Project description

Proposed software application is based on watching a maximum number of movies by visiting nearby theaters with minimal possible cost by our user. This application mainly has three functionalities which solves the problem for users. We will go through each of the functionality and an overview how I would approach to the solution of the problem

1) **Finding nearest movie theater you can reach**

As part of this functionality, we need to find the nearest movie theater user can reach from his existing location. We can represent a graph where nodes of graphs will be all theaters (or intermediate junctions) available in N km radius of the user location and edges which have time and cost it takes to reach from one node to another. Input for this functionality will be a graph which represents possible ways you can reach all theaters within N km radius along with time and cost for each edge. We need to determine the nearest movie theater user can reach by travelling through any possible route available. This problem can be solved using **dijstra's** shortest path algorithm.

Implementation details:

Input 1: Input for graph is taken from a CSV file named graph.csv with header **location1**,**location2**,**cost**,**time_taken_in_sec**
This file contains information of the entire map with theatres and intermediate locations which can be used to reach the theater.
Input 2: Another file nodes.csv contain information about location name and a boolean which represents whether the location is theatre or not
Input 3: current location name of user

Algorithm:

1. Read locations file and construct Map of location to object Node which contains information about location name and whether its a theater or not
2. Read graph file and construct Map of Node to List of Vertex(which contains information about node, time and cost it takes from the other node). This is representing information in a weighted graph form
3. Compute shortest time it takes from user's current location to rest of all locations using **Dijstra's** algorithm
4. Filter and pick nearest theatre among all possible locations user can travel

Reason why this algorithm is chosen:

As part of this functionality, we have locations and information of time and cost between each of them. Locations can be treated as nodes in graphs and information between locations is edge. This can be looked as an undirected weighted graph. Now the problem is narrowed down to

finding the nearest node from a given node in an undirected weighted graph. We can achieve this by applying **Dijstra's** algorithm and compute shorted time it takes to travel from users location to all other locations

Time complexity Analysis:

Let's assume number of locations are in order of L and number of all possible paths are in P (number of vertices in graph)

Loading all locations into a node map takes an iteration of all locations. So time complexity for this step is O(L) ---- (1)

Constructing graph Map from graph input needs to be iterated through all possible paths(vertices). So time complexity for this step is O(V) --- (2)

We iterate through Step 3 mentioned in the Algorithm section for L times. In step 3, for a given location we iterate through all the possible paths connecting all different locations which we didn't iterate in the earlier steps. So overall at the end of all L iterations, this step will cover all possible paths for all the locations which is O(V) -- (3)

We also compute the minimum distance node in the current iteration to use in the next iteration. This takes O(L) for each iteration and we do this L times. So, total complexity for this step is O(L^2) --- (4)

Final complexity for this algorithm is the sum of all the above steps.

(1) + (2) + (3) + (4) = O(L) + O(V) + O(V) + O(L^2)
= O(L) + 2*O(V) + O(L^2)
= O(L^2) + O(V) (as we know 2*O(V) = O(V) and O(L) + O(L^2) = O(L^2))
= O(V + L^2)

So, time complexity is O(V + L^2)

Empirical time analysis:

Performance test results

---- Theatre finder test started -------
Nearest theatre is { 'name': a4, 'is_theatre': true}. Travel time is: 105
Test 1 took(5 edges) 16 ms
Nearest theatre is { 'name': a50, 'is_theatre': true}. Travel time is: 2521
Test 2 took(201 edges) 15 ms
Nearest theatre is { 'name': a27, 'is_theatre': true}. Travel time is: 24
Test 3 took(1001 edges) 39 ms
Nearest theatre is { 'name': a75, 'is_theatre': true}. Travel time is: 41

Test 4 took(10001 edges) 81 ms
Nearest theatre is { 'name': a109, 'is_theatre': true}. Travel time is: 41
Test 5 took(15001 edges) 45 ms
Average time for all test cases: 39ms
---- Theatre finder test ended -------

2) **Drive to nearest theater within given time with minimal possible cost**

As part of this functionality, the user should get the exact path he needs to travel inorder to reach the theater within desired time with minimal possible cost. Input would be a similar graph which was mentioned in the first functionality. Even though a path might take you the fastest time to reach the theater but if the cost is higher and if there is an alternate path which can minimize cost and take to theater within desired should be preferred. The idea is to solve this problem in recursive approach and use **dynamic programming** to avoid recomputing intermediate result for efficient solution

Implementation details:

Input 1: Input for graph is taken from a CSV file named graph.csv with header **location1**,**location2**,**cost**,**time_taken_in_sec**
This file contains information of the entire map with theatres and intermediate locations which can be used to reach the theater.
Input 2: Another file nodes.csv contain information about location name and a boolean which represents whether the location is theatre or not
Input 3: current location name of user
Input 4: Theatre location name
Input 5: Duration in seconds (duration in which user needs to reach in theatre)

Algorithm:

1. Read locations file and construct Map of location to object Node which contains information about location name and whether its a theater or not
2. Read graph file and construct Map of Node to List of Vertex(which contains information about node, time and cost it takes from the other node). This is representing information in a weighted graph form
3. Compute path which has lowest cost within given duration using recursive approach with dynamic programming to avoid recomputation

Reason why this algorithm is chosen:

As part of this functionality, we have locations and information of time and cost between each of them. Locations can be treated as nodes in graphs and information between locations is edge. This can be looked as an undirected weighted graph. Given the user's location and theatre

location we need to compute the path with minimum cost involved and also the user can reach the theatre within the user's duration time. This problem is solved by recursive solution on finding cost from all the connected locations to the theatre node recursively and taking minimum of them. As this is a recursive solution and does computation on any node to find the minimum path several times, so to avoid this we used **dynamic programming** to store intermediate results and use it to avoid recomputation.

