {'A': 0, 'B': inf, 'C': inf, 'D': inf, 'E': inf, 'F': inf, 'G': inf, 'H': inf, 'I': inf}  
*# costs = {'A': 0, 'B': -inf, 'C': -inf, 'D': -inf, 'E': -inf, 'F': -inf, 'G': -inf, 'H': -inf, 'I': -inf}*parents = {}  
result = search('A', 'B', adj\_list, costs, parents)  
print(f"shortest path={backpedal('A', 'B', result)}")  
*# print(f"longest path={backpedal('A', 'B', result)}")*



3 Develop an algorithm to check whether the given graph is bipartite (a graph whose set of vertices can be divided into two parts such that each edge of the graph connects each vertex from one part to some vertex of the other part, i.e. there are no edges between vertices of the same part of the graph). (11 points)

def isBipartite(vertex, adj):  
 color = [-1] \* vertex  
 my\_queue = []  
  
 for i in range(vertex):  
 if color[i] == -1:  
 my\_queue.append([i, 0])  
 color[i] = 0  
  
 while len(my\_queue) != 0:  
 p = my\_queue[0]  
 my\_queue.pop(0)  
  
 current\_vertex = p[0]  
 color\_current\_vertex = p[1]  
  
 for vert in adj[current\_vertex]:  
 if color[vert] == color\_current\_vertex:  
 return False  
  
 if color[vert] == -1:  
 if color\_current\_vertex == 1:  
 color[vert] = 0  
 else:  
 color[vert] = 1  
 return True  
  
  
V, E = 4, 8  
adj\_list = [[1, 3], [0, 2], [1, 3], [0, 2]]  
print("Yes" if isBipartite(V, adj\_list) else "No")



4 A practice question on the **Fillword** puzzle. The condition guarantees that all the words in the list can be crossed out in the **Fillword** puzzle, then to get the answer it is enough to count the number of each letter in the crossword puzzle, and then going through the words that are fed to the input, reduce the corresponding counter by 1. At the end you just need to display the letters on the screen. Try via width or depth search (17 points)

def fill\_word\_puzzle(graph, word, filled):  
 my\_queue = []  
 for letter in word:  
 my\_queue.append(ord(letter) - ord("A"))  
 fromV = my\_queue[0]  
  
 while my\_queue:  
 toV = my\_queue.pop(0)  
 if toV >= len(graph):  
 break  
 array = graph[toV]  
 filled[toV] = True  
 if not my\_queue or not array[my\_queue[0]]:  
 break  
 if not my\_queue:  
 print("Found!", end=" ")  
 count\_letters(graph, filled, fromV)  
 else:  
 print(f"{word} wasn't found")  
  
  
def count\_letters(graph, filled, fromV):  
 print("Frequency of letters:")  
 size = len(graph)  
 visited = [False] \* size  
 my\_queue = [fromV]  
 while my\_queue:  
 toV = my\_queue.pop(0)  
 arr = graph[toV]  
 print(f"{chr(toV + ord('A'))} with 1" if filled[toV] else f"{chr(toV + ord('A'))} with 0")  
 for i in range(size):  
 if arr[i] != 0:  
 if not visited[i]:  
 my\_queue.append(i)  
 visited[toV] = True  
  
  
word\_graph = [  
 [0, 1, 0, 0, 0],  
 [0, 0, 1, 0, 1],  
 [1, 1, 0, 0, 1],  
 [0, 0, 0, 0, 0],  
 [0, 1, 0, 1, 0]  
]  
  
filled = [False] \* len(word\_graph)  
count\_letters(word\_graph, filled, 0)  
fill\_word\_puzzle(word\_graph, "EBCA", filled)

