Design Document

Table of Contents

##### [Runtime Analysis of every Method Used: 1](#_Toc75252187)

##### [Algorithm Chosen: 1](#_Toc75252188)

##### [Alternative Approach: 1](#_Toc75252189)

##### [Methods That constitute for the Algorithm to work (Helper Methods): 2](#_Toc75252190)

### Runtime Analysis of every Method Used:

Function: display\_trainers

Analysis: O(n2)+c

Function: show\_all

Analysis: O(n) + O(n) => O(n)+c

Function: display\_recruit\_list

Analysis: O(n2) + O(n2) + c => O(n2)+c

### Algorithm Chosen:

Adjacency Matrix is chosen as the algorithm implementation strategy as the lookup needs to be done fast and assign weights for each edge.

Adjacency Matrix

* Uses O(n^2) memory
* It is fast to look up and check for presence or absence of a specific edge  
  between any two nodes O(1)
* It is slow to iterate over all edges
* It is slow to add/delete a node; a complex operation O(n^2)
* It is fast to add a new edge O(1)

### Alternative Approach:

Adjacency List

* Memory usage depends more on the number of edges (and less on the number of nodes), which might save a lot of memory if the adjacency matrix is sparse
* Finding the presence or absence of specific edge between any two nodes  
  is slightly slower than with the matrix O(k); where k is the number of neighbour nodes
* It is fast to iterate over all edges because you can access any node neighbour directly
* It is fast to add/delete a node; easier than the matrix representation
* It fast to add a new edge O(1)

## Methods That constitute for the Algorithm to work (Helper Methods):

Helper 1:

‘’’ this is the method is a helper method for the main methods which will read the input file and helps in parsing the data from the input file‘’’

def read\_input(self, input\_file):  
 try:  
 with open(input\_file, 'r') as f:  
 self.lines = f.read().splitlines()  
  
 for line in self.lines:  
 line\_data = line.split('/')  
 self.trainerSubjects.append((line\_data[0], line\_data[1:]))  
  
 self.trainerSubjects = sorted(self.trainerSubjects)  
  
 self.create\_uniq\_list\_of\_subjects()  
 self.create\_uniq\_list\_of\_trainers()  
 self.create\_graph()  
  
 self.show\_all()  
  
 except Exception as e:  
 print("Error" + e.\_\_str\_\_())

Helper 2:

‘’’ This is the method that will create a unique list of subjects for further analysis in cost calculation‘’’

def create\_uniq\_list\_of\_subjects(self):  
 first\_list = []  
 self.total\_uniq\_subjects = list(first\_list)  
 for t in self.trainerSubjects:  
 self.total\_uniq\_subjects.extend(x for x in t[1] if x not in self.total\_uniq\_subjects)  
  
 self.total\_uniq\_subjects = sorted(self.total\_uniq\_subjects)

Helper 3:

‘’’ This is the method that will create a unique list of trainers for further analysis in cost calculation‘’’

def create\_uniq\_list\_of\_trainers(self):  
 for t in self.trainerSubjects:  
 self.total\_uniq\_trainers.append(t[0])  
 self.total\_uniq\_trainers = sorted(list(set(self.total\_uniq\_trainers)))

Helper 4:

‘’’ This is the helper method to create a graph as a adjacency matrix‘’’

def create\_graph(self):  
 self.edges = [[0 for j in range(len(self.total\_uniq\_subjects))] for i in range(len(self.total\_uniq\_trainers))]  
  
 for i in range(len(self.total\_uniq\_trainers)):  
 for j in range(len(self.total\_uniq\_subjects)):  
 if self.trainerSubjects[i][0] == self.total\_uniq\_trainers[i] and self.total\_uniq\_subjects[j] in \  
 self.trainerSubjects[i][1]:  
 self.edges[i][j] = 1