Time complexity Analysis:

Let's assume number of locations are in order of L and number of all possible paths are in P (number of vertices in graph)

Loading all locations into a node map takes an iteration of all locations. So time complexity for this step is O(L) ---- (1)

Constructing graph Map from graph input needs to be iterated through all possible paths(vertices). So time complexity for this step is O(V) --- (2)

To compute worst case time complexity, each iteration we can take there are V paths from source node and there are L locations connected from the source location.
We can take T(V) = L*T(V-1) recursively because first iteration goes through all L nodes in worst case and call each node recursively which can have V-1 vertices in the worst case as we keep track of visited vertex and dont visit that again

T(V) = L*T(V-1) = L* L* T(V-2) (As T(V-1) = L*T(V-2))
=> T(V) = L * L * …*L (V times)
=> T(V) = O(L^V) in the worst case (Dynamic programming can bring down this number down when there are recomputation happening from a given node with given duration. But worst case will still be this) --- (3)

Final complexity for this algorithm is the sum of all the above steps.
(1) + (2) + (3) = O(L) + O(V) + O(L^V) = O(L^V)

So, worst case upper bound is O(L^V)

Empirical time analysis:

Performance test results

---- Theatre path finder test started -------
Minimum path and cost within given duration: { 'path': a1 -> a2 -> a3 -> a4, 'duration': 24}
Test 1 took(5 edges) 2 ms
Minimum path and cost within given duration: { 'path': a1 -> a18 -> a21 -> a4, 'duration': 1878}
Test 2 took(201 edges) 2 ms

Minimum path and cost within given duration: { 'path': a1 -> a40 -> a6 -> a10, 'duration': 1122}
Test 3 took(1001 edges) 20 ms
Minimum path and cost within given duration: { 'path': a1 -> a128 -> a175 -> a19 -> a20, 'duration': 959}
Test 4 took(10001 edges) 2778 ms
Minimum path and cost within given duration: { 'path': a1 -> a143 -> a109 -> a129 -> a88 -> a78, 'duration': 628}
Test 5 took(15001 edges) 23307 ms
Average time for all test cases: 5221ms
---- Theatre path finder test ended -------

3) **Watch maximum number of movies possible within the allocated time**

Once a user reaches the theater by time Tstart and he has time until Tend, this functionality should return a maximum number of movies with movie names to watch. Input to this functionality will be all the shows it will play for that particular day for that particular theater. Each show will be represented with start time and duration and input will be a collection of shows. To solve this problem, we can use **heaping** to achieve the solution and give the user the maximum number of movies he can watch within his allocated time.

Implementation details:

Input 1: Filename which has entire information on show startTime and duration of each show
Header of the CSV file "**Movie name,Start time,Duration**"
Input 2: Start time when user reached theater
Input 3: End time till when user can spend watching movies

Algorithm:

1. Read showtimes file and construct List of ShowTime object
2. Sort the list if not sorted already
3. Start from the movie which starts after startTime and start pushing it into **priority queue with min heap** implementation on endTime of the movie
4. When we try to push the next movie into priority queue, we need to check if the top element's endTime is less than current movie's start time else we need to push this as well. If the top movie is already seen then we need to take the next top element to check the condition.
5. If priority queues top element is less than startTime of next movie, we need to consider top movie is part of our solution set and empty the priority queue and start over again
6. Repeat the process until startTime of the movie is less than the users endTime

Reason why this algorithm is chosen:

As part of this functionality, we need to find the maximum number of movies users can watch within start and end time. To get the maximum number of movies we need take the movie with least endTime from all the movies that start before that end time. We can do this by sorting on end Times for all the movies that started before. This can be achieved with priority queue with min heap on movie end time as we start push movies as it comes and min heap makes sure that movie which ends first will be on top. Whenever we check the next movie, it will be O(1) to fetch and check if there is any movie which ends before the current movie.

Time complexity Analysis:

Lets us assume number of shows is in order of N

To get shows list and load it into a list from file takes O(N) --- (1)

Iterate through all shows and insert each show in a priority queue which implements min heap. If priority queues top element is less than startTime of next movie, we need to consider top movie is part of our solution set and empty the priority queue and start over again

Insert an element in min heap is O(log N)
In the worst case we might have to insert all possible shows so time complexity will be O(N*log N) --- (2)

Total time complexity: O(N) + O(N*log N) ((1) + (2)) = O(N*log N)

Empirical time analysis:

Performance test results

---- Movie Lister test started -------
Maximum number of movies user can watch 4
Test 1 took(9 records) 35 ms
Maximum number of movies user can watch 425
Test 2 took(1001 records) 60 ms
Maximum number of movies user can watch 644
Test 3 took(3001 records) 59 ms
Maximum number of movies user can watch 745
Test 4 took(5001 records) 44 ms
Maximum number of movies user can watch 890
Test 5 took(10001 records) 57 ms
Average time for all test cases: 51ms
---- Movie Lister test ended -------