Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Master Thesis

Τίτλος Διατριβής Title of Thesis	**Μια Android εφαρμογή για το «Ερώτημα Σχεδιασμού Ταξιδιού».** **An android application for the "Trip Planning Query".**
Ονοματεπώνυμο Φοιτητή/ Full name of student	**Κων/νος Καββαλάκης** **(Konstantinos Kavvalakis)**
Πατρώνυμο/ Father's name	**Στέφανος**
Αριθμός Μητρώου	**ΜΠΣΠ/10017**
Επιβλέπων/Supervisor	**Νικόλαος Πελέκης** **(Nikolaos Pelekis),Λέκτορας**

Ημερομηνία Παράδοσης **Οκτώβριος 2013**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Νικόλαος Πελέκης Λέκτορας	Ιωάννης Θεοδωρίδης Καθηγητής	Ιωάννης Σίσκος Καθηγητής

ABSTRACT

The objective of the current master thesis is the implementation of an efficient android application that utilizes the *Iterative Doubling* algorithm. It runs in a mobile device with known computational processing limitations. Finally as the user of the application imposes, the optimal path should start from a specified source, pass through several points of interest (POIs) and reach specified destination. These problems are generally known as “Trip Planning Queries” (TPQ).

Trip Planning Queries (TPQ) belong to NP-hard problems. In general there are many proposed algorithms which result in an approximate or inefficient in terms of complexity solution. The previously mentioned drawbacks prevent the use of those algorithms in many implementations. However the solution presented in this thesis reduces complexity by imposing the order of visiting locations. The algorithm which is used manages to find a route of minimum length by visiting points from each category sequentially before it steps to the next category. This type of queries are named “Sequenced Route Queries” (SRQ) and their implementation could be feasible even in a mobile device as it is proposed and utilized by an android application in the current thesis.

The algorithm used is the *Iterative Doubling*, an alternative to the EDJ(Enhanced Dijkstra - section 2.3.3) layered approach. The current approach helps to avoid exploring facilities that are far away from the starting point. The algorithm manages to reduce the computational cost of exploring every possible facility and therefore it performs better in case that we have to process local data. Realistic sequenced route queries mainly refer to such local geographical data.

The application manages to exclude such distant facilities by setting an initial threshold that defines a range in kilometers. Also the user has as an option to choose between some initial ranges, depending on the minimum trip distance that he is willing to cover. With this threshold nodes of POIs without range does not take part during Dijkstra computations and so this fastens the running time. If the initial range although could not resume to an optimal path then the threshold is doubled and the algorithm is executed again.

The data for the needs of the application were collected from OpenStreetMaps project [13] and mainly concern mainly Attica. They represent several geographical points and each of them belongs to a respective category of interest such as hotels, restaurants etc. The basic data element keeps information about its geographic coordinates, the label that identifies the facility and its category. They are initially formed as xml separated files (per category) and they are utilized as node objects. Another advantage of the implemented algorithm is that it does not require the explicit construction of the graph that is built during each Dijkstra’s computation. The final outcome is a route with the minimum distance that passes through a point from each category.

ΠΕΡΙΛΗΨΗ

Ο σκοπός της παρούσας μεταπτυχιακής διατριβής είναι η υλοποίηση μιας android εφαρμογής που θα ενσωματώνει τον αλγόριθμο «Επαναληπτικού Διπλασιασμού», κατάλληλο όπως προτείνεται να τρέχει αποδοτικά σε μια κινητή συσκευή με δεδομένους επεξεργαστικούς περιορισμούς. Η εφαρμογή που υλοποιήθηκε υπολογίζει την βέλτιστη διαδρομή, η οποία όπως απαιτεί ο χρήστης πρέπει να ξεκινάει από ένα σημείο αρχής, να περνάει ακολουθιακά από διαφορετικά σημεία ενδιαφέροντος και καταλήγει σε έναν επιθυμητό προορισμό. Αυτή η κατηγορία προβλημάτων είναι γνωστή σαν Trip Planning Queries (TPQ).

Τα ερωτήματα που αφορούν το σχεδιασμό διαδρομών(TPQ) είναι μη ντετερμινιστικά πολυωνυμικού χρόνου προβλήματα(NP problems).Γενικά πολλοί προτεινόμενοι αλγόριθμοι παρουσιάζουν λύσεις είτε προσεγγιστικές είτε μεγάλης πολυπλοκότητας με αποτέλεσμα να μην είναι χρηστικοί στην υλοποίηση. Παρόλα αυτά η παρούσα εφαρμογή χαλαρώνει την πολυπλοκότητα του προβλήματος, καθορίζοντας την σειρά προσπέλασης των σημείων ενώ ψάχνει να βρει την διαδρομή με το ελάχιστο μήκος με το να επισκέπτεται ακολουθιακά τα σημεία από την μια κατηγορία στην επόμενη. Αυτά του τύπου τα ερωτήματα ονομάζονται «ερωτήματα διαδοχικής διαδρομής» (SRQ) και η απάντηση τους θα μπορούσε να είναι εφικτή και σε κινητές συσκευές όπως προτείνετε και υλοποιείται από την παρούσα εργασία.

Ο αλγόριθμος «Επαναληπτικού Διπλασιασμού» (Iterative Doubling) είναι μια παραλλαγή του EDJ (Enhanced Dijkstra- ενότητα 2.2.3), και βασίζεται στο να αποφεύγει να εξερευνάει απομακρυσμένα σημεία. Με τον τρόπο αυτό μειώνει τον επεξεργαστικό όγκο και αποδίδει καλύτερα για τοπικά δεδομένα που αποτελούν και τις πιο συνηθισμένες περιπτώσεις σε τέτοιου είδους προβλήματα. Στην πράξη τέτοια προβλήματα εύρεσης βέλτιστης διαδρομής με ενδιάμεσους προορισμούς συνήθως συναντώνται σε τοπική ακτίνα.

Τα δεδομένα αυτά συλλέχτηκαν από το OpenStreetMaps [13] και αφορούν το λεκανοπέδιο Αττικής. Αναπαριστούν γεωγραφικά δεδομένα που ανήκουν σε συγκεκριμένες κατηγορίες ενδιαφέροντος. Αυτή η κατηγοριοποίηση μπορεί να θεωρηθεί σαν ένα σύνολο τέτοιων σημείων με παρόμοια χαρακτηριστικά όπως τράπεζες, εστιατόρια κτλ. Η βασική μονάδα δεδομένου κρατάει την πληροφορία των συντεταγμένων κάθε σημείου, τον τίτλο της εγκατάστασης και την κατηγορία που ανήκει. Η εφαρμογή φιλοξενεί αυτά τα δεδομένα σε στατικά με τη μορφή xml αρχείων. Ανάλογα με τις επιλογές του χρήστη τα κατάλληλα αρχεία διατρέχονται ώστε να δομήσουν δυναμικά τον ζητούμενο γράφο . Δεν είναι απαραίτητα η άμεση δημιουργία του.Τέλος η εφαρμογή καταφέρνει και αναπτύσσει την βέλτιστη διαδρομή που περνάει από ένα τουλάχιστον σημείο ανά κατηγορία ενδιαφέροντος.

CONTENTS

1. Introduction

Efficient TPQ evaluation could become an important new feature of advanced navigation systems and can prove useful for other geographic applications as has been advocated in previous work [12]. For instance, state of the art mapping services like MapQuest, Google Maps, Google Directions service and Microsoft Streets & Trips, currently support queries that specify a starting point and only one destination, or a number of user specified destinations. The functionality and usefulness of such systems can be greatly improved by supporting more advanced query types, like TPQ.[1]

TPQ can be considered as a generalization of the Traveling Salesman problem (TSP) which is NP-hard. The reduction of TSP to TPQ is straightforward. By assuming that every point belongs to its own distinct category, any instance of TSP can be reduced to an instance of TPQ. From the current spatial database queries, TPQ is mostly related to the time or the distance which is parameterized in the continuous Nearest Neighbors (NN) queries. In those we assume that the query point is moving with a constant velocity and the goal is to incrementally report the nearest neighbors over time as the query moves from an initial to a final location. However, none of the methods developed to answer the above queries can be used to find a “good” solution for TPQ.[1]

Usually, TPQ problems are not being efficiently handled from embedded navigation applications in mobile devices, due to their NP-hard algorithmic complexity. This kind of devices, have limited computational resources and capabilities, so most of the time mobile applications provide an already computed result of such Travel Planning Queries. Moreover, we can further reduce the above TPQ problem to SRQ problem by assuming that while traveling the order of the visited categories is *predefined*. As it is proved, finally leveraging such queries with proper algorithms and a proper mobile database (in this case SQLite) we can develop embedded SRQ algorithms in mobile devices that can run efficiently. Such an algorithm is proposed in [3] and has been implemented by the current application that the Thesis introduces.

The goal is of the thesis application is to plan an optimal route from a source to a destination point by visiting facilities of a perspective type on the way (SRQ case). The speed up technique that is implemented is *Iterative Doubling.* This approach does not assume the explicit construction of a graph and reduces radically the Dikjstra computations to compute the weights of inner-layers edges. The application modifies the EDJ layer approach that requires as many computations as the cardinality of the available points of interest (POIs). The following chapter explain in details the above algorithms along with some others.

To clearly determine the problem that the application solves, we have to start describing the data sets that were used for the experimental results and the preprocessing stage before algorithm starts. The data were collected from the OpenStreetMap project in xml file format. They are labeled and hold their geographic coordinates (latitude, longitude). The application serialize these files to construct lists of POI objects that then holds in main memory for quick access. These POI objects finally are transformed to node objects during the graph construction.

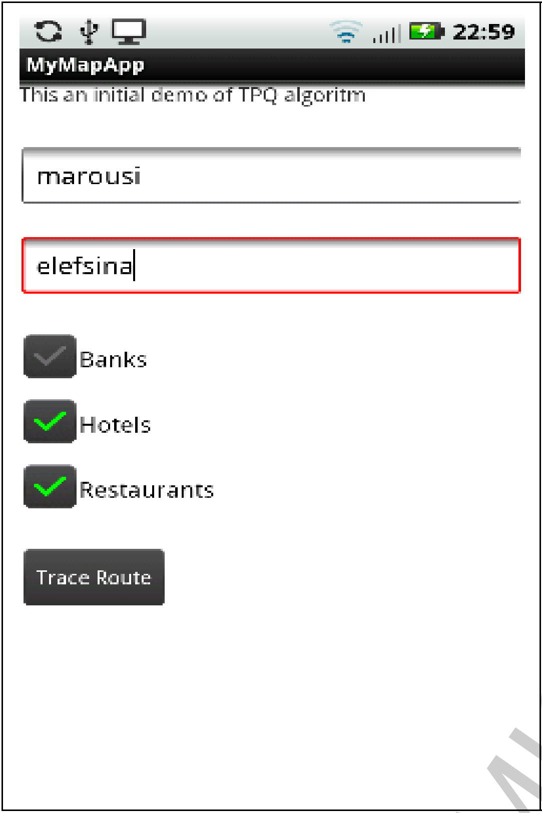
The preprocess stage is triggered by the user who inputs a starting point S and a destination point E. Then chooses from a set of categories R with prefixed order, (R ≤

C). The application after the algorithm execution returns the optimal route that starts at E, passes through at least one point from each category in R and ends at E. As an example of a use case, consider that somebody plans to travel from Marousi to Elefsina and wants to stop at a Hotel, and a Restaurant (two fixed order middle destinations). The implemented application with a database that stores the distance matrix of the above objects computes efficiently a feasible route that minimizes the total traveling distance (or time) (fig.1, fig.2).

Although the static data are always available the application needs to have internet access when a user gives for the first time a source and a destination point in order to call Google GeoCoding service and initialize these two nodes. Also the application calls Google Distance Matrix to store distances between nodes of the graph. The graph is not explicitly built and this is on more advantage of the implemented algorithm, because the application does not calculate all the inner layer distances. When the graph is constructed the final step is to call the Google Direction Api to map the optimal route. The response of the above Google services is generally a JSON string that is parsed and serialized to JSON objects.

The application uses some techniques and conventions to make the recommended solution feasible. Such a convention is the locality of the data sets that best fit to thesis problem. Nevertheless the data form OpenStreetMaps are not so enriched for Attica although they were chosen without compromising the guaranteed optimality of the experimental results. Also an embedded SQLite database was used to cache the already requested distances for future access. This makes the application to be extended dynamically and continuously every time is used.

The following chapter presents a summary of the studied papers on the TPQ and other SRQ relative problems along with some proposed algorithms and their limitations and complexity issues. Results are depending on the number of Poi categories and categories' cardinality. This literature review would help to understand the notion of queries in Geographic Information Systems (GIS). The chapter 3 presents a UML class diagram with the basic modules that take part in the application along with a summary of the programming techniques that were used. Chapter 4 is a case study of the developed application and finally in chapter 5 are noted the conclusions of the master thesis work.



22:59

MyMapApp

This an initial demo of TPQ algoritm

marousi

elefsina

Banks

✓ Hotels

✓ Restaurants

Trace Route

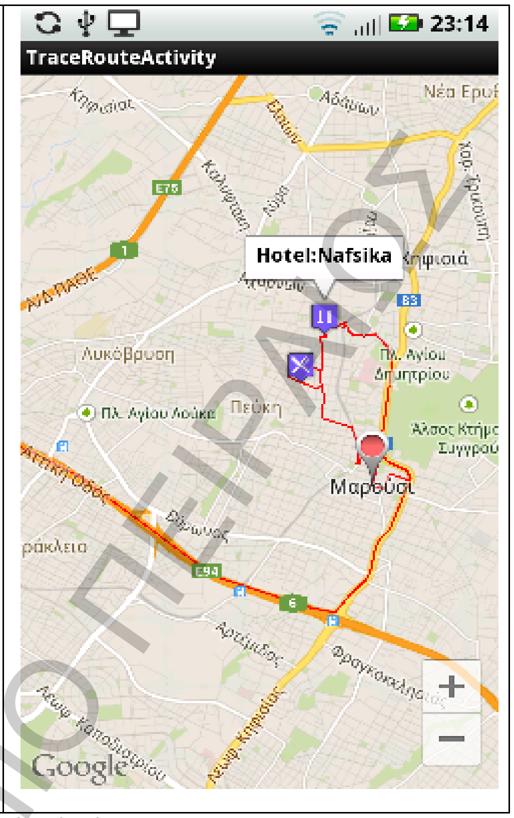


Fig.1: user inputs and POI sets **Fig.2:** the computed optimal route

2. Literature Review

This chapter tries to recursively describe the problem, starting with an introduction of the general TPQ problem and some approximate solutions and ending up with the proposed OSR solution on mobile devices. The first section based on paper [1] proposes a novel type of query in spatial databases and studies methods for answering this query efficiently. Approximation algorithms that achieve various approximation ratios are presented, based on two important parameters: The total number of categories m and the maximum category cardinality ρ. The second section based on paper [2] presents a similar problem introducing the Optimal Sequenced Route Query and also provides two different solution aspects. Finally the last section based on paper[3] quotes two speed up solutions over the OSR query. The mobile application was developed to efficiently evaluate the *Iterative Doubling* solution that delegates the OSR and is described in details in section 2.3.3.

2 .1. TPQ Fast Approximation Algorithms

This section introduces four algorithms for answering TPQ queries, with various approximation ratios in terms of m and ρ. It gives two practical, easy to implement solutions better suited for external memory datasets, and two more theoretical in nature algorithms that give tighter answers, better suited for main memory evaluation. These algorithms are tested above for a practical scenario that considers an application in road network.

This subsection defines formally the generated TPQ problem and introduces the basic notation that will be used in the rest of the section.[1]

2.1.1 Problem Formulation

The solutions for the TPQ problem are considered on metric graphs. Given a connected graph G(V,E) with n vertices V={v_1,…,v_n} and s edges E ={e_1,…,e_n}, we denote the cost of traversing a path v_1,…,v_n with c(v_1,…,v_n)≥0. The metric graph in this case, satisfies the triangle inequality condition and there are not cyclic references between its nodes. Given a set of m categories C={$C_{1,\dots,}C_m$} (where m ≤ n) and a mapping function π : $v_i \rightarrow C_j$ that maps each vertex $v_i \in V$ to a category $Cj \in C$, the TPQ problem can been defined as follows: Given a set R $\subseteq$ C (R ={ $R_{1,}$ $R_{2, \dots}$,R_k}), a starting vertex S and an ending vertex E, identify the vertex traversal T = {S, $\upsilon_{t1, \dots ,}$ $\upsilon_{tk,}$ E} (also called a trip) from S to E that visits at least one vertex from each category in R (i.e. $\bigcup_{i=1}^{k} \pi(\upsilon_{ti}) = R$) and has the minimum possible cost c(T) (i.e., for any other feasible trip T¨ satisfying the condition above, c(T) ≤ c(T¨)).

The total number of vertices is denoted by n, the total number of categories by m, and the maximum cardinality of any category by ρ. For ease of exposition, it will be assumed that R = C thus k = m . Generalizations for R ⊂ C are straightforward (as will be discussed shortly).

2.1.2 Fast approximation algorithms

The section examines several approximation algorithms for answering the trip planning query in main memory. For each solution are provided the approximation ratios in terms of m and ρ. These solutions and the approximation boundaries denote the complexity of the TPQ problem although the studied paper suggests the fast main memory space for its results. They follow two greedy algorithms with tight approximation ratios with respect to m.

2.1.2.1 Approximation in Terms of m

Nearest Neighbor Algorithm: The most intuitive algorithm for solving TPQ is to form a trip by iteratively visiting the nearest neighbor of the last vertex added to the trip from all vertices in the categories that have not been visited yet, starting from S.[1]

```
Algorithm 1 $A_{NN}$ ($G^{C,}$R,S,E)
1.v = S, {1, … , m}, $T_a$ = {S}
2.for k =1 to m do
3. v = the nearest NN(v, $R_i$) for all i ∈ I
4. $T_a$ ← {v}
5. I ← I – {i}
6.end for
7.$T_a$ ← {E}
```

Table 1: nearest neighbor algorithm

For simplicity, consider that a given complete graph G^C, contains one edge per vertex pair representing the cost of the shortest path in the original graph G.

Minimum Distance Algorithm: This section introduces a novel greedy algorithm, called A_{MD}, that achieves a much better approximation bound, in comparison with the previous algorithm.

```
Algorithm 2 $A_{MD}$ ($G^{C,}$R,S,E)
1.U = ∅
2.for k =1 to m do
3.    U ← π(v) = $R_i$ : c(S,v) + c(v,E) is minimized
4.v = S, $T_a$ ← {S}
5. while U ≠ ∅ do
6.  v = NN(v,U)
7. $T_a$ ← {v}
8. Remove v from U
9. end while
10. $T_a$ ← {E}
```

Table 2: minimum distance algorithm

The above NN algorithm gives a $2^{m+1} - 1$ approximation (with respect to the optimal solution) and the MD give an m+1 or m approximate solution. *Both solutions have a tight approximation boundary that do not give the guaranteed optimal solution we are looking for.*

2.1.2.2 Approximation in Terms of ρ

This section introduces an Integer Linear Programming approach for the TPQ problem which achieves a linear approximation bound ρ (the maximum category cardinality). We can consider an alternative formulation of the TPQ problem with the constraint that S=E and denote this problem as a Loop Trip Planning Query(LTPQ) problem.

Let A = (a_{ji}) be the m x (n+1) incidence matrix of G, where rows correspond to the m categories, and columns represent the n+1 vertices (including v_0 = S = E). A's elements are arranged such that a_{ji} = 1 if π(v_i) = R_j , a_{ji} = 0 otherwise. Clearly, ρ= $\max_j \sum_i a_{ji}$. Each category contains at most ρ distinct vertices. Let indicator variable y(v) = 1. If vertex v is in a given trip and 0 otherwise. Similarly, let x(e) = 1 if the edge e is in a given trip and 0 otherwise For any S ⊂ V, let δ(S) be the edges contained in the cut (S, V \ S).[1]

In order to get a feasible solution for LP_{LTPQ}, we apply the randomized rounding scheme. *Randomized Rounding:* For solutions obtained by $LP_{LTPQ.}$, set y(v) = 1_ if y(v_i) $\geq \frac{1}{\rho}$ If the trip visits vertices from the same category more than once, randomly select one to keep in the trip and set y(v_j) = 0 for the rest. [1]

The Thesis approach needs an optimal route guarantee and also it does not care about the maximum category cardinality rather than the locality of the requested data set.

2.1.2.3 Approximation in Terms of m and ρ

The Generalized Minimum Spanning Tree (GMST) problem, is closely related to the TPQ problem. Also the TSP problem is closely related to the Minimum Spanning Tree (MST) problem, where a 2-approximation algorithm can be obtained for TSP based on MST. In similar fashion, it is expected that one can obtain an approximate algorithm for TPQ problem, based on an approximation algorithm for GMST problem. [1,16,20]

Unlike the MST problem which is in P, GMST problem is in NP. Suppose we are given an approximation algorithm for GMST problem, denoted A_{GMST} . The paper constructs an approximation algorithm for TPQ problem as shown in Algorithm 3.[1]

Algorithm 3 APPROXIMATION ALGORITHM FOR TPQ BASED ON GMST

```
1.Compute a β-approximation $Tree_a^{GMST}$ for G rooted at S using $A_{GMST}$
2.Let LT be the list of vertices visited in a pre-order tree walk of $Tree_a^{GMST}$
3.Move E to the end of LT
4.Return $T_a^{TPQ}$ as the ordered list of vertices in LT
```

Table 3: approximation algorithm for TPQ based on GMST

2.1.3 Applications in Road Networks.

An interesting application of TPQs is on road network databases. Given a graph N representing a road network and a separate set P representing points of interest (gas

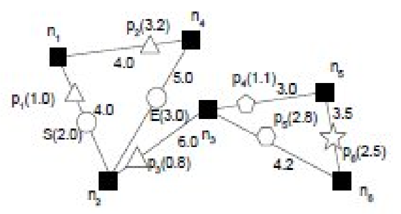


Fig. 3. A simple road network

stations, hotels, restaurants, etc.) located at fixed coordinates on the edges of the graph, we would like to develop appropriate index structures in order to answer efficiently trip planning queries for visiting points of interest in P using the underlying network N. Figure 3 shows an example road network, along with various points of interest belonging to four different categories.[1]

The road network presentation uses techniques from [4, 5, 6]. In summary, the adjacency list of N and set P are stored as two separate flat files indexed by B^+-trees. For that purpose, the location of any point $p \in P$ is represented as an offset from the road network node with the smallest identifier that is incident on the edge containing p. For example, point p_4 is 1.1 units away from node $n_{3.}$

Implementation of A_{NN} Nearest neighbor queries on road networks have been studied in [6], where a simple extension of the well known Dijkstra algorithm [7] for the single-source shortest-path problem on weighted graphs is utilized to locate the nearest point of interest to a given query point. As with the R-tree case, straightforwardly, the algorithm of [6] can be utilized to incrementally locate the nearest neighbor of the last stop added to the trip, that belongs to a category that has not been visited yet. The algorithm starts from point S and when at least one stop from each category has been added to the trip, the shortest path from the last discovered stop to E is computed. *In practice this demands the explicit construction of a layered graph and finally it is prohibitive in terms of running time and space consumption. The thesis application modifies the above approach according to the Iterative Doubling such that no explicit construction of the layered graph is needed.*

Implementation of $A_{MD.}$ The algorithm locates a point of interest p: $\pi(p) \in R_i$ (given R_i) such that the distance c(S, p, E).is minimized. The search begins from S and incrementally expands all possible paths from S to E through all points p. Whenever such a path is computed and all other partial trips have cost smaller than the tentative best cost, the search stops. The key idea of the algorithm is to separate partial trips into two categories: one that contains only paths that have not discovered a point of interest yet, and one that contains paths that have. Paths in the first category compete to find the shortest possible route from S to any p Paths in the second category compete to find the shortest path from their respective p to E. The overall best path is the one that minimizes the sum of both costs.[1]

Algorithm 4 ALGORITHM MD Query FOR ROAD NETWORKS

Require: Graph N, Points of interest P, Points S,E, Category R_i

```
1:For each $n_i \in$ N: ni.cp = $n_i.c_{-p}$ = ∞
2: PriorityQueue PQ = {S}, B = ∞, $T_B$ = ∅
3: while PQ not empty do
4:     T= PQ.top
5:     if T.c ≥ B then return $T_B$
6:     for each node n adjacent to T.last do
7:         T' = T
8:         if T' does not contain a p then
9:           if $\exists p : p \in P, \pi(p) = R_i$ on edge (T'.last, n) then
10:             T'.c +=c(T'.last,p)
11:             T' ← p , PQ ← T'
12:           else
13:             T'.c += c(T'.last,n), T' ← n
14:             if $n_i.c_{-p}$ > T'.c then
15:                 $n_i.c_{-p}$ = T'c, PQ ← T¨
16:         else
17:             if edge (T', n) contains E then
18:                    T'.c += c(T'.last, E), T' ← E
19:                    Update B and $T_B$ accordingly
20:             else
21:                    T'.c += c(T'.last, n), T' ← n
22:             if n.$c_p$ > T'.c then
23:                    n.$c_p$  = T'c, PQ ← T¨
24:       endif
25:    endfor
26: endwhile
```

Table 4:algorithm MD query for road networks

The algorithm proceeds greedily by expanding at every step the route with the smallest current cost. Furthermore, in order to be able to prune trips that are not promising the algorithm maintains two partial best costs per node n $\in N$. Cost n.c_p (n$_i$.c_{-p}) represents the partial cost of the best trip that passes through this node and that has (has not) discovered an interesting point yet. After all points(one from each category $R_i \in R$) have been discovered by iteratively calling this algorithm, an approximate trip for TPQ can be produced. It is also possible to design an incremental algorithm that discovers all points from categories in R concurrently.[1]

The developed Thesis application takes advantage of the already constructed Google graphs and the location of POIs along with their distances, and finds the shortest path with the minimum cost without the need to maintain the intermediate non-important point/nodes of the underline network. These intermediate points-conjunctions are a black box to the thesis algorithm, reducing this way the computational cost and the DB access. So the above algorithm does not consist a feasible solution on a mobile device.

2.2.The Optimal Sequenced Route Query

An unexplored form of NN queries named optimal sequenced route (OSR) query primarily in metric spaces such as road networks are my thesis main concern. The OSR strives to find a route of minimum length starting from a given source location and passing through a number of *typed* locations in a particular order imposed by the user. The PNE algorithm that is proposed in this study, progressively issues NN queries on different point types to construct the optimal route for the OSR query. The proposed algorithm is quite close to the one that finally was implemented in the android application that thesis presents.

2.2.1 Introduction – Motivation

Suppose we are planning a Saturday trip around the town as follows: first we intend to visit a shopping center in the afternoon to check the season's new arrivals, then we plan to dine in an restaurant in early evening, and finally, we would like to watch a specific movie at late night. Naturally, we intend to drive the minimum overall distance to these destinations. That is, we need to find the locations of the shopping center g_i, the restaurant lj, and the theater p_k that shows our movie, where traveling between these locations in the given order would result in the shortest travel distance (or time). Note that in this example, a time constraint enforces the order in which these destinations should be visited; we usually do not have dinner in the afternoon, or go for shopping at late night.

This type of queries where the order of points to be visited is given and fixed, are known as the optimal sequenced route queries or *OSR* for short. Figure 4, shows that the OSR query cannot be optimally answered by simply performing a series of independent nearest neighbor searches from different locations. The figure 4 shows a network of equally sized connected square cells, three different types of *point sets* shown by white, black and gray circles representing shopping centers, restaurants, and theaters, respectively, and a starting point *s* (shown *by* △).

A greedy approach to solve OSR is to first locate the closest shopping center to *p*, *g*2, then find the closest restaurant to *g*2, *l*2, and finally find the closest theater to *l*2, *p*2.Assuming the length of each edge of a cell is 1 unit, the total length of the route found by this greedy approach, (s, g2, l2, p2), shown by dotted lines in the figure, is 15 units. However, the route (s, g1, l1, p1) (shown with solid lines in the figure) with the length of 12 units is the optimum answer to our query. Note that g1 is not the closest shopping center to p and l1 is actually the farthest restaurant to g1. Hence, the optimum route for an OSR query can be significantly different from the one found by the greedy approach.[2]

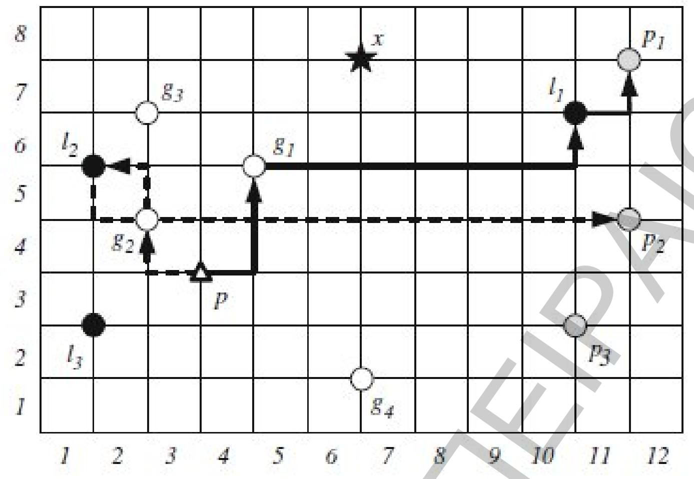


Fig. 4. A network with tree different types of point sets

2.2.2 Problem definition

The terms and notations that appears throughout this section, formally define the OSR query, and discuss the unique properties of OSR that are utilized in the suggested solutions. In [1] one can find a table that summarizes the set of notations.

Let *U*1,*U*2, . . . ,*Un* be *n* sets, each containing points in a *d*-dimensional space R*d*, and *D(· , ·)* be a distance metric defined in R*d* where *D(· , ·)* obeys the triangular inequality. To illustrate, in the example of *Fig. 2*, *U*1, *U*2, and *U*3 are the sets of black, white, and gray points, representing restaurants, shopping centers and theaters, respectively.[2]

Assume that we are given a sequence M = (M1,M2, . . . , Mm). For a given starting point p in Rd and the sequence M, the OSR query, Q(p, M), is defined as finding a sequenced route R = (P1, . . . ,Pm) that follows M where the value of the following function L is minimum over all the sequenced routes that follow M:

$$L(p, R) = D(p,P1) + L(R)$$

Note that L(p, R) is in fact the length of route Rp =p ⊕R. Q(p, M) = (P1,P2,. . . ,Pm) denotes the optimal SR, the answer to the OSR query Q. Without loss of generality, this optimal route is unique for given *p* and *M*.1 For example in the above figure 2 we can consider *(U*1,*U*2,*U*3*) = (black*, *white*, *gray)*, *M = (*2, 1, 3*)*, and *D* is the Manhattan distance, the answer to the OSR query is *Q(p*, *M) = (g*1, *l*1, *p*1*)*. A *candidate* SR is used to refer to all sequenced routes that follow sequence *M*. [2]

2.2.3 The Dijkstra based solution

This section studies a different naive approach which slightly improves the brute-force approach. We are given an OSR query with a starting point *p*, a sequence *M*, and point sets {*UM*1 , . . . ,*UMm* }. We construct a weighted directed graph *G* where the set *V* $=\cup_{i-1}^{m}$ *UMi*∪{*p*} are the vertices of *G* and its edges are generated as follows. The vertex corresponding to *p* is connected to all the vertices in point set *UM*1.Subsequently, each vertex corresponding to a point *x* in *UMi* is connected to all the vertices corresponding to the points in *UMi*+1 ,where 1 ≤ *i* < *m* − 1. Figure 5 illustrates an example of such graph. As shown in the figure 5, the graph *G* is a *k-bipartite* graph where *k* = *m*+1. The weight assigned to each edge of *G* is the distance between the two points corresponding to its two end-vertices.

This graph is showing in fact all possible candidate sequenced routes (candidate SRs) for the given *M* and the set of *UMi* 's. To be precise, it shows all the routes *Rp* = *p* ⊕ *R* where *R* is a candidate SR. By definition, the optimal route for the given OSR query is the candidate SR, *R*, for which *Rp* has the minimum length. Considering graph *G*, we notice that the OSR problem can be simply considered as finding the shortest paths (i.e., with minimum weight) from *p* to each of the vertices that correspond to the points in *UMm* (i.e., the last level of points in Fig. 5), and then returning the path with the shortest length as the optimal route. This can be achieved by performing the Dijkstra's algorithm on graph G.[2]

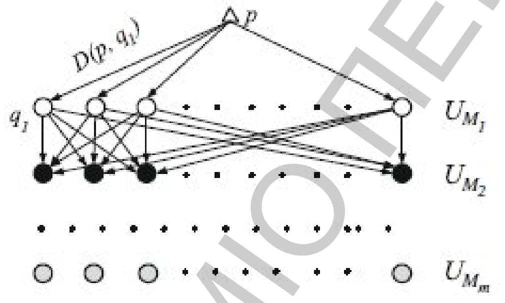


Fig. 5. Weighted directed graph *G* for sequence *M*

There are two drawbacks with this solution. First, the graph *G* has |*E*|=|*UM*1|+ $\sum_{i=1}^{m-1}$ |*UMi*|×|*UMi*+1| directed edges which is a large number considering the usually large cardinality of the sets *Ui*. For instance, for a real world dataset with 40,000 points and |*M*| = 3, a graph *G* has 124 million edges. The time complexity of the Dijkstra's classic algorithm to find the shortest path between two nodes in graph *G* is *O(*|*E*| log |*V*|*)*. Hence, the complexity of this naive algorithm is *O(*|*UMm*||*E*| log |*V*|*)*. Second, this huge graph must be built and kept in main memory. Although there exist versions of the Dijkstra's algorithm that are adjusted to use external memory [8], but they result in so much of overhead which makes them hard to employ for OSR queries. This renders the classic Dijkstra's algorithm to answer OSR queries in real-time impractical. *In order to improve the performance of this naïve Dijkstra-based solution, we can issue a range query around the starting point p and only select the points that are closer to p than L(p, Rg(p,M))*[2]. This is because the length of any route R which includes a point outside this range is greater than that of the greedy route Rg(p, M). Therefore, we build the graph G using only the points within the range instead of all the points. This is the enhanced version of the Dijkstra's algorithm (EDJ).

2.2.4 OSR Solution in metric spaces

Proposed solutions for OSR queries such as EDJ or LORD, although they are efficient in vector spaces, *are impractical or inefficient for a sequence M in a metric space (road networks).* Even though both EDJ and LORD can be applied to both vector and metric spaces, their extensive usage of the *D(・, ・)* function renders them inefficient for metric spaces where the distance metric is usually a computationally complex function.

When applied on road networks, both EDJ and LORD require significant number of distance computations, each of them corresponds to finding a shortest path in the road network. This makes EDJ and LORD infeasible for road networks. Likewise, R-LORD can only be applied to vector spaces since it is based on utilizing R-tree index structure.[2]

In my android application the road network (or general spatial networks) as also happens here are modeled as weighted graphs where the intersections are represented by nodes of the graph and roads are represented by the edges connecting the nodes. The weights can be the distances of the nodes or they can be the time it takes to travel between the nodes (representing shortest times). The distance between any two points on the nodes or edges of the graph is the length of the shortest path connecting them via the graph edges.

Although the triangle inequality is the only requirement of the model as proposed to the current algorithm, there is the assumption that the graph model of the road network is *undirected* so the distance function is symmetric.

PNE Algorithm

```
Function NN(point p, dataset Ui)
returns the closest point to p in Ui

Function NextNN(point p, point n, dataset Ui)
returns q ≠ n, the next closest point to p in Ui. s.t.
D(q,p) ≥ D(n,p);

Algorithm PNE(point p, sequence M)
01. MinHeap H = {};
02. q = NN(p, $U_{M1}$);
03.add ((q)),D(p,q)) to H;
04.do {
05.   remove route PSR from H; //shortest route
06. k = |PSR|;
07. if(k = m) then
08.   return PSR;
09. else
10.   $P_{k+1}$ = NN($P_{k+1,}$ $U_{Mk+1}$);
11.   PSR´ = ($P_1$,…,$P_{k,}$ $P_{k+1}$);
12.   add (PSR´, L(p, PSR´)) to H;
13.   if(k > 1) then
14.         $P_k$´ = NextNN($P_{k-1,}$ $P_k$, $U_{Mk}$);
```

```
15.         PSR´ = (P_1,…,P_{k-1},P_k);
16.  else
17.         P_k´=NextNN(p, P_1, U_{M1});
18.         PSR´=(P_k´);
19.  add(PSR´,L(p, PSR´)) to H;
20.} while |PSR| < m;
```

Table. 5: Pseudo-code of the PNE algorithm for a metric space

Progressive neighbor exploration (PNE), for OSR queries in metric spaces for arbitrary values of *M*, uses efficient fast nearest neighbor algorithms such as INE [10] or VN3 [9] utilized for road network databases to replace the extensive use of distance computation operations in LORD. It utilizes the progressiveness of these algorithms to efficiently build candidate sequenced routes and refine them. Similar to EDJ (and LORD), PNE addresses OSR in both vector and metric spaces. However, it is suitable for the spaces where the computation of the distance metric is very expensive. Notice that PNE uses the same road network model specified by its underlying nearest neighbor algorithm.[2]

Table 5 shows the pseudo-code of the PNE algorithm. The idea behind PNE is to incrementally create the set of candidate routes for *Q(p,M)* in the same sequence as *M*, i.e., from *p* toward *UMm*. This is achieved through an iterative process in which we start by examining the nearest neighbor to *p* in *UM*1 , generating partial SR from *p* to this neighbor, and storing the candidate route in a heap based on its length. At each subsequent iteration of PNE, a partial SR *(e.g., PSR = (P*1 *,P*2 , . . . *,P|PSR|))* from top of the heap is fetched and examined as follows[2]:

1. If |*PSR*| = *m*, meaning that the number of nodes in the partial SR is equal to the number of items in *M* and hence *PSR* is a candidate SR that follows *M*, the *PSR* is selected as the optimal route for *Q(p*, *M)* since it also has the shortest length.
2. If |*PSR*| < *m*:
   (a) First the last point in *PSR*, $P_{|PSR|,}$ (which belongs to $U_{M|PSR|}$) is extracted and its next nearest neighbor in $U_{M|PSR|+1}$, $P_{|PSR|+1}$, is found. This will guarantee that (a) the sequence of the points in *PSR* always follows sequence specified in *M*, and (b) the points that are closer to $P_{|PSR|}$ and hence may potentially generate smaller routes are examined first. The fetched *PSR* is then updated to include $P_{|PSR|+1}$ and is put back in to the heap.
   (b) We then find the next nearest neighbor in $U_{M|PSR|}$ to $P_{|PSR|-1}$, $P'_{|PSR|}$, generate a new partial SR, $PSR' = (P1 ,P2 ,…,P_{|PSR|-1} ,P'_{|PSR|})$, and place the new route in to the heap. This is because once the point $P_{|PSR|}$, which we can assume is the *k*-th nearest point in $U_{M|PSR|}$ to $P_{|PSR|-1}$, is chosen in step (a) above, the *(k* + 1*)*-st nearest point in $U_{M|PSR|}$ to $P|_{PSR|-1}$ (e.g., $P'_{|PSR|}$) is the only next point that may generate a shorter route and hence, must be examined. If |*PSR*| = 1, we find the next nearest point in U_{M1} to *p*.

Recall that the OSR query was to drive toward a shopping center, a restaurant, and then a theater (i.e., $M = (2, 1, 3)$ and $|M| = m = 3$). Table 6 depicts the values stored in the heap in each step of the algorithm. In step 1, the first nearest *si* to *p*, *s*2, is found and the first partial SR along with its distance, *(s*2 : 2*)*, is generated and placed in to the heap. In step 2, first *(s*2 : 2*)* is fetched from the heap. Since for this route $|PSR| < 3$, the above steps 2(a) and 2(b) are performed. More specifically, first the next nearest *ri* to *s*2, *r*2, is found; the partial SR is updated by adding *r*2 to it; and is placed back into the heap. Second, the next nearest *si* to *p*, *s*1, is found and is placed in to the heap. Similarly, this process is repeated until the route on top of the heap follows the sequence *M*(i.e., *(s*1, *r*1, *t*1*)* in step 13). Note that we only keep one candidate SR(i.e., route with *m p*oints) in the heap. That is, if during step 2(a) a route with *m* points is generated, it is only added to the heap if there is no other candidate SR with a shorter length in the heap. Moreover, after a candidate SR is added to the heap, any other SR with longer length will be discarded. For example, in step 6, adding the route *(s*2, *r*3, *t*3*)* with the length of 14 to the heap will result in discarding the route *(s*2, *r*2, *t*2*)* with the length of 15 from the heap (crossed out in the figure). However, by keeping *k* routes in the heap and continuing the algorithm until *k* routes are fetched from the heap, we can easily address a variation of OSR where *k* routes with the minimum total distances are requested.[2]

The only requirement for PNE is a nearest neighbor approach that can progressively generate the neighbors (i.e., a distance browsing algorithm [11]). Hence, by employing an approach similar to INE [10] or our VN3 [9], which are explicitly designed for metric spaces, PNE can address OSR queries in metric spaces. In theory, PNE can work for vector spaces in a similar way; however, it is inefficient for these spaces where distance computation is not expensive. The reason is that PNE explores the candidate routes from the starting point which may result in an exhaustive search. Instead, R-LORD optimizes this search by building the routes in the reverse sequence utilizing the R-tree index structure.[2]

Step	Heap contents (partial candidate route R : *L(p,R)*)
1.	*(s2 : 2)*
2.	*(s1 : 3), (s2, r2 : 4)*
3.	*(s2, r2 : 4), (s3 : 4), (s1, r2 : 6)*
4.	*(s3 : 4), (s2, r3 : 5), (s1, r2 : 6), (s2, r2, t2 : 15)*
5.	*(s2, r3 : 5), (s4 : 5), (s1, r2 : 6), (s3, r2 : 6) (s2, r2, t2 : 15)*
6.	*(s4 : 5), (s1, r2 : 6), (s3, r2 : 6), (s2, r1 : 12) (s2, r3, t3 : 14), ~~(s2, r2, t2 : 15)~~*
7.	*(s1, r2 : 6), (s3, r2 : 6), (s4, r3 : 11), (s2, r1 : 12) (s2, r3, t3 : 14)*
8.	*(s3, r2 : 6), (s1, r3 : 9), (s4, r3 : 11), (s2, r1 : 12) (s2, r3, t3 : 14), ~~(s1, r2, t2 : 17)~~*
9.	*(s1, r3 : 9), (s3, r3 : 9), (s4, r3 : 11), (s2, r1 : 12) (s2, r3, t3 : 14),~~(s3,r2,t2:17)~~*
10.	*(s3, r3 : 9), (s1, r1 : 10), (s4, r3 : 11), (s2, r1 : 12) (s2, r3, t3 : 14), ~~(s1, r3, t3 : 18)~~*
11.	*(s1, r1 : 10), (s4, r3 : 11), (s2, r1 : 12), (s3, r1 : 12) (s2, r3, t3 : 14), ~~(s3, r3, t3 : 18)~~*
12.	*(s4, r3 : 11), (s2, r1 : 12), (s3, r1 : 12), (s1, r1, t1 : 12) ~~(s2, r3, t3 : 14)~~*
13.	*(s2, r1 : 12), (s3, r1 : 12), (s1, r1, t1 : 12) ~~(s4, r3, t3 : 20)~~*

Table 6. PNE for the example of **Table 5**

2.3. Speeding up SR Queries

The proposed solution of this study is based on the combination of a distance sensitive doubling technique and contraction hierarchies and is in orders of magnitudes faster than either a naive approach of previous results and produces the answers in an instant for realistic queries without compromising guaranteed optimality. This type of route query becomes feasible even on mobile devices .[3]

This last proposed algorithms and precisely the "Iterative Doubling" was chosen for the purpose of the academic Thesis in order to implement an android application that computes the optimal route from a source to a destination, through a set of Pois that belong to separate sequenced categories.

2.3.1 Introduction

Given a graph G(V,E) with edge costs w ⊂ ℝ and a collection C = {C1,C2, . . . ,Ck} of facilities with Ci belongs to V . For example, G could be the road network of Athens, w the travel times on the road segments, C1 the locations of all gas stations in the network, C2 the locations of all ATMs, etc.

A query is specified by a source s and a target t as well as a sequence of facility classes (p1, p2, . . . , pl). We are interested in finding the shortest path from s to t in G visiting a facility in C_{p1} followed by a facility in C_{p2} . . ., followed by a facility in C_{p3}. This type of query is referred to as already mentioned in the previous chapters sequenced route query . Answering such a query allows us to find for example the fastest route home from work visiting an ATM, a gas station and a post-office. The order in which the facilities have to be visited is fixed. Dropping the restriction on the order essentially turns this problem (for non constant l) into the NP-hard travelling salesperson problem (TSP). On the other hand, in most practical scenarios, l is rather small, and as our query procedure for fixed order turns out to be very efficient, a brute force exploration of all possible orders is actually possible.[3]

This study proposes two speed-up techniques for answering sequenced route queries. The first is based on a general preprocessing technique for ordinary shortest path queries called contraction hierarchy [12] which can be extended to deal with sequenced route queries. The second technique that the implemented android application adapts, makes use of the fact that likely most sequenced route queries are more of a local kind (doing things on the way back home from work rather than on a cross-country trip), and results in a certain distance sensitivity. The algorithms – in contrast to [11] always compute the optimum solution and do so faster by orders of magnitudes being able to deal with network sizes that could not be processed before. This paper claims the fast query times for sequenced route queries also give answers to queries without fixed order as long as the number of facilities to be visited remains moderate (as seems to be the case in many real-world scenarios).[3]

2.3.2 Summing up of the previous related work.

Sequenced route queries have appeared in several contexts in this essay. In section 2.2.3, the authors consider sequenced route queries in Euclidean space and describe an approach called the EDJ algorithm which creates for a sequenced route query (s, t, p1, . . . , pl) a directed, acyclic layered graph consisting of l+2 layers 0, 1, . . . , l+1. Layer 0 and l+1 consist only of the source and the target respectively. The nodes of layer i correspond to all facilities of type pi. Between layers i and i+1, we have a complete bipartite graph, where the (directed) edge from node v(i) in layer i to node w(i+1) in layer 1 has cost of and corresponds to the shortest path from v to w in G (in [3] this is simply the Euclidean distance). In Figure 6 we see such a layered graph for a query (s, t, p1, p2, p3) (the nodes in layer 1 could correspond to locations of ATMs, layer 2 nodes to gas stations, and layer 3 nodes to grocery stores). Once this layered graph has been constructed, running Dijkstra from s or even simpler, relaxing the edges from top to bottom yields the desired optimal route. In practice however, the construction of such a layered graph is prohibitive, both in terms of running time as well as of space consumption. Remember that we are dealing with thousands of facilities in one single class. So in [2] the authors propose a new algorithm – LORD – which avoids the explicit construction of the complete layered graph by an adaptive threshold technique.

LORD is refined to R-LORD using a range query data structure for nearest neighbor queries to more efficiently prune the search space. The case where the underlying space is not the Euclidean space but a road network is discussed in section 2.2.4 but no experimental results are reported in the paper – probably because computing the (now shortest path) distances between nodes of consecutive layers is very costly, even though the pruning by (R-)LORD reduces the number of such costly computations.[3]

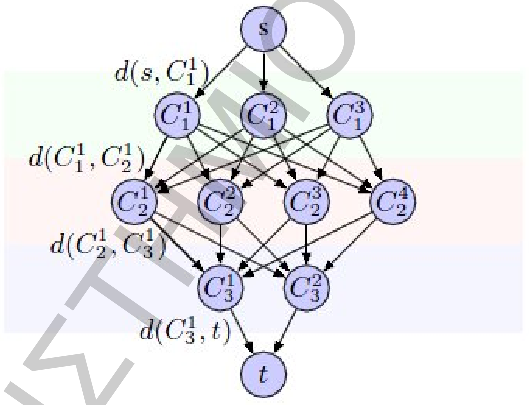


Fig. 6. EDJ layer approach: The enhanced Dijkstra based approach builds an explicit layered graph to cover all inter layer distances. Each edge (Cli ,Cki+1) represents the shortest path from Cli to Cki+1 in the underlying road network and is weighted with d(Cli ,Cki+1). A single Dijkstra computation from s recovers the optimal s − t route.

These following sections introduce two main tools for speeding-up sequenced route queries. While both techniques can be employed independently, the combination of both yields the best speed-up compared to *the naive EDJ approach*. The first speed-up technique – iterative doubling – works well, if the actual result path is relatively short – probably the most frequent type of query result in practice –, avoiding the exploration of facilities that are far away from source and target. [3]

The second technique – contraction hierarchies (CH) – has been developed in the context of fast point-to-point shortest path queries [12]. The proposed algorithm in section 2.3.4, extends CH in a natural way to speed-up the computation of inter-layer distances. This technique applies equally well for local and non-local queries. Both speed-up techniques do not compromise optimality of the result.

2.3.3 Iterative Doubling

The algorithm modifies the EDJ algorithm as follows:

- Run Dijkstra from s to compute distances d0 to all nodes C_1
- Run a single Dijkstra starting at all nodes in C_1 where each node $v \subset C_1$ has initial distance value $d_0(v)$ until all nodes in C_2 are settled. This computes shortest path distances d_1 from s via at least one node in C_1.
- Run a single Dijkstra starting at all nodes in C_2 where each node $v \subset C_2$ has initial distance value $d_1(v)$ until all nodes in C_3 are settled. This computes shortest path distances d_2 from s via at least one node in C_1 and one node in C_2

…

- Run a single Dijkstra starting at all nodes in C_L where each node $v \subset C_L$ has initial distance value $d_L\text{-}_1(v)$ until the target is settled. This actually computes the shortest path from via at least one node in C_1, at least one node in C_2, . . . , at least one node in CL to t.

Clearly, the running time of this approach is essentially that of performing L Dijkstra runs on the graph—which is already a considerable improvement to EDJ which essentially required $\sum|\mathrm{Ci}|$ many Dijkstra computations to compute the weights of all inter-layer edges.[3]

Although there is still an obvious source of inefficiency here. Realistic sequenced route queries are expected to be mostly local (typical commuter distances are 40km to 60km at most which translates to 60 to 90 minutes). It seems very inefficient to explore facilities that are hundreds of kilometers (and hours of driving) away.

If we assume now that we know the length (duration) of the optimal path from s to t visiting facilities in the given order; let that length be D. We could stop each (!) Dijkstra computation above once we reach distance D and still guarantee that we find the optimal path since no subpath of the optimal path can have length more than D. Note that in case the optimal path is rather short – let's say it takes 70 to 100 minutes – this will drastically reduce the search space of every single Dijkstra. Unfortunately we do not know the optimal route's exact length D a priori, this is where the iterative doubling part comes into play. We start with some estimation/lower bound D´ for D which can be pretty small (let's say 10 minutes). We use the above sequence of computations except for one important difference: we abort each Dijkstra run once we have settled all nodes at distance at most D´.[3]

Two things can happen: a) the computation does not reach t – so our estimation D´, was too small, we double D´ and repeat. b) the computation does reach t – so we have a valid path from s to visiting facilities on the way in the right order on a path of distance D´´. It is not hard to see that this solution is optimal.

2.3.4 CH enhanced Iterative Layer Search

Contraction hierarchies ([12]) are a preprocessing scheme that allow for the faster answering of shortest path queries in road networks. The key component of the preprocessing phase is the iterative removal/contraction of nodes in order of increasing 'importance' (nodes at dead-ends or degree-two nodes are removed first, important junctions are contracted last) while preserving the shortest path distances between the remaining nodes. This is achieved by adding a so called shortcut (u,w) between any pair of neighbor nodes u,w of v, if the shortest path from u to w is uvw. The shortcut is created with cost equal to the sum of the costs of edges (u, v) and (v,w). Having removed all nodes but one, all constructed shortcuts are added to the original graph and the nodes are labeled 1 . . . n according to the contraction order. The modified graph has the interesting property that for any pair s, t of nodes, there exists a representation of the shortest path from s to t which can be divided into two parts, one part starting at s and only following edges to nodes with larger label followed by a part which only follows edges to nodes with smaller label. This special property of the augmented graph (original edges plus shortcuts) is then exploited in the query phase by a bidirectional Dijkstra starting at source and target simultaneously.[3] This leads to query times for s-t queries which are about 1000 times faster than ordinary Dijkstra due to the drastically reduced search space, see [12].

To extend this idea to our problem of speeding-up inter-layer Dijkstra computations we have to go a bit more into detail. An edge e = (a, b) $\subset$ E is called an upward edge iff a < b, that is, the node ID or label (after adding shortcuts and elabelling) of a is smaller than that of b. A path p = (e_1, e_2, . . . , e_k) with $e_i \subset$ E is called upward path iff all $e_i \subset$ p are upward edges. We define downward edge and downward path accordingly. Using these definitions we define $G'_{S,v'}$ to be the union of all upward paths starting in v and $G'_{t,v'}$ to be the union of all downward paths ending in v. So the crucial property of a shortest s − t path in a CH-enhanced graph is that it has the form (s, . . . , u, . . . , t) where (s, . . . , u) $\subset G'_s$ and $(u, . . . , t)^T \subset G'_{t'}$.[3]

An edge e = (a, b) ⊂ E is called an upward edge iff a < b, that is, the node ID or label (after adding shortcuts and elabelling) of a is smaller than that of b. A path p = (e_1, e_2, . . . , e_k) with $e_i \subset$ E is called upward path iff all $e_i \subset$ p are upward edges. Also define downward edge and downward path accordingly. Using these definitions we define $G'_{S,v'}$ to be the union of all upward paths starting in v and $G'_{t,v'}$ to be the union of all downward paths ending in v. So the crucial property of a shortest s − t path in a CH-enhanced graph is that it has the form (s, . . . , u, . . . , t) where (s, . . . , u) $\subset G'_s$ and $(u, . . . , t)^T \subset G'_{t'}$.

CH also answers a simple s − t shortest path query by performing two interleaved Dijkstra computations, one starting in s, the other starting in t. The former one only considers edges in G'_s , the latter only edges in $G'_{t'}$.When both Dijkstra computations settle a node v $\subset(G'_s \subset G'_t)$ d(s, v) + d(v, t) is an upper bound for d(s, t) and the shortest path is realized by min $_{(G'_s \subset G'_t)}$ (d(s, v) + d(v, t)). This method can be extended to one to many shortest path computations where the task is to find all shortest paths from a node s ⊂ V to a set of nodes T ⊂ V. The conceptually easiest method is to mark all edges in the downward graph for each t ⊂ T and use and Dijkstra computation from s which considers all edges in (G'_s and all marked edges.[3]

The same methodology can be even further extended by the following preprocessing step. For each facility/POI class we construct the downward graph for this facility class by taking the union of the downward graphs of all nodes in that facility class. These downward graphs can be represented by a one bit marker for each edge and facility/POI class indicating whether the edge belongs to the respective downward graph. Then, during query processing, the ordinary inter-layer Dijkstra is replaced by a Dijkstra operating on the union of the upward graphs of the settled nodes of the current facility class and the downward graph for the next facility class. This speed-up technique does rely on locality of the queries but exhibits a considerable speed-up in all cases.[3]

2.4 Conclusion

During this literature review I have considered the problem of answering sequenced route queries and developed an efficient algorithm programmatically that could take advantage of the *Iterative Doubling* technique that according to the paper [3] allows the exact and fast computation of realistic queries involving common tasks/points of interest in a local area range. The focus in this literature review has been the case where the order in which the points of interests are to be visited is fixed. The very fast query times for fixed order queries allows for a straightforward treatment of unordered or only partially ordered queries by simply enumerating all possible orderings. Another interesting topic for future research is the transition from fixed edge costs to parameterized ones. This extension seems natural under the assumption that the ”cost“ of an edge could be the required travel time or the battery consumption necessary to cross this road segment.

The proposed solution based on the distance sensitive doubling technique is in orders of magnitudes faster than either a naive approach or previous PNE algorithm and the mobile application would try to produce the answers in an instant for realistic queries without compromising guaranteed optimality developing and solve efficiently the OSR problem. We could expect that fast query times, for such a route query would become feasible on the mobile devices.

3. Implementation of TPQ in Android

The basic concern of the thesis project is the development of an Android application that is able to solve the SQR problem “Getting things done in the way home” by implementing *Iterative Doubling* (section 2.3.3) algorithm using Java (for android) technology and an embedded (sqlite) database. In order for a user that wants to test the application and view the results a friendly User Interface (UI) implemented as part of the MainActivity (thread). Another activity that the main activity triggers to execute the algorithm is the TraceRouteActivity. Both basic threads and some basic assets and techniques have been described in details in the following sections.

3.1 UML Class Diagram

Figure 7 shows the class diagram of the android java application that I developed for my master Thesis. In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. The goal is to show only those attributes and operations that are useful for the particular diagram.

As it is obvious in the UML Diagram (fig. 7) the basic java class where the algorithm is running and all the calculations are taking place is the TraceRouteActivity. More precisely this activity class is responsible to display the GoogleMap (mMap) and to design the computed optimal route. It associates all the modules that are essential for the computations. In few words it is the basic Controller of the application. An AsyncTask class that is executed in a parallel tread is triggered by TraceRouteActivity in order to load the data and compute the optimal route.

Static data that concern the POIs are kept in xml format files in the Asset directory of the mobile device. As soon as the user sets up the application and triggers the Activity the data are serialized to aggregate as (uml in figure 7) three maximum PoiSet class objects, one for each of the three Categories of Interest by instantiating the xml elements to Poi Class objects in main memory for quick access. The association relationship in the UML diagram between Poi and Node Object denotes that I construct a node for each Poi inside the distance range and then added to the BipGraph as it is shown by the aggregation relationship.

Other essential module classes are those classes that extend the XMLParser class (figure 7). DMParser provides the driving distance between two nodes if it is not already saved in DB. The Connection Object is the DAO object that represents a connection and it is handle by the DatabaseHandler Object that is responsible for the CRUD operations in DB.

Finally the RouteTask class returns with the Waypoints Object that optimize the route and then constructs the request to the Google Directions Api for the routing information in JSON format. The GoogleParser serialize this information to instantiate the Google map and display the route. Further information about UML class diagram and the objects that represents refer to [14].

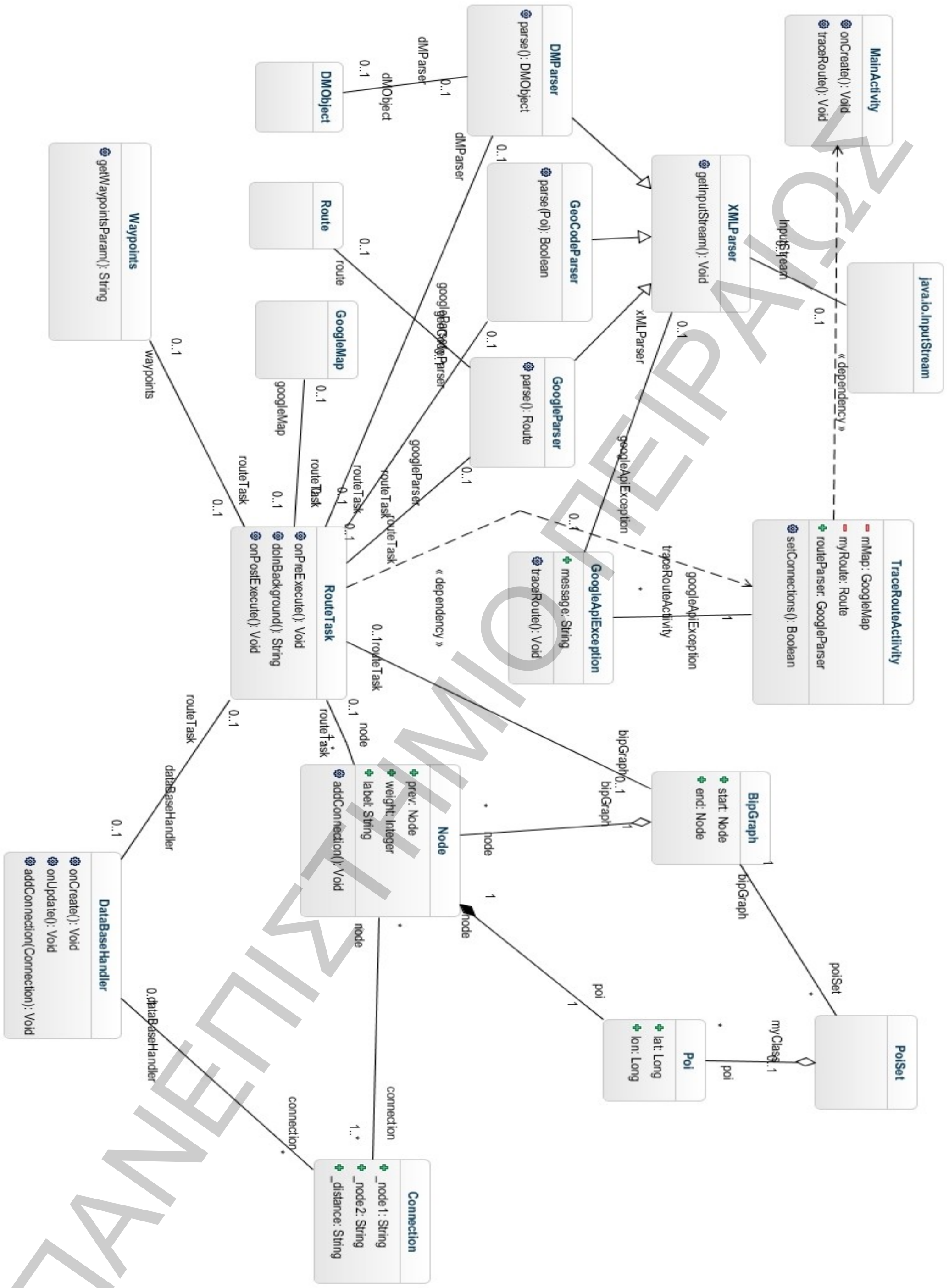


Fig.7: Uml class diagram

3.2 Basic Project Modules

3.2.1 MainActivity

MainActivity is the entry part of the developed Android application and consists of a Linear Layout with two EditText elements where the user inputs the start and end points of the route, a button and three check boxes where the user decides about the Categories of Pois that he wants to pass through while travelling. Finally when the button is clicked, the other Activity TraceRouteActivity is triggered. MainActivity's mainly purpose is to collect and provide the following information to the basic controller module (RouteTask) that processes the algorithm and displays the result:

- MainActivity.*SOURCE (the texted source point)*
- MainActivity.*DESTINATION (the texteddestination point)*
- MainActivity.*BANKS (Boolean isChecked BANKS Poi Category)*
- MainActivity.*RESTAURANTS (Boolean isChecked Poi Category)*
- MainActivity.*HOTELS (Boolean isChecked Poi Category)*

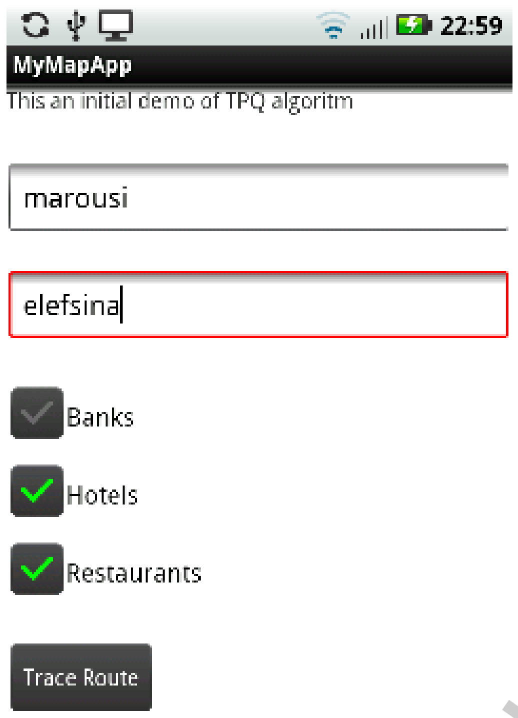
The content layout of MainActivity is shown in the below screenshot (figure 8).Screenshot in figure 9 shows the layout of TraceRouteActivity.

3.2.2 TraceRouteActivity

TraceRouteActivity's contents Layout is the Google map (fig. 8) where the optimal route is depicted after the successful finish of the algorithm. This Activity imports the main modules/classes that are very structural to the application and then calls the RouteTask<AsyncTask> that is an Asynchronous Threat to execute the algorithm without interrupting the main UI thread that displays the map.

TraceRouteActivity holds the most important static variables-objects such as the GoogleMap fragment object, the setConnections function (see uml in figure 7) that is the basic module that takes part during Dijkstra computations where we decide to set the weights of nodes, and finally provides the addPolyline and setUpMap modules in the onPostExecute method of the AsyncTask to draw the optimal route with the intermediate POIs on the map.

Once the these global scope variables are set then the activity calls a new thread that runs asynchronously in parallel, the RouteTask that utilize the “Iterative Doubling” algorithm in order to compute the optimal Path. The actions that manipulate the data and result the POIs that optimize our route, are taking part in the doInBackground(String…) process as it is shown in the uml interaction diagram below (fig. 10).


22:59
MyMapApp
This an initial demo of TPQ algoritm
marousi
elefsina
Banks
✓ Hotels
✓ Restaurants

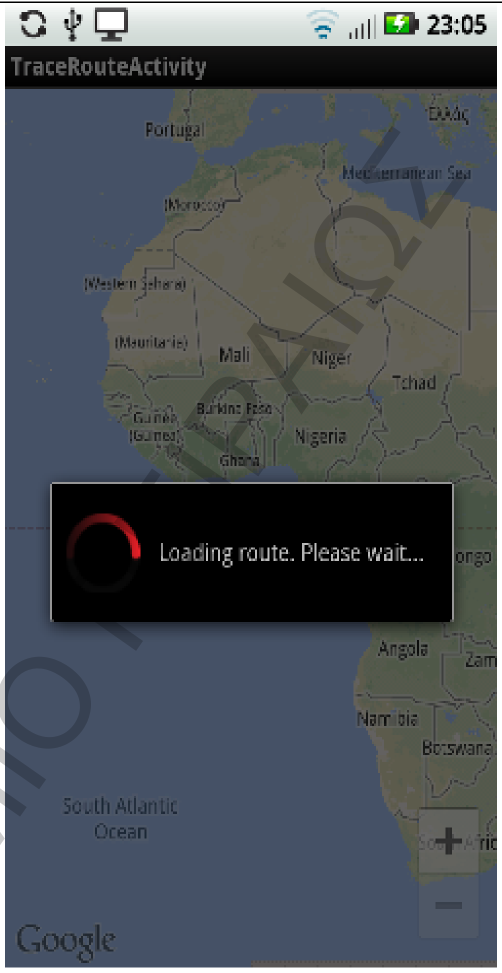

23:05
TraceRouteActivity
Loading route. Please wait...

Fig.8:initial entry screen **Fig.9**: Loading…

The steps before the algorithm, that are part of the preprocessing are the following:

- Call the The Google Geocoding API [15] to get the geographic coordinates of the source and destination points. They are necessary to the next request to the distance matrix api.
- Read and load to the main memory the xml files that include the points of interest of each category according to the user choice
- Initialize the DatabaseHandler object so we can have access to the DB.
- Initialize the graph with the source and destination

After the above steps the application sets all the relative nodes and their connections calling the setConnections method that computes the distances between nodes, saves and constructs the Distance Matrix Table in the DB if it is not already made and computes the Dijkstra's shortest path as the algorithm implies. The key feature here is that the activity holds a static variable called *estimated_distance* with initial value 10000m (10km) that is the initial threshold that the algorithm uses to exclude nodes from Graph that are far away. The user can also choose to initialize himself the *estimated_distance* from the settings of the application. If this threshold fails then it is doubled until the graph has been build and the optimal route has been found.

The SQLite Database is not preinstalled. There is a preprocessing stage where the distances of the edges of the graph are cached while calling the Google Distance Matrix Api [16] that returns the driving distance (or time) between the sequenced candidate optimal point couples. But for the academic purpose and to be closely to the experimental results of chosen algorithm, we assume that the DB has already been built and the required distances are cached. This is essential because of the Google Api limitations in usage by the academic purpose of the application. If a user has already ran a route, then one could easily fasten the time to retrieve the route by choosing the “Use cache” options from the application “*Settings”* menu.

The DB schema that caches the distances of an already visited node for quick access in the next use, consists from only one simple table that has four columns. A primary key, the node (a string label) from where a “directed” edge starts, the node (a string label) where ends and the driving distance between them. The “directed” term it has not a literal meaning, but is used to determine the traversal order of nodes during Dijkstra’s calculations. A combinatorial index between these two nodes has been created.

SQLite is an Open Source database. SQLite supports standard relational database features like SQL syntax, transactions and prepared statements. The database requires limited memory at runtime (approx. 250 KB) which makes it a good candidate from being embedded into application runtime. It supports the data types TEXT (similar to String in Java), INTEGER (similar to long in Java) and REAL (similar to double in Java). All other types must be converted into one of these fields before getting saved in the database.

SQLite is embedded into every Android device. Using an SQLite database in Android does not require a setup procedure or administration of the database. You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.

After the optimal route has been computed the Pois from each Category of interest among with the source and destination points are used as parameters to the final Google Directions Api request. We parse the response and finally in the onPostExecute method the optimal route is drawn in the map (see chapter 4).

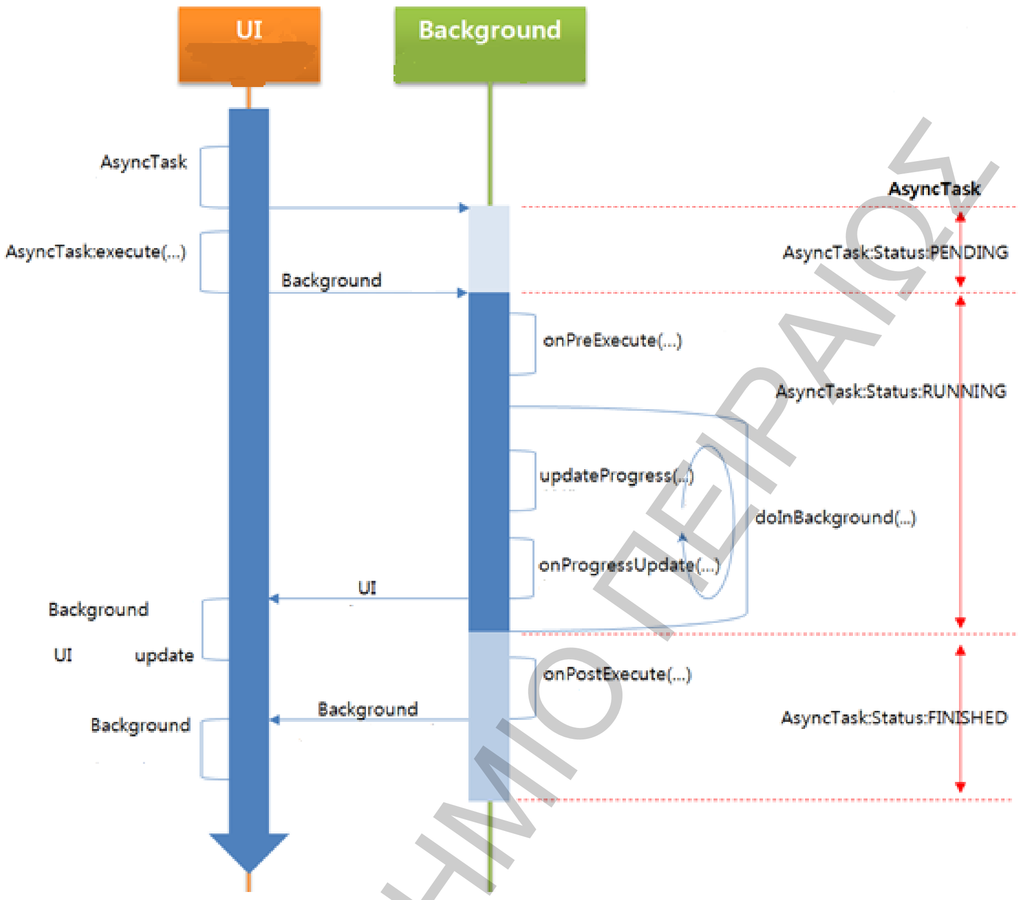


Fig.10: asynk task uml interaction diagram

3.3 Essential modules

The class in Fig. 11 is the model of the layered graph that covers all inner layer distances as it was firstly imposed by the EDJ layer approach. The android application permits maximum tree categories of Pois with particular order imposed by the type of locations. The graph is constructed dynamically and the POIs that are located over the search range of the application algorithm are excluded from the graph.

The Node Class is essential to implement the Dijkstra's algorithm as it models the basic node element and provides proper Setters and Getters methods. Node objects update their weights during the algorithm when a new minimum distance from the source node is validated. They also keep a reference to the node that precedes with the minimum distance. The Object's equal function is overwritten to cover the comparison needs between nodes. The Bipartite Graph Class as it is already shown in the UML is an aggregation of Node class instances.

The POI objects and their geographic coordinates along with other relative metadata are initially stored in xml files each one represents a Poi Category. In order to utilize proper object in main memory and access all these information I took advantage of the Simple library. Simple is a high performance XML serialization and configuration framework for Java. Its goal is to provide an XML framework that enables rapid development of XML configuration and communication systems. This framework aids the development of XML systems with minimal effort and reduced errors. It offers full object serialization and deserialization, maintaining each reference encountered. In essence it is similar to C# XML serialization for the Java platform, but offers additional features for interception and manipulation. [16]

The *Iterative Doubling* algorithmic calculations are implemented during the doInBackground process (figure 10) and finally result to the construction of a Bipartite Graph with all the nodes marked as *visited* according to the Dijkstra run. The optimal path is taken backward starting from the end node and exploring the previous nodes within the shortest path. A pseudo code of the specific process is shown in table 6.

Pseudo Code of 'Iterative Doubling' implementation

```
Function setConnection(node one, node two, threshold){
    If one.weight +getDistance(one,two) > threshold
        If two.weight > one.weight +getDistance(one,two)
            two.updateWeight();
            two.setPrevious(one);
        one.addEdge(two);
        return true
    Else return false;
}
Bipartite graph;
graph.add(source);
Repeat
  Foreach (Node node of first_layer)
      If setConnection(source, node)
          graph.add(node);
  Foreach (Node node of other_layers)
      If setConnection(upperlayer_node, node)
          graph.add(node);
  Foreach(Node node of last_layer)
      If setConnection(node, destination)
          graph.add(destination);
  If graph.getDestination == NULL
    threshold = thresholdx2;
  Until (graph.getDestination !== NULL)
  Return the nodes of optimal route in reverse order starting from Destination node;
```

Table 11. Pseudo code of implemented algorithm

Poi Class and *PoiSet Class* are properly annotated to show the power of the *Simple* library. The application uses the following annotations: *@Root, @Attribute* and *@ElementList* making the XML serialization quite easy process. Each annotation contains a name attribute, which can be given a string providing the name of the XML attribute or element. This ensures that should the object have unusable field or method names they can be overridden, also if your code is obfuscated explicit naming is the only reliable way to serialize and deserialize objects consistently. *@ElementList* annotation supports common relationships. This allows an annotated schema class to be used as an entry to a Java collection object that in my case I use an ArrayList that holds the Poi Object of each PoiSet instance.

The GoogleParser, GeoCodeParser and DMParer classes are responsible to consume JSON strings and to generate the corresponding JSONObjects that are the inputs of the main modules that are already have been mentioned.

3.4 Assets and permissions

The POIs are initially kept in xml format files in the application asset directory. They are serialize to PoiSet objects in order to keep all the information we need to generate the nodes, the graph and the connections/edges between nodes. Due to these purpose the thesis application keeps tree XML file one for each Poi Category (bank.xml, restaurant.xml, hotels.xml) in the Assets directory. Figure 11 shows a small sample of that data that where extracted from OpenStreepMaps.org[13].

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx version="1.0" creator="GPSBabel http://www.gpsbabel.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.topografix.com/GPX/1/0"
xsi:schemaLocation="http://www.topografix.com/GPX/1/0
http://www.topografix.com/GPX/1/0/gpx.xsd">
<time>2011-12-15T14:01:39Z</time>
<bounds minlat="36.686540500" minlon="23.037696400"
maxlat="38.350403700" maxlon="24.054799200"/>
<wpt lat="37.994816500" lon="23.341969300">
  <name>Bank:Marfin Egnatia Bank</name>
  <cmt>Bank:Marfin Egnatia Bank</cmt>
  <desc>Bank:Marfin Egnatia Bank</desc>
</wpt>
<wpt lat="37.994665000" lon="23.342719300">
  <name>Bank:Alpha Bank</name>
  <cmt>Bank:Alpha Bank</cmt>
  <desc>Bank:Alpha Bank</desc>
</wpt>
<wpt lat="37.995012100" lon="23.343975800">
  <name>Bank:Εθνική Τράπεζα</name>
  <cmt>Bank:Εθνική Τράπεζα</cmt>
  <desc>Bank:Εθνική Τράπεζα</desc>
</wpt>
<wpt lat="37.996053300" lon="23.344140000">
  <name>Bank:Αγροτική Τράπεζα</name>
  <cmt>Bank:Αγροτική Τράπεζα</cmt>
  <desc>Bank:Αγροτική Τράπεζα</desc>
</wpt>

…

</gpx>
```

Fig.11: Banks.xml

With all these features that have been mentioned till now and the essence of the previous functional modules it becomes obvious that some extra permissions for accsess external resources such as Google services, and SQLite database tables are essential from the thesis application. The application in order to use the Google Maps has already been registered as in Google APIs Console in debug mode as it is imposed by [18]. Android permissions for Internet Access and Read and Write in external storage also required.

In figure 1 we can see the initial application screen (main layout) where the user has typed the source and destination address. Soon after the user clicks over the “Trace” button the application loads the data and runs the algorithm. At this time the *AsyncTask* (RouteTask) loads an indicator for the user to understand that there is a running process while he waits for the result (figure 2). Figure 12 shows the optimal route with markers that represent the source the destination and the Pois from where the route passes through. Finally figure 21.shows a proper exception message in case something bad happens while we make request to any of the several Google APIs and a “query limitation” or “internet failure” happens.

4. Evaluation

This chapter would try to prove the concurrency and correctness of the algorithm. The main purpose of the application is to answer efficiently the SRQ, adapting the “Iterative Doubling” among other algorithms as a suitable algorithm that can run to a mobile application in a feasible way. In addition, I would try to prove the guaranteed optimality, giving the steps of the algorithm in a graphical way utilizing a small set of Pois for the proof of content.

4.1 Case Study

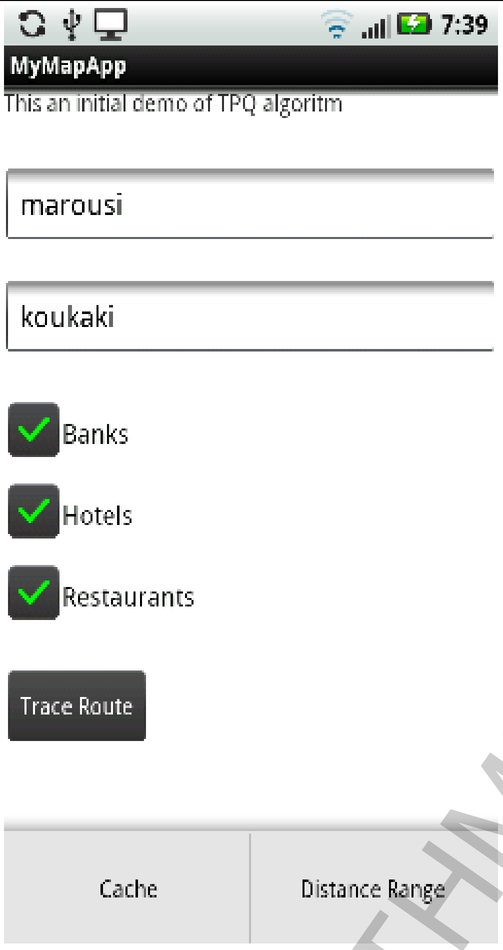
This section describes briefly the main steps of the application by studying a case where the user wants to drive from Marousi to Koukaki and on his way he wants to cover the minimum distance by his car and during his way to stop by a bank, to take money from an ATM machine, to book a room in a hotel, soon afterwards to have a lunch in a restaurant and finally to be in his time in a professional appointment in Koukaki. (figure 12)The goal here is to drive the minimum distance (or spent the less time) in order to reach his appointment. For a foreigner the application suggests a POI that has never been discovered and moreover minimizes this effort.

If the user has cover this route once, he can visit the setting menu by clicking the *menu button* of his mobile device and then choosing the cache option. Then assuming that has already tried once this route choices option “Use cache” (figure 12). This means that the embedded db does not check if the pair of nodes with their distances are available and fetches immediately its distance. In case a use choose “Do not use cache“, the DB checks once for the pair existence that means an extra query for each edge of the road graph and if the search is successful fetches the result, otherwise request form the Google Distance Matrix Api the POI’s distance for caching.

Another available setting is one to choose the range of the distance that approximates the length of the optimal route (see figure 15). This can perform better if the source and destination are quite far away because by default the initial range before the doubling where the algorithm searches the optimal route is 10km.

In figure 16 is the “Loading…” screen that indicates that a process is taking place and the user should be patient (although is quite fast process). As soon the calculations end up the user can see the optimal route along with the POIs of the chosen Categories where it passes through. Tapping a marker shows the facility label for the user convenience. If something fails during the *loading process* the use is informed by a proper error message (figure 19). These messages concern mainly the Google services and are thrown when a user loses internet connection if not cache option is checked or if a Google request’s *over query limit* is reached.

In figure 18 all POIs are marked with a respective marker icon. These are the data instantiations as nodes over the Google map. Their initial form is shown in figure 11.


MyMapApp

This an initial demo of TPQ algoritm

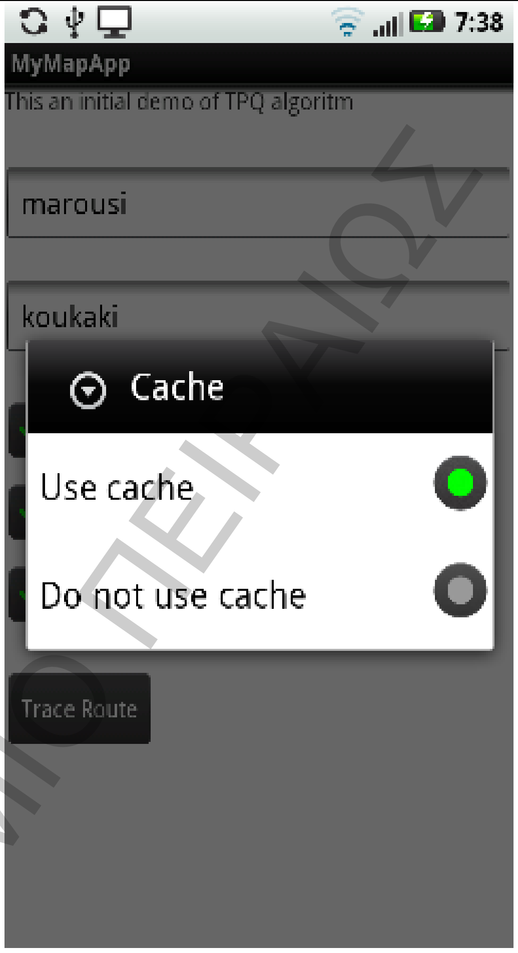
marousi

koukaki

✓ Banks

✓ Hotels

✓ Restaurants


MyMapApp

This an initial demo of TPQ algoritm

marousi

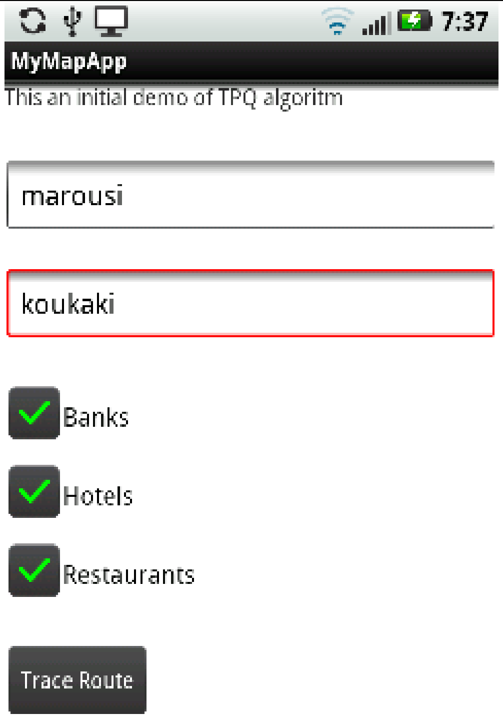
koukaki

Cache

Use cache ●

Do not use cache ○

Fig.12: Setting menu(button) **Fig.13**: cache settings



This an initial demo of TPQ algoritm

marousi

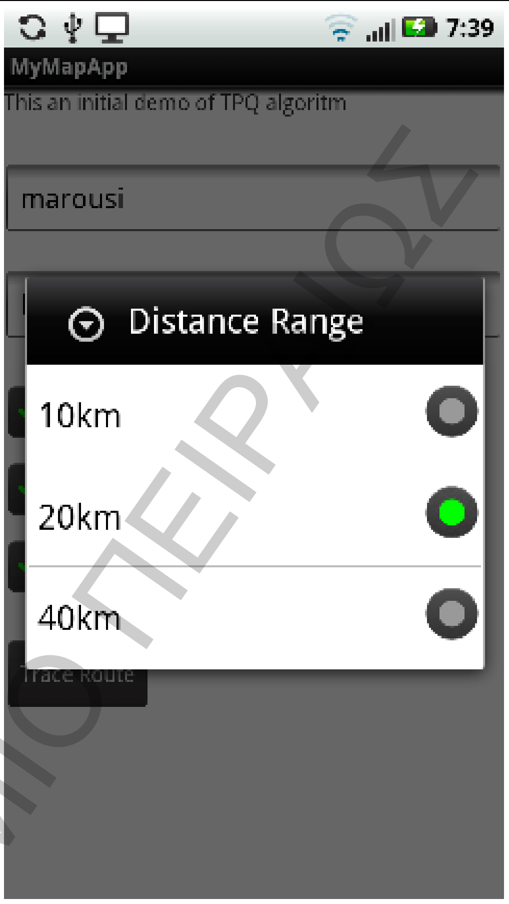
koukaki

✓ Banks

✓ Hotels

✓ Restaurants

Fig.14:initial entry screen



This an initial demo of TPQ algoritm

marousi

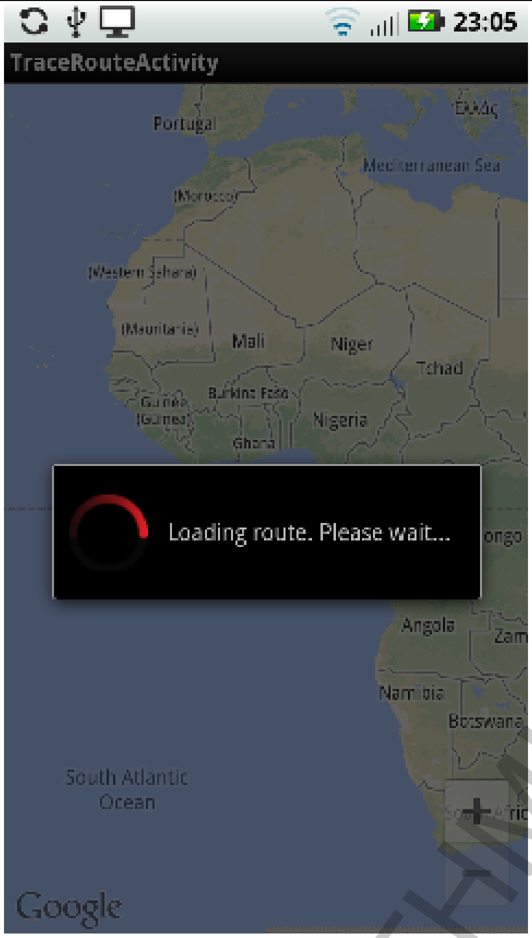
Distance Range

10km

[✓ 20km]

40km

Fig.15: The POIs on Map along with the route



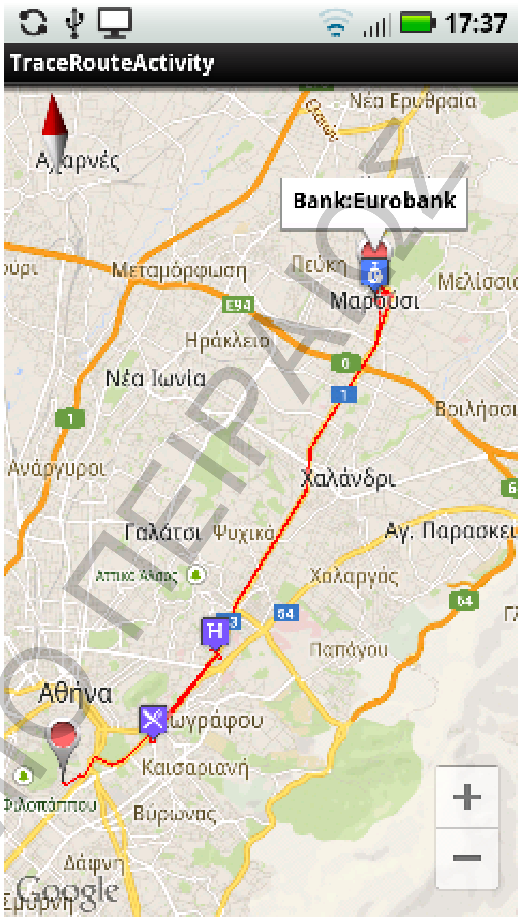
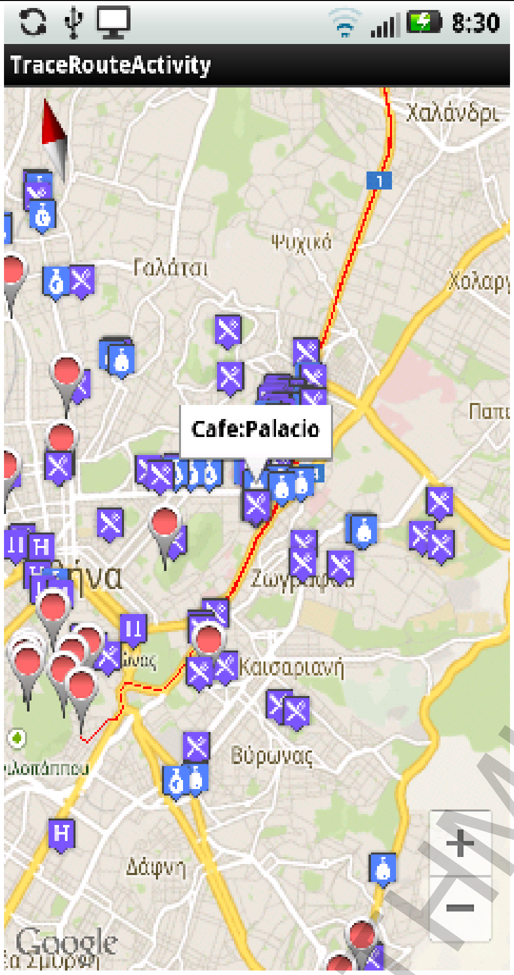
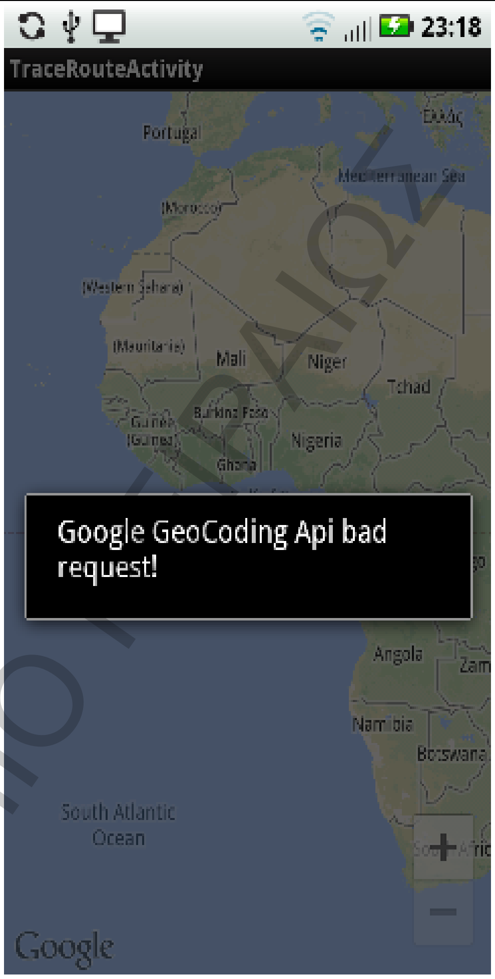


Fig.16: Loading… **Fig.17:** Optimal Route





Google GeoCoding Api bad request!

Fig.18: POIs marked over map **Fig.19:** Custom Exception

4.2 Algorithm steps

In order to have a better inspection of the application algorithm while it is running, a proper reduction the POIs' cardinality (p) was made, while the starting address was set to "Marousi" and the ending address to "Koukaki". The POI Categories are *Hotels* and *Restaurants* and are the following:

1)Set of HOTELS	2) Set of RESTAURANTS
a)Hotel Pantheon (**node 4.0**)	a)Cafe:Starbacks (**node 5.0**)
b)Hotel Nafsika (**node 4.1**)	b)Restaurant:Goodys (**node 5.1**)
c)Hotel Aquamarina(**node 4.2**)	c)Σαλτο Μορταλε (**node 5.2**)
d)Hotel Medousa (**node 4.3**)	

The distances between nodes and the steps of the algorithm (each time it doubles the initial estimated distance) are shown in color below in the graphic representation (figure

20). Finally at the third step (green) terminates with optimal path: source,node4.1,node5.2,end.

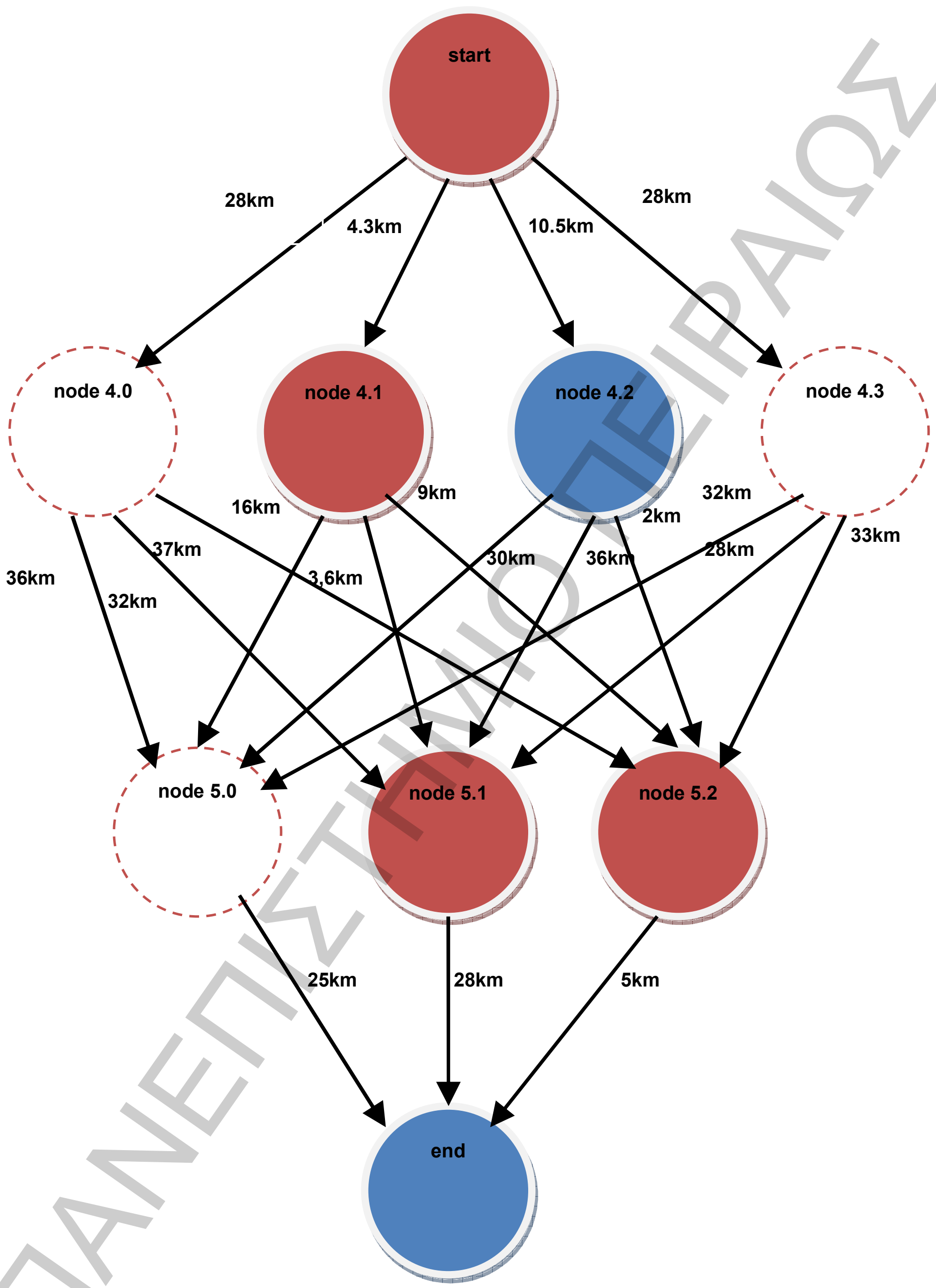


Fig.20: Algorithm execution in steps (final step reaches the end node and terminates)

The algorithm starts with initial threshold (range) 10km. We can distinguish the algorithmic steps as follows:

a) In the three first Dijkstra computations the red nodes are marked as red. The end node finally is out of range and the threshold is doubled.

b) The algorithm starts again with new threshold of 20km. After the three Dijkstra computations the blue nodes are visited marked along with the red from the previous step.

c) As soon as we reached the marked as visited the end node we have a shortest path from start to end that is given in reverse order starting from the destination to its previous node 5.2 and so on.

4.3 Experimental results

Experiments were performed on the road network of Attica,

with 306 POIs from OpenStreetMap[13] in respect with the three Categories (Banks,Hotels and Restaurants) that were used. An augmented graph with 540000 edges on a single core Motorola defy with android 2.2.2 version. As performance metrics I use the process time as well as the total nodes in the constructed Graph (which is equivalent to the number of settled nodes during Dijkstra's computations) the total nodes and the number of edges settled during each iteration as a more robust and platform independent indicator (table 12).

path	***time (sec)***	***settled nodes***	***total nodes***	***settled edges***	***distance covered (meters)***
marousi-koukaki	65	180	306 (3C)	6876	15031
marousi-koukaki	22	67	112 (2C)	2299	14673
marousi-elefsina	25	90	130 (2C)	2264	32632
marousi-elefsina	80	245	306 (3C)	7823	31879

Table 12. Experimental results

5.Conclusion and future work

With this work I tried to consider the problem of answering sequenced route queries and developed an efficient android application that utilizes a speed-up technique to allow the computation in few seconds for realistic sequenced route queries involving common points of interest. The focus of this work has been the case where the order in which the points of interests are to be visited is fixed. This speed-up technique “Iterative Doubling” works well, if the actual result path is relatively short and works well for local data[3]. As we can see from table 12 when we decide to travel for a long distance and while we grow up the involving points of Interest the queries are getting slow.

The restrictive factor that slows down the algorithm is the explicit calculation of large number of inner edges although many of them are pruned by our estimation bound. The second speed-up technique the extended CH can be applied as a future work with the condition that should be running in two parallel threads in a more advanced mobile device with a dual core.

The future of the android application is a very straight forward. Further new features can be added such as the choice of the travel mean (i.e. by feet, cycling) or more POI categories with no prefixed order rather than the user could chose the order himself. Also some assistive tables could be developed in the external memory so they can cache the already optimal route by assuming that the POI sets are mainly static. If we would decide to update these xml format data we can easily do this by appending the new nodes at the end of the respective category. I hope and I wish through this work to help any student who is interested in leveraging the power of android programming and Google services to manage spatial data and answering similar queries with the power of a smart mobile device (with sensors, gps and a variety of capabilities) .

6. REFERENCES

1. Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng On Trip Planning Queries in Spatial Databases.

2. Mehdi Sharifzadeh, Mohammad Kolahdouzan, Cyrus Shahabi, The optimal sequenced route query, The VLDB Journal (2008)

3. Jochen Eisner, Stefan Funke, Sequenced Route Queries: Getting Things Done on the Way Back Home,

4. S. Shekhar and D.-R. Liu. Ccam: A connectivity-clustered access method for networks and network computations. TKDE, 9(1):102–119, 1997

5. M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In SIGMOD, pages 443–454, 2004.

6. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In VLDB, pages 802–813, 2003.

7. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms. The MITPress, 1997.

8. Hutchinson, D., Maheshwari, A., Zeh, N.: An external memory data structure for shortest path queries. Discret. Appl. Math. 126(1), 55–82 (2003). http://www.dx.doi.org/10.1016/S0166-218X(02)00217-2

9. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based K nearest neighbor search for spatial network databases. In: Nascimento,M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B. (eds.) Proceedings of the 30th International Conference on Very Large Data Bases: VLDB'04, pp. 840–851 (2004)

10. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: Freytag, J.C., Lockemann, P.C., Abiteboul, S., Carey, M.J., Selinger, P.G., Heuer, A. (eds.) Proceedings of the 29th International Conference on Very Large Data Bases: VLDB'03, pp. 802–813 (2003).

11. H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The partial sequenced route query with traveling rules in road networks. GeoInformatica, 15(3):541–569, 2011.

12. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In WEA, pages 319–333, 2008.

13.http://downloads.cloudmade.com/europe/southern_europe/greece/attiki#downloads_breadcrumbs

14. http://en.wikipedia.org/wiki/Class_diagram

15. https://developers.google.com/maps/documentation/geocoding/

16. https://developers.google.com/maps/documentation/distancematrix/

17. http://simple.sourceforge.net/

18.https://developers.google.com/maps/documentation/android/start#creating_an_api_project

19. http://www.sqlite.org/docs.html