ASSIGNMENT QUERY AND ITS IMPLEMENTATION IN MOVING OBJECT DATABASES

by

Ali Rıza Konan

B. S., Computer Engineering, Istanbul Technical University, 2003

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2006

ASSIGNMENT QUERY AND ITS IMPLEMENTATION IN MOVING OBJECT DATABASES

APPROVED BY:

Prof. Taflan İmre Gündem .....
(Thesis Supervisor)

Dr. Ayşe Başar Bener .....

Assist. Prof. İlkay Boduroğlu .....

DATE OF APPROVAL: 13.09.2006

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the enthusiastic supervision of Prof. Taflan İ. Gündem during this thesis study. This work would not have been possible without the support and encouragement of him.

I would like to thank Dr. Ayşe Başar Bener and Assist. Prof. İlkay Boduroğlu for their willingness to serve on the committee, and for their constructive suggestions.

I would also like to thank my supervisor in my job, Şerife Dalcı, for her kind support and understanding in difficult times.

My special thanks and appreciation goes to all my mates and friends who shared all the happiness and pains I felt.

Last but not the least, I would like to thank my family. I would not have made it without them. I am fortunate indeed to have such unquestioning love and support. This thesis is dedicated to them.

ABSTRACT

ASSIGNMENT QUERY AND ITS IMPLEMENTATION IN MOVING OBJECT DATABASES

With the rapid development of wireless communications as well as positioning technologies, the concept of moving objects has become more and more important. Moving Objects Databases (MOD) are being used in a wide range of location based services that are of growing interest in many application areas. In the literature, several queries such as nearest neighbor, reverse nearest neighbor, k-nearest neighbor, proximity queries etc. have been considered in moving object databases. Differently from these, in this thesis, a novel operator is proposed as a query type for moving object databases, and also a possible implementation is presented. The aim of the proposed assignment query is to solve the assignment problem, which is also known as weighted bipartite matching. In short, our objective is to find a perfect matching between two set of objects in a manner that minimizes the total cost. For instance, a set of people is to be assigned to a set of taxi-cabs with minimal total travel time.

On the other hand, working with moving object databases, we have to give near real-time responses to user queries to provide an efficient solution for the problem. Unfortunately, the problem of finding a minimal-cost matching for a general bipartite graph is known to have an $O(N^3)$ time algorithm. Thus, we realized that classical solutions having a time complexity of $O(N^3)$ become infeasible for this type of moving object database application. In this thesis, we propose an assignment query that responds in a reasonable time period for MOD. Furthermore, we employ a Q+Rtree index structure to cope with the high update and querying overhead of MOD. At the end, we discussed the performance issues to improve efficiency and showed that the time complexity of our application meets the needs of users in mobile environment.

ÖZET

HAREKETLİ NESNE VERİTABANI SİSTEMLERİ İÇİN ATAMA OPERATÖRÜ VE UYGULAMASI

Kablosuz iletişim alanında olduğu kadar konum bulma teknolojilerinde de yaşanan hızlı gelişmeler sayesinde, hareketli nesneler kavramı gittikçe önem kazanmaktadır. Hareketli Nesne Veritabanı (Moving Object Databases - MOD) sistemleri, birçok alanda giderek artan bir ilgiye sahip olan konum tabanlı hizmetlerde yaygın olarak kullanılmaya başlanmıştır. Literatürde, hareketli nesne veritabanı sistemlerine ilişkin en yakın komşuyu bulma, karşıt en yakın komşuyu bulma, k - en yakın komşuyu bulma gibi birçok sorgulama çeşidi mevcuttur. Bunlardan farklı olarak, bu tez çalışmasında, hareketli nesne veritabanı sistemleri için bir sorgulama tipi olarak yeni bir operatör önerilmiş ve olası bir uygulaması üzerinde durulmuştur. Önerilen atama operatörünün amacı, literatürde ağırlıklı iki parçalı eşleştirme olarak da bilinen atama problemini çözmektir. Özetle, amaç, iki nesne kümesi arasında maliyeti en aza indirecek olan kusursuz atamayı bulabilmektir. Örneğin, bir grup müşterinin bir grup taksiye en az maliyetli olacak şekilde atanması gibi.

Hareketli nesne veritabanı sistemi üzerinde çalışıyor olmamız dolayısıyla, probleme verimli bir çözüm üretebilmek için kullanıcı sorgularına gerçek zamana yakın şekilde cevap verilmesi gerekmektedir. Ancak, en az maliyetli genel iki parçalı eşleştirme probleminin zaman karmaşıklığının $O(N^3)$ olduğu bilinmektedir. Bu yüzden, probleme ilişkin bilinen klasik çözüm yöntemlerinin hareketli nesne veritabanı sistemlerinde kullanılması mantıksız olacaktır. Bu çalışmada, mantıklı bir zaman dilimi içerisinde yanıt verebilecek bir atama operatörü önerilmiştir. Ayrıca, hareketli nesne veritabanı sistemlerindeki yoğun güncelleme yükünü karşılayabilecek bir Q+R ağaç indeksleme yapısından yararlanılmıştır. Son olarak, önerdiğimiz çözümün verimini artırabilecek performans konuları üzerinde durularak uygulamamızın hareketli ortamdaki bir kullanıcının isteklerini karşılayabilecek zaman karmaşıklığına sahip olduğunu gösterdik.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF SYMBOLS xi
1. INTRODUCTION 1
2. PRELIMINARY CONCEPTS AND RELATED WORK 5
2.1. R-Tree and Q-Tree Index Structures 5
2.1.1. R-tree 6
2.1.2. Q-tree 12
3. THE PROPOSED ASSIGNMENT QUERY AND THE SYSTEM TO PROCESS IT 15
3.1. Proposed Assignment Query 16
3.2. Implementation of the Proposed Assignment Query 19
3.2.1. Index Configuration of Our Moving Object Database 19
3.2.1.1. Q+Rtree Index Construction 20
3.2.1.1.1. Building an R*-tree for Quasi-static Objects 20
3.2.1.1.2. Building a Quad-tree for Fast Moving Objects 20
3.2.1.1.3. Combining the Quad-tree and R*-tree Structure 21
3.2.1.2. Updating the Q+Rtree 22
3.2.2. Finding Candidate Vehicles From Moving Object Database 26
3.2.3. Cost Computation 28
3.2.4. Matching Algorithm 31
4. COMPARATIVE PERFORMANCE RESULTS 35
4.1. Details of the Simulation Program 36
4.2. Methodology 37
4.3. Performance Results 38
4.3.1. Results for Randomly Chosen Costs 38
4.3.1.1. Results for Relatively Large Cost Values 39

4.3.1.2. Results for Relatively Close Cost Values 41
4.3.2. Results for Different Number of People and Vehicles 43
4.3.2.1. Results for Higher Number of People 43
4.3.2.2. Results for Higher Number of Vehicles 45
4.3.3. Results for DH Algorithm's Sensitivity to Number of Vehicles 47
5. CONCLUSION 50
APPENDIX A: PARTITIONING A REGION WITH A Q-TREE INDEX 52
APPENDIX B: REPRESENTING A REGION WITH A Q-TREE INDEX 53
APPENDIX C: OVERVIEW OF THE WHOLE SYSTEM 54
REFERENCES 55

LIST OF FIGURES

Figure 2.1. A sample R-tree index 9

Figure 2.2. R*-tree indexing structure 13

Figure 3.1. An example for weighted bipartite graph and flow problem 18

Figure 3.2. An example city map 23

Figure 3.3. Constructing a Q+Rtree 24

Figure 3.4. Flow diagram of updating Q+Rtree process 27

Figure 3.5. Flow diagram of finding candidate vehicles 29

Figure 3.6. Computing weights on a road network 32

Figure 3.7. The greedy algorithm for finding maximum weight matchings 35

Figure 4.1. A sample run of the simulation program 38

Figure 4.2. Time complexity of matching algorithms for large cost values 41

Figure 4.3. Time complexity of DH algorithm for large cost values 42

Figure 4.4. Total costs produced by the matching algorithms for large cost values 43

Figure 4.5. Time complexity of matching algorithms for close cost values 44

Figure 4.6. Total costs produced by the matching algorithms for close cost values 44

Figure 4.7. Time complexity of matching algorithms for cases in Table 4.1 46

Figure 4.8. Total costs produced by the matching algorithms for cases in Table 4.1 ... 47

Figure 4.9. Time complexity of matching algorithms for cases in Table 4.2 48

Figure 4.10. Total costs produced by the matching algorithms for cases in Table 4.2 ... 49

Figure 4.11. Comparative execution times of DH algorithm for constant number of people 50

Figure 4.12. Comparative total costs produced by DH algorithm for constant number of people 51

Figure 4.13. Total costs produced by DH algorithm for 10,000 people 52

Figure A.1. Indexing space with a Q-tree index ……..……..... 53

Figure A.2. Q-tree representation of a geographical region ……... 54

Figure A.3. Overview of the whole system …… 55

LIST OF TABLES

Table 4.1. Cases for higher number of people 44

Table 4.2. Cases for higher number of vehicles 46

LIST OF SYMBOLS

ARCS	Set of arcs in weighted bipartite graph
$assign_{pv}$	Binary array indicating the assignment
$COST_{pv}$	Cost of a path between a person p and a vehicle v
G	Weighted bipartite graph
M	Size of set P
N	Size of set V
P	Set of people
V	Set of vehicles

1. INTRODUCTION

The rapid and continued advances in positioning systems, e.g., GPS, wireless communication technologies, and electronics in general promise to render it increasingly feasible to track and record the changing positions of objects capable of continuous movement. The coming years will witness dramatic advances in wireless communications as well as positioning technologies. As a result, tracking the changing positions of objects capable of continuous movement is becoming increasingly feasible and necessary [10]. As existing Database Management Systems (DBMS) are not well equipped to handle continuously changing data, Moving Object Databases (MOD) are considered to keep track of object locations and support location-aware queries [21].

To keep the track of moving objects and process user queries, the frequent location changes are to be considered. In a moving object database system, location updates are sent to the database dynamically. These location values are then used to evaluate the user queries. Due to continuous changes in locations and limited resources (battery power, bandwidth and etc.), it may become infeasible for the database to keep track of the actual location of every moving object in the workspace [10, 20, 22]. Using the past values provided by the database may result in inaccurate inferences. However, limiting the degree of uncertainty between the actual location value and the database value makes the answers to the queries more reliable. In other words, the user queries can be evaluated in a probabilistic range that gives the approximate location of the object [21].

In literature, there are many types of queries for moving object databases. A survey of query types in mobile environments is given in [29, 32]. One of the most commonly used query type is range queries [14]. A range query describes a region in space and asks for all points or the number of points in the region. Another widely-known query type in MOD is nearest neighbor queries [13]. The nearest neighbor query ranks all objects in terms of their distance from a query object. Apart from these query types, in this thesis, we propose an operator for moving object databases as a new query type called *ASSIGN_OBJ*

that solves the assignment problem and we focus on a possible implementation for this query.

Assignment problem is also known as *the weighted bipartite matching problem* in literature. This means that the assignment is done over a bipartite graph. A graph is bipartite if it has two kinds of nodes and the edges are only allowed between nodes of different kind. A matching in a bipartite graph therefore assigns left-hand vertices to right-hand vertices of the bipartite graph. The objective is to find a perfect matching between these two set of nodes [23, 24]. In our work, left-hand vertices of the bipartite graph are considered as *query objects* where right-hand vertices are *data objects*. Our goal is to minimize the total cost after the assignment is completed.

The problem of finding a minimal-cost matching for a general bipartite graph is known to have an $O(N^3)$ time algorithm (see [11, 12] for this and other background on matching). As we are working with MOD and have to response in real time, the degree of the time complexity of the assignment should be reduced to an acceptable range. This means that we cannot use the classical solutions for the minimal cost weighted bipartite matching. Our proposed implementation for the assignment query tends to have an approximately $O(N)$ time complexity that seems suitable for MOD applications. As we are dealing with users in a mobile environment, response time of the system should be reasonable. This is because users tend to change their locations during the assignment process. We intend to meet the needs of users in a mobile environment by employing a linear-time matching algorithm.

We define three types of assignment queries depending on the following mobility cases of query and data objects:

i. Both query and data objects are mobile
ii. Query objects are mobile while data objects are stationary
iii. Query objects are stationary while data objects are mobile

In this thesis, we concentrate on the third case above and propose a possible implementation for this case. To give a motivating scenario, we can consider a number of

people requesting a service like taxi-cab, car assistance, first-aid unit or ambulance. In this scenario, people are modeled as query objects and vehicles to service are modeled as data objects.

Only the data objects send location updates to the moving object database. Thus, we employ an efficient storage structure in order to keep the location values of the data objects and process search queries. In our proposed system to process the assignment query, we utilize a combined tree index structure, namely *Q+Rtree* (see [2]), for our moving object database. Underlying reasons to choose this approach to build our storage structure are explained below:

The design of index structures is based on both optimizing the search performance and supporting efficient data updates. Although it is not stated clearly, index structures assume that the rate of updates in a database system will be smaller than the rate of querying. However, this claim is not valid for newly challenging areas like moving object database systems. Updates are received continuously in those applications such a way that the rate of updates may exceed the rate of querying. Consequently, index structures may give infeasible results because of the large overhead of keeping the index updated with the latest data. Recent efforts focus on indexing moving object data assuming that objects move in a restrictive manner (e.g. in straight lines with constant velocity) [1, 2]. In this work, we employ an index structure explicitly designed to perform well for both querying and updating. We assume a more relaxed model of object movement. Our storage structure is developed with a modified version of the combination of Q-tree [5] and R*-tree [3] structures.

After in-depth investigation of the recent papers and publications in literature, we observed that differentiating the *fast moving objects* from *quasi-static objects* reduces the update cost to a great extent by utilizing a different index structure for each. In our storage structure, which is a modification of Q+Rtree structure in [2], quasi-static objects are stored in an R*-tree and fast moving objects are stored in a Quad-tree. Objects may switch between these two trees when they change their moving status, e.g., if a data object moves out of a range, it will change from quasi-static state into the fast-moving state. Although R*-tree alone has a good query performance, it gives a poor index update performance.

Conversely, Quad-tree's index updating performance is good (when an appropriate page size is chosen), but its query performance is worse when compared with the R*-tree. Combining these two tree structures together, a Q+Rtree gives better performance for both index updating and query evaluation [2].

The rest of this report is organized in the following order. Firstly, in Section 2, we introduce the preliminary concepts and related work to give a better understanding of the proposed algorithms and supporting index structures in the system. In Section 3, the proposed assignment query for moving object databases is presented, and a possible implementation of it is described in detail. Also, the details of our storage structure are given in Section 3. Performance issues are discussed in Section 4. Lastly, the thesis is concluded in Section 5.

2. PRELIMINARY CONCEPTS AND RELATED WORK

To give a clear understanding of our work, we give information about preliminary concepts and related work in this section. We believe that giving some introductory information on our indexing structure will be reasonable to understand the subsequent sections of this report.

As it is mentioned in Section 1, we utilized and modified a previously proposed combined tree index in [2], namely Q+Rtree, in our storage structure. This combined index structure is composed of an R*-tree and a Q-tree index. In this section, we give detailed descriptions of R-trees (also, a possible R*-tree index for MOD is explained) and Q-trees to clarify the construction of a Q+Rtree.

2.1. R-Tree and Q-Tree Index Structures

In this thesis, the moving object database system is composed of combined R*-tree and Q-tree index structures [3, 5] to provide more efficient update and search mechanisms on location data.

Taking the advantage of utilizing a digital map of the topographical area where the moving objects are located and relaxing the object movement, the thesis is mainly separated from the related work in literature. Our index structure handles fast moving and quasi-static objects separately to take a more accurate picture of the reality and result in better performance. There is no assumption made about the future positions of objects. Objects are not obliged to move according to well-behaved patterns and there are no restrictions, like the maximum velocity, placed on objects either.

As it is mentioned before, this thesis is based on the two categories of moving objects called *quasi-static objects* and *fast moving objects*. Quasi-static objects are indexed over an R*-tree while fast moving objects are indexed over a Q-tree:

Firstly, we observe that an appreciable number of the moving objects do not move at high speeds in time. Those objects are in a quasi-static state, which means the object is not completely static but rather is moving within a small region of space. For instance, two customers call for a nearest taxi around. To assign the nearest taxi to a customer, taxi cabs are considered as moving objects. Available taxi cabs may be stationary (parked / moving very slow) or moving fast (driving for a customer). In this thesis, stationary ones are treated as quasi-static objects and kept in the R*-tree.

Secondly, in our case, fast moving objects are considered as taxi cabs, ambulances, first aid units, and etc. that are ready to service and already driving on a road or freeway. These moving vehicles are moving at high velocities. Thus, rate of updates is also high. To reduce the overhead of update cost, Quad-tree (Q-tree) indexing structure is used for fast moving objects.

In the following sections, R*-tree and Q-tree indexing structures are explained in detail to give a clear understanding of the thesis.

2.1.1. R-tree

Moving at smaller velocities, quasi-static objects are expected to have much lower update frequency. The movement area of quasi-static objects is also small, e.g. a parked taxi cab that waits for a customer in a taxi station, an ambulance waiting for an emergency case in the garden of a hospital. For those quasi-static objects, using an R-tree based index structure, proposed in [15], is preferable so as to provide good query performance.

R-trees are tree data structures that are similar to B-trees, but are used for spatial access methods i.e., for indexing multi-dimensional information, (X, Y) coordinates of geographical data. A common real-world usage for an R-tree might be: “Find all museums within 2 miles of my current location”. The data structure splits space with hierarchically nested, and possibly overlapping, boxes. Each node of an R-tree has a variable number of entries (up to some pre-defined maximum). Each entry within a non-leaf node stores two pieces of data; a way of identifying a child node, and the bounding box of all entries within this child node [15, 25].

The insertion and deletion algorithms use the bounding boxes from the nodes to ensure that nearby elements are placed in the same leaf node (in particular, a new element will go into the leaf node that requires the least enlargement in its bounding box). Each entry within leaf nodes stores two pieces of information; a way of identifying the actual data element (which, alternatively, may be placed directly in the node), and the bounding box of the data element. Similarly, the searching algorithms (for example; intersection, containment, nearest) use the bounding boxes to decide whether or not to search inside a child node. In this way, most of the nodes in the tree are never traversed during a search. Like B-trees, this makes R-trees suitable for databases, where nodes can be paged to disk when needed [26].

Furthermore, an R-tree is a height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required [26].

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form **(I, tuple-identifier)** where tuple-identifier refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed [15]:

$$\mathbf{I = (I_0, I_1, I_2, \ldots, I_{n-1})}$$

Here n is the number of dimensions and I_i is a closed bounded interval [a,b] describing the extent of the object along dimension i. Alternatively I_i may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely. Non-leaf nodes contain entries of the form **(I, child-pointer)** where child-pointer is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries.

Let **M** be the maximum number of entries that will fit in one node and let **m ≤ (M / 2)** be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties [15]:

a- Every leaf node contains between m and M index records unless it is the root.

b- For each index record (I, tuple-identifier) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.

c- Every non-leaf node has between m and M children unless it is the root.

d- For each entry (I, child-pointer) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.

e- The root node has at least two children unless it is a leaf.

f- All leaves appear on the same level.

The following Figure 2.1 shows a sample for R-tree indexing structure:

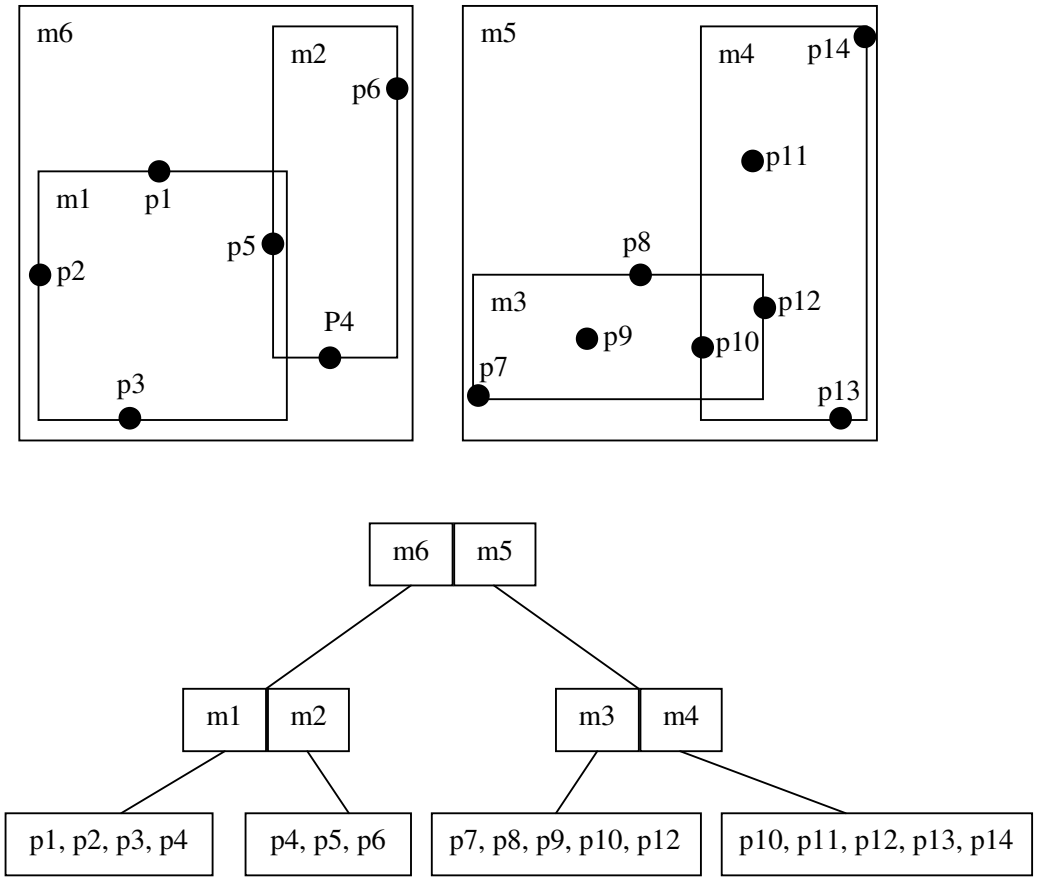


Figure 2.1. A sample R-tree index

All nodes in the R-tree, both internal nodes and leaf nodes, represent d-dimensional hyper-rectangles. While the leaf level nodes are rectangles containing moving objects, the rectangles for internal nodes contain rectangles one level below. The rectangle boundaries should be kept as tight as possible in order to improve query evaluation. These rectangles are called ***Minimum Bounding Rectangles*** or **MBR**s. On the contrary of B-Tree, overlapping is allowed for the MBRs of nodes at the same level in an R-Tree. Consequently, traversing several paths in the tree may be necessary to search an object in one of the leaf rectangles. When a node becomes overfull it undergoes a split. Efficient heuristics and pruning are used to reduce the expected number of paths visited by subsequent searches [1].

In our storage structure, **R*-tree** index, which is a variant of R-tree, is used with making slight changes to adapt our assignment problem. R*-tree structure is proposed in [3] covering the following contributions:

- **The area covered by a directory rectangle should be minimized**. i.e. the area covered by the bounding rectangle but not covered by the enclosed rectangles, the dead space, should be minimized. Thus, it will improve performance since decisions which paths have to be traversed, can be taken on higher levels

- **The overlap between directory rectangles should be minimized.** Thus, also decreases the number of paths to be traversed.

- **The margin of a directory rectangle should be minimized.** Here, the margin is the sum of the lengths of the edges of a rectangle. Assuming fixed area, the object with the smallest margin is the square. Thus, minimizing the margin instead of the area, the directory rectangles will be shaped more quadratic. Essentially, queries with large quadratic query rectangles will profit from this optimization. More important, minimization of the margin will basically improve the structure. Since quadratic objects can be packed easier, the bounding boxes of a level will build smaller directory rectangles in the level above. Thus, clustering rectangles into bounding boxes with only little variance of the lengths of the edges will reduce the area of directory rectangles.

- **Storage utilization should be optimized.** Higher storage utilization will generally reduce the query cost as the height of the tree will be kept low. Evidently, query types with large query rectangles are influenced more since the concentration of rectangles in several nodes will have a stronger effect if the number of found keys is high.

In our work, the quasi-static data objects are vehicles like taxi cabs, ambulances, or first aid units that waits for a call from people. These vehicles mostly stay motionless or move at slow velocities in a small region to pick up a customer. Thus, we can place these quasi-static objects in MBRs of our modified R*-tree structure. If we employ an R*-tree for the quasi-static objects, they can move out of their current MBR with a small chance. At this point, we can also take the advantage of using *Lazy Update Approach* proposed in [4]. In this manner, a large amount of index updating overhead is decreased. This approach

updates the structure of the index only when an object moves out of the corresponding MBR. If the new position is still within the MBR, only the position of the object in the leaf node is updated. Furthermore, if we take a look at the distribution of objects, quasi-static objects are often crowded together (e.g. taxi stations, parked vehicles,…), which makes the low-level MBR of the R*tree small and packed. This will increase the precision of the index and speed up searching (in terms of efficient pruning during query evaluation) [2].

As we have a more relaxed approach in our work, we modified some properties of the R*-tree described above. For instance, the number of object pointers in a leaf node should not be limited. This means that there may be arbitrary number of vehicles in any MBR. In addition, no overlaps between MBRs should be allowed. The modification we have made for the R*-tree structure is detailed below:

(1) The level above the leaf nodes in the tree are for topographical regions. Differently from a traditional R*-tree, no overlap between the nodes at this level can occur. In a traditional R*tree, overlaps between MBRs at each level are allowed.

(2) There are no presumed restrictions on the number of object pointers in the leaf nodes. All objects that belong to a given topographical region are inserted below that region. This also makes insertion an easier procedure since there is only one place to insert an object. Conversely, in traditional R*tree, an object could be inserted into any node and large amounts of computation needs to be performed to determine which node it should be inserted into.

(3) Objects on the boundary of an MBR can easily move out of the current MBR. If an object passes the boundary of its current MBR and then quickly returns into that MBR, following insertions and deletions may be necessary. Thus, the update performance can degrade. To prevent this problem, ***Extended MBRs* (EMBRs)** proposed in [4] is used in our work. EMBR defined for only leaf nodes is a slightly large bounding rectangle instead of an MBR. Unnecessary

update cost is avoided by this approach. The EMBR is larger than the corresponding MBR.

(4) There may be quasi-static data objects that are not assigned to any MBR. In our work, we immediately assign these objects to the nearest MBR. This is a novel approach for previous related works on R*-tree that also reduces the update cost of the system. This is because quasi-static objects send updates only when they move out of their current MBR.

A topographical R*-tree structure is given in Figure 2.2:

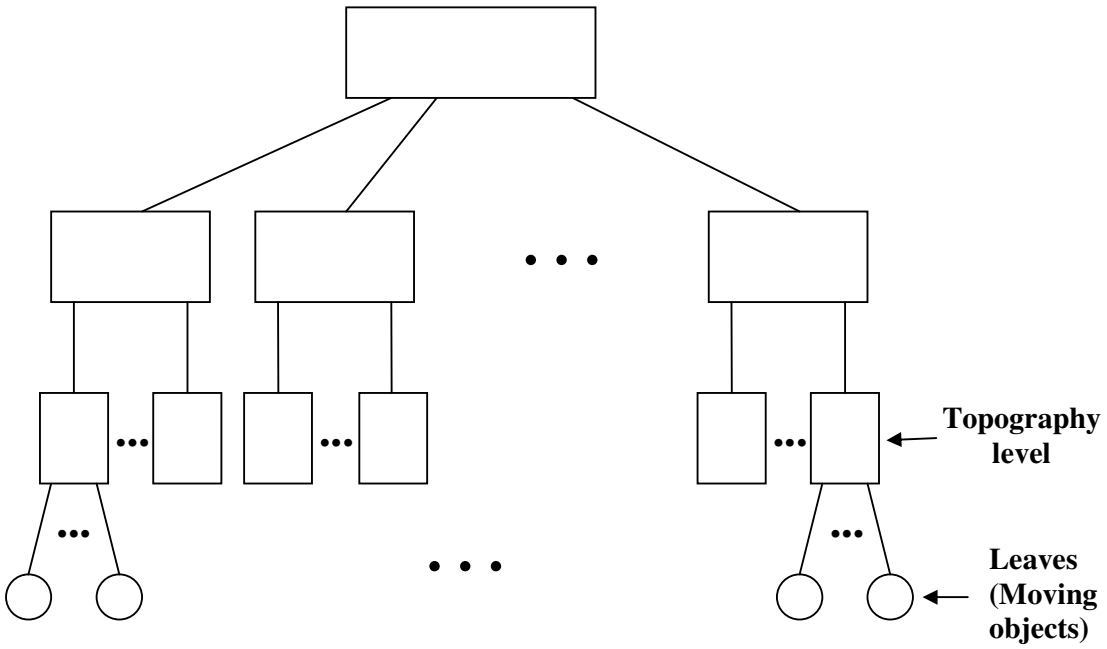


Figure 2.2. R*-tree indexing structure

2.1.2. Q-tree

In our work, a Quad-tree index, proposed in [15], is employed for the modeling of fast moving objects. The reason for not using an R*-tree indexing is that these fast-moving objects are likely to often move out of their current MBRs. Thus, a significant increase is

observed in the index updating cost. Furthermore, fast-moving objects are mostly probable to be spread over a wide topographical region instead of staying together. If we have used an R*-tree index for those fast moving objects, the leaf nodes would have to carry a great number of objects and the querying performance would degrade expectedly. On the other hand, Quad-tree indexing structure is a feasible solution for this case. In a Q-tree, geographical area, where the fast moving objects are located, is separated into local quadrants. Each quadrant is supposed to carry a certain number of objects. When the number of the objects in such a quadrant is low, the coverage area of the quadrant can be very large. Therefore, even if objects are moving fast, there is a small chance that the object will move out of its current quadrant, which makes it easy to update the index. If the object still stays in its current quadrant, no update to the index structure is needed [2].

The Quad-tree is a tree data structure in which each internal node has up to four children. Quad-trees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quad-tree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a *Q-tree*. All forms of Quad-trees share some common features [5]:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the Quad-tree

In a Q-tree, each node is split along all d dimensions, leading to 2^d children. It was originally designed for 2-dimensional data, so each node has four children. It is a rooted tree so that every internal node has four children. Every node in the tree corresponds to a square. If a node has children, their corresponding squares are the four quadrants.

Quad-tree structures have also received great interest in the field of image processing. As it is described below, a quad-tree is a spatial data structure built by a recursive decomposition of space into quadrants. Applied to images, it allows representing image content, compacting or compressing image information, and querying images. In recent

years, numerous image-based approaches have used this structure. The contribution of quad-tree in image applications is confirmed by several authors in literature.

Dividing a region into four sub-regions and constructing a sample Q-tree is illustrated in Appendix A and Appendix B, respectively.

Quad-trees can store many kind of data, we describe the variant that stores a set of points. For the definition a simple recursive splitting of squares is continued until there is only one point in a square. Let *P* be a set of points.

The definition of a quad-tree for a set of points in a square

$$Q = [x1_Q : x2_Q] £ [y1_Q : y2_Q]$$

is as follows:

- If $|Pj| \leq 1$ then the quad-tree is a single leaf where Q and P are stored.
- Otherwise let Q_{NE}, Q_{NW}, Q_{SW} and Q_{SE} denote the four quadrants.

Let $x_{mid} := (x1_Q + x2_Q)=2$ and $y_{mid} := (y1_Q + y2_Q)=2$, and define

$P_{NE} := \{ p \in P : p_x > x_{mid} \text{ and } p_y > y_{mid} \}$

$P_{NW} := \{ p \in P : p_x \leq x_{mid} \text{ and } p_y > y_{mid} \}$

$P_{SW} := \{ p \in P : p_x \leq x_{mid} \text{ and } p_y \leq y_{mid} \}$ and

$P_{SE} := \{ p \in P : p_x > x_{mid} \text{ and } p_y \leq y_{mid} \}$

The quad-tree consists of a root node n, *Q* is stored at n. In the following, let *Q*(*v*) denote the square stored at v. Furthermore v has four children: The X-child is the root of the quad-tree of the set P_X for X Є { NE, NW, SW, SE }[5].

3. THE PROPOSED ASSIGNMENT QUERY AND THE SYSTEM TO PROCESS IT

In literature, there are a number of well-known query types for MOD such as range queries and k-nearest neighbor (k-NN) queries. These types of queries have received a lot of attention from researchers working on MOD [29]. In this thesis, we focus on a novel query type to solve the assignment problem in MOD. We call this type of queries as *Assignment Queries*.

In assignment problem (also known in literature as the weighted bipartite matching problem [23, 24]), the aim is to assign n items to another m items in a manner that minimizes or maximizes an optimization objective. In this thesis, we propose an assignment operator as a type of query for MOD and a possible implementation of it. In an assignment query for MOD, we have two set of objects. The first set consists of *query objects* that are requesting a kind of service, e.g. people calling an ambulance in an emergency case, or a taxi-cab for transportation purposes. On the other hand, we have another set of *data objects* that are dedicated for a kind of service such as vehicles like taxi cabs, ambulances, and etc. An overview of the whole system is given in Appendix C.

We can define three different types of assignment queries by considering the mobility of query and data objects. First type of assignment query may be defined on mobile query objects and mobile data objects. The second type may be defined on mobile query objects and stationary data objects. We can define the third type of assignment query on stationary query objects and mobile data objects. In this thesis, we concentrated on implementation of the third type of assignment query where the query objects are stationary and the data objects are mobile. A good example for this type of query may be finding taxi-cabs for people in a city. We concentrated on this example throughout the rest of the thesis. Also, our implementation can be extended to support the other two types of assignment queries.

Since moving object databases do not store the precise values of object locations at all times, the solution of the assignment problem in MOD will not be exact. Thus, finding an approximate solution for the assignment problem is also acceptable.

Proposed assignment query, details of our implementation and update mechanism of our storage structure are explained in the following sections.

3.1. Proposed Assignment Query

Let P be the set of people with size M, V the set of vehicles with size N, and $COST_{pv}$ the cost of a path between a person p and a vehicle v. A person has to be assigned to a single vehicle and a vehicle only takes one person. We define an $M*N$ sized binary array variable, $assign_{pv}$. The elements of this array take value 1 if and only if a person p is assigned to a vehicle v, otherwise 0, as given in (1):

$$For\ all\ p \in P,\ \ all\ v \in V : assign_{pv} \in \{0,\ 1\} \quad \textbf{(1)}$$

The fact that a person p is assigned to a single vehicle v is given by equation (2). Similarly, to express that a vehicle v services a single person p, we have constraints (3):

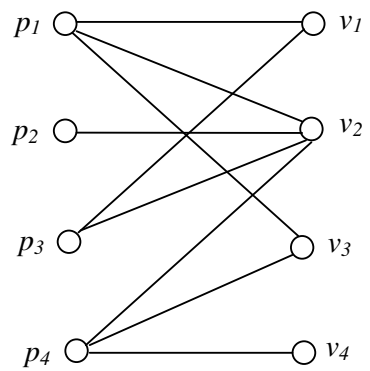
$$For\ all\ p \in P \ \ : \ \ \sum_{v \in V} (assign_{pv}) = 1 \quad \textbf{(2)}$$
$$For\ all\ v \in V \ \ : \ \ \sum_{p \in P} (assign_{pv}) = 1 \quad \textbf{(3)}$$

We minimize the objective function, which is the sum of $COST_{pv}$ for the variables $assign_{pv}$ that are at 1, as given in (4). The resulting mathematical model is formed by the equations (1) to (4):

$$\text{minimize} \sum_{p \in P} \sum_{v \in V} (COST_{pv} \cdot assign_{pv}) \quad \textbf{(4)}$$

The assignment problem may be looked at as a flow problem [30, 35]. It is sufficient to define a weighted bipartite graph $G = (P,\ V,\ ARCS,\ COST_{pv})$, where P is a set of nodes representing the people, V a set of nodes for the vehicles, and $ARCS$ the set of arcs describing the possible assignments of vehicles to people (Figure 3.1). Every arc (p,v) is

labeled with the weight $COST_{pv}$ that indicates the cost of the path between p and v. We create a source node S that is connected to every person-node p by an arc (S, p). We then also create a sink node T to which are connected all vehicle-nodes v by arcs (v, T). An optimal assignment corresponds to a flow in G with minimum total cost. Consequently, the flow will trace M disjunctive paths from S to T that indicate the assignments. Note that the total number of paths from S to T equals M at maximum. This is why there may be unassigned vehicles in V after the assignment while all people in P must be assigned to a vehicle.



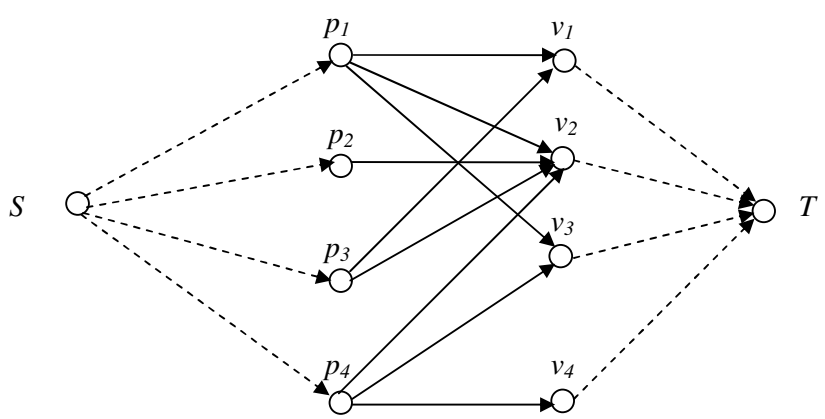


Figure 3.1. An example for weighted bipartite graph and flow problem

The constraints given by equation (1) may be replaced by simple non-negativity conditions. It is not necessary to specify that the variables $assign_{pv}$ must not be larger than 1 because this is guaranteed through the constraints (2) and (3). The model can be modified to deal with sets of people and vehicles of different size as it is mentioned above. If, for

instance, we have more people than vehicles, we keep the constraints (3) so that every vehicle services one person. Thus, we replace the constraints (2) by (5).

$$\textit{For all } p \in P \;:\; \sum_{v \in V} (assign_{pv}) \leq 1 \quad \textbf{(5)}$$

Certain assignments may be infeasible. In such a case, the value of the corresponding variable $assign_{pv}$ will naturally be forced to 0.

Finally, we can formulate ***ASSIGN_OBJ***, proposed assignment query for our moving object database, as follows:

ASSIGN_OBJ (***P:*** set of people, ***OUTpv****:* output)

where,

P is an array structure that keeps the (x,y) coordinates and identity value of each person,

OUTpv is a binary valued output matrix that keeps the person-vehicle assignment results.

The assignment query does not take inputs like vehicle identity values, coordinates of vehicles to be assigned, or costs of the paths between people and vehicles. This is because we determine the information about the vehicles and costs for the weighted bipartite graph by utilizing our moving object database. For instance, we find the candidate vehicles from our dynamically updated database and then calculate the weights due to our cost function.

Our assignment query, *ASSIGN_OBJ*, begins with finding the candidate vehicles to be assigned over the moving object database. Then, the costs of paths between people and those candidate vehicles are computed before doing the assignment job. The costs of the paths between each person and its candidate vehicles are placed as weights on our bipartite graph. In the next chapters, candidate vehicle searching and cost computation processes are also presented.

3.2. Implementation of the Proposed Assignment Query

We have to determine the left-hand and right-hand vertices of the weighted bipartite graph at first. In our case, left-hand vertices are sampled as people requesting a kind of service from vehicles. These person-nodes are taken as input by the assignment query at the beginning. We assume that the coordinates and path identity values of people are collected from their PDAs or mobile phones. Then, we have to find out the right-hand vertices of our bipartite graph. These left-hand vertices correspond to candidate vehicles to be assigned. We determine the candidate vehicles for each person by traversing our moving object database. Finally, to begin processing the assignment job, the weights of arcs between left-hand and right-hand vertices are determined by the cost computation process. All of these implementation steps are presented in this section.

First of all, we explain the index configuration of our moving object database in this section. Giving information on our indexing structure will be helpful to clearly understand the modules of our assignment query, especially the candidate vehicle searching process. After giving details of our moving object database, we continue with explaining the implementation modules of our assignment query.

3.2.1. Index Configuration of Our Moving Object Database

All the index structures offered for moving objects in literature had to struggle with two opposing requirements. The first one is about placing large capacity internal nodes in order to prevent the objects from moving out of those nodes' coverage area. In this manner, the probability of changing the index structure after an update request decreases noticeably. The second issue is on improving the querying performance. To enhance the query processing over the tree index, utilizing small and tight internal nodes becomes reasonable so that the index can efficiently support queries. We take the advantage of separating quasi-static and fast moving objects by building a loose Q-tree index for fast objects and a tight R*-tree index for quasi-static objects. Consequently, we succeed in improving both update and querying efficiency using a combined Q+Rtree index [2].

In this part, we describe how to construct the Q+Rtree index and then, clarify the update mechanism of Q+Rtree. We also comment on how we modified the Q+Rtree and adapt it to our problem.

3.2.1.1. Q+Rtree Index Construction

Building a Q+Rtree consists of three main steps. The process begins with constructing the topographical R*-tree for quasi-static objects, then the Q-tree is constructed for fast moving objects. Finally, these two tree index structures are combined to form the Q+Rtree by linking the leaf nodes of each tree. This process is explained in the following sections of the thesis.

3.2.1.1.1. Building an R*-tree for Quasi-static Objects

An R*-tree for the objects in quasi-static state can be simply built by either inserting the objects one by one into the R-tree or performing a bulk loading operation [2, 31]. However, in our work, a different approach is chosen to build the index. We build an R*-tree index over topographical regions where the quasi-static objects are densely located instead of building the index using the individual quasi-static objects. Determining the regions to build the index is done by means of analyzing geographical maps if available, or investigating the past behaviour of objects, i.e. determining the regions where the free taxi cabs are situated in a period of time. It is important to emphasize that the boundaries of the topographical regions represented in the index structure should be specified carefully in order to force quasi-static object not to move out of their current MBRs for a certain amount of time. Once this index is created, the quasi-static objects are inserted into the index while treating the topographical objects from the previous step simply as MBR one level above the new leaf level – at which the objects are stored.

3.2.1.1.2. Building a Quad-tree for Fast Moving Objects

The second step is to build a Quad-tree over the entire map where the fast moving objects are located by inserting all objects not in the slow-moving cells. This process is identical to the regular Quad-tree operations.

3.2.1.1.3. Combining the Quad-tree and R*-tree Structures (Q+Rtree)

It is obvious that the Q-tree and the R*-tree overlap topographically, as both of them represent the objects on the same geographical area. For efficiency purposes, pointers are located at the leaves of the Q-tree. Pointer of a quadrant in the Q-tree indicates that the related R*-tree node is contained by or intersecting that quadrant. The R*-tree nodes pointed to by a pointer in the Q-tree can be at different levels. A sample Q+Rtree for our case is illustrated in the Figure 3.2 and Figure 3.3 below:

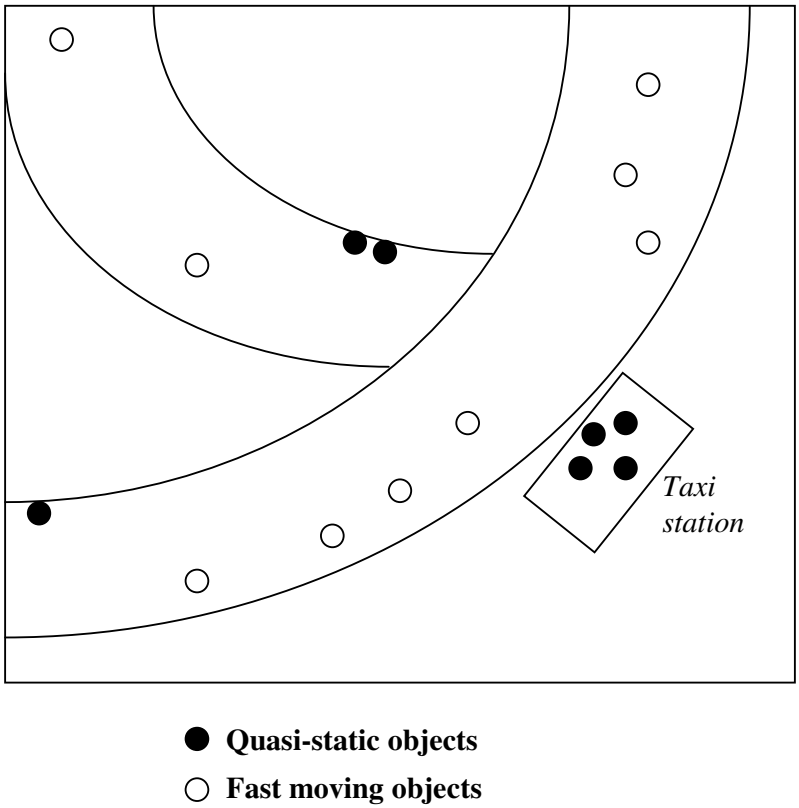


Figure 3.2. An example city map

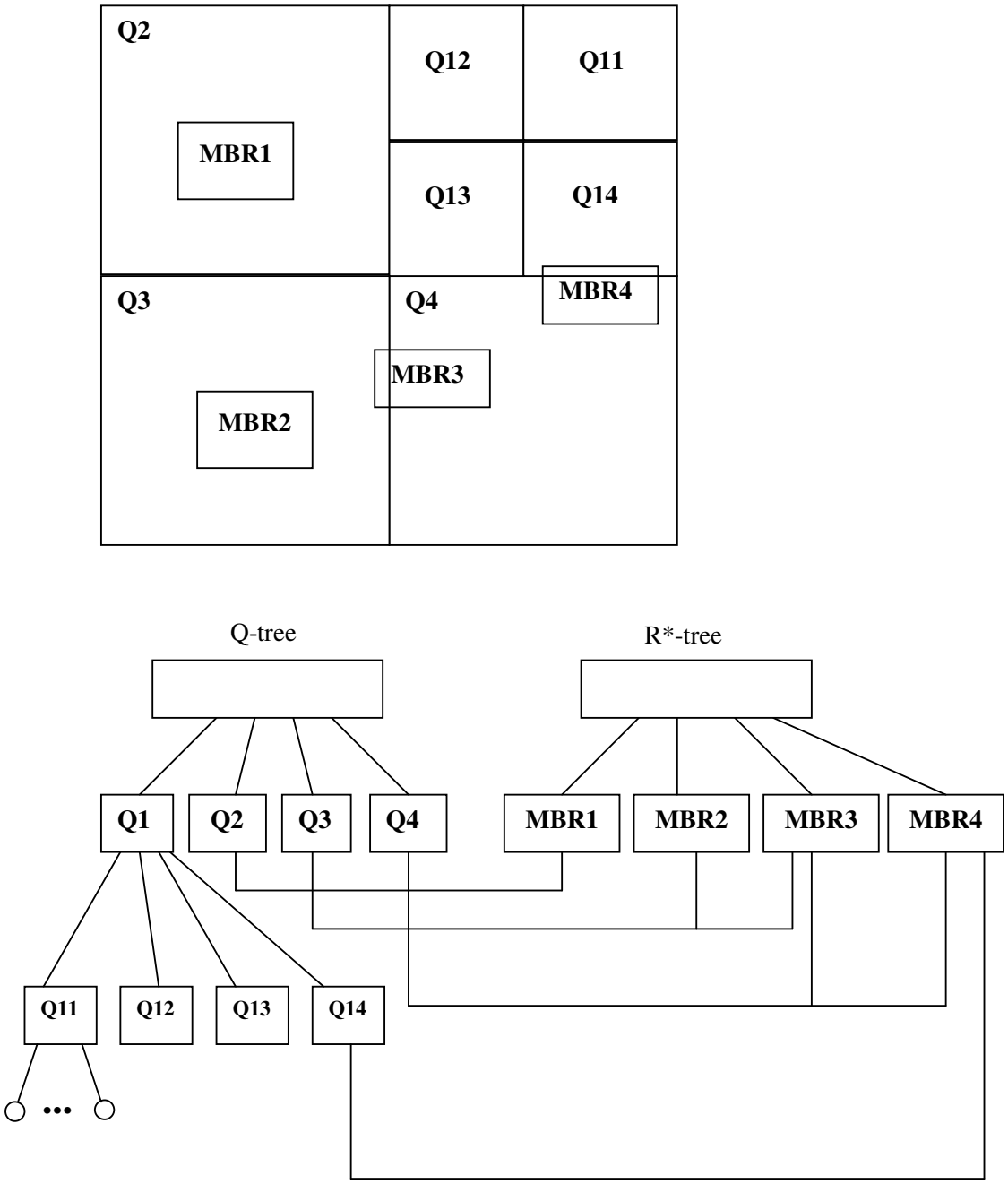


Figure 3.3. Constructing a Q+Rtree

3.2.1.2. Updating the Q+Rtree

We described how to construct the Q+Rtree. In this section, the update process of previously constructed Q+Rtree is explained.

When an update arrives with an object's new location, it could be one of the following cases [2]:

1. The object is currently in the R*-tree, and its new position is still in the R*-tree. We need to check if its current MBR contains the new location or not.

(a) If the new position is still in its current MBR, just modify the position of that object in the leaf node.

(b) If the new position is not in its current MBR, delete it from current node and insert it with its new position.

2. The object is currently in the R*-tree, its new position is not in the R*-tree: Since the R*-tree contains all the slow moving regions, this corresponds to the scenario that the quasi-static object leaves the slow-moving area and moves into a fast-moving area, such as a freeway. The object should be deleted from the R*-tree and inserted into the Q-tree.

3. The object is currently in the Q-tree, and its new position is in the R*-tree: This means that a fast moving object leaves the fast-moving area and moves into a slow-moving area. The object should be deleted from the Q-tree and inserted into the R*-tree.

4. The object is currently in the Q-tree, and its new position is still in the Q-tree: This means that a fast moving object keeps moving in the fast-moving area. Thus, we need to check if its current quadrant contains the new location or not.

(a) If the new position is still in its current quadrant, just modify the position of that object entry.

(b) If the new position is not in its current quadrant, delete it from current node and insert it into the new node.

In [2], it is mentioned that most of the objects are in slow-moving areas and stay in quasi-static states most of the time. It is also reported that the majority of the situations are

expected to be *Case 1.a* above that does not result in a large update overhead. In our proposed system for the assignment problem, this presumption is not always valid since we only keep the vehicles in the Q+Rtree index. In other words, the vehicles in a quasi-static state may not stay in its current MBR for a long period of time. After a call from a person, the requested vehicle begins to move and, then we have to treat that object as a fast moving object to be placed in Q-tree. Thus, we propose a novel algorithm for the update process of Q+Rtree.

The objects in the Q+Rtree index may be in quasi-static mode or moving at high velocities time to time interchangeably. Therefore, we had to revise our update algorithm to get over this problem. The proposed solution is about giving first priority to traversing the Q-tree index when finding the location of the object position to be updated. By this means, we find the probable quadrant of the object in the Q-tree index, and then, link to the leaves in the R*-tree if necessary. The benefit is that we do not make unnecessary traversals over the R*-tree as the Q-tree and R*-tree overlap topographically and there are links between the leaves of both tree indexes.

To reduce update overhead of the system, a quasi-static object sends a location update only when it leaves its current MBR due to the Lazy Update Approach.. On the other hand, a fast moving object have to send a location update only when it passes to a new quadrant or slows down and turns into quasi-static state.

The proposed algorithm for the update process is illustrated in Figure 3.4.

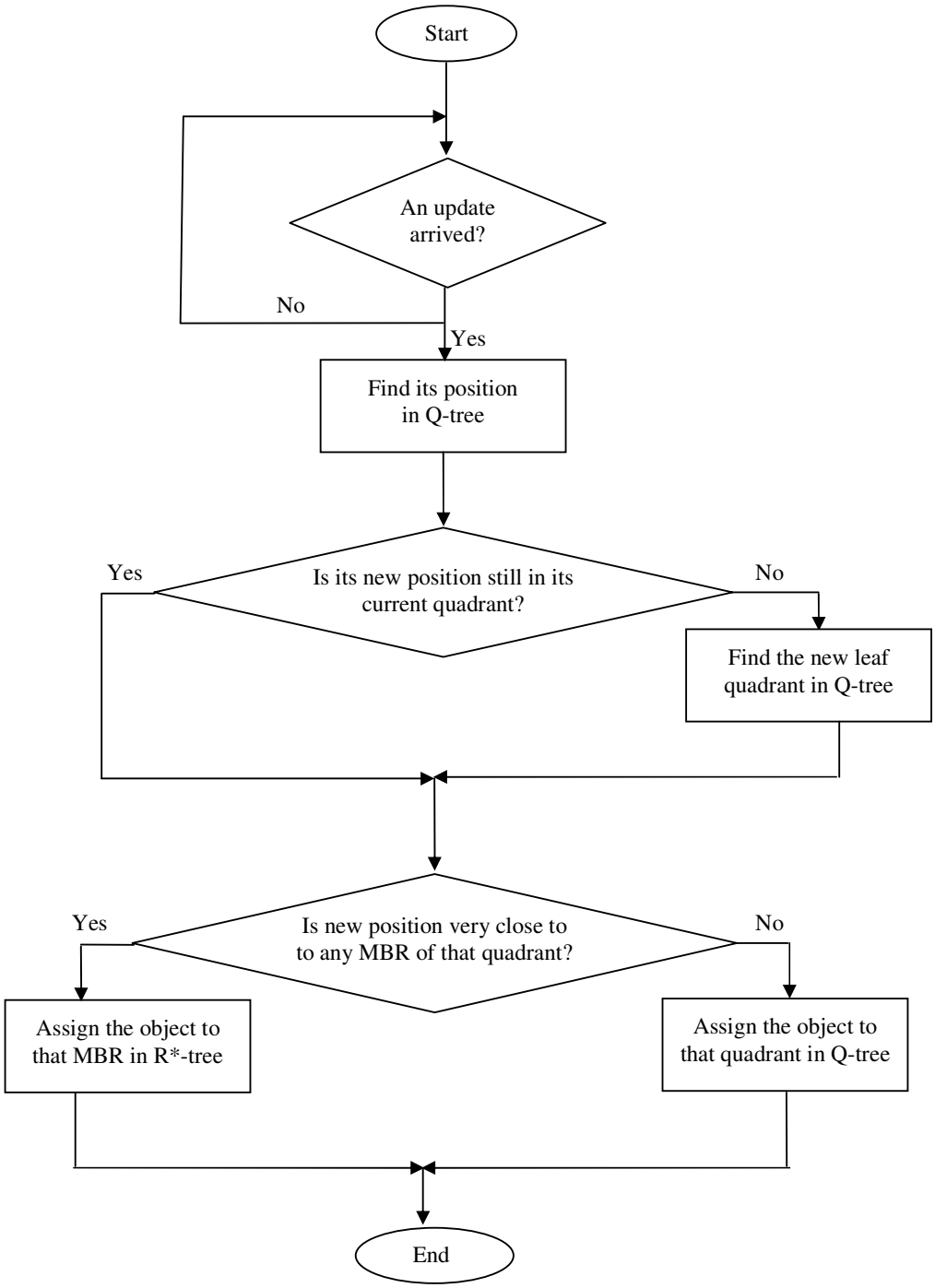


Figure 3.4. Flow diagram of updating Q+Rtree process

3.2.2. Finding Candidate Vehicles From Moving Object Database

To solve the assignment problem, we have to find out the candidate set of vehicles denoted by *V* in the previous sections. Using all the vehicles on the area to form the set *V* would be a performance reducing approach. The elements of set *V* are determined by using all the topographical area covered by our moving object database by choosing reasonable ones to assign. Any vehicle *v* on the area may be a candidate vehicle for a customer *p*. Thus, we cannot presume the right ones at the beginning.

For instance, let a person *p* wants to call a taxi cab. There may be a number of appropriate vehicles (taxi cabs driving on a freeway or waiting in a taxi station) to be assigned around that person's area. To choose the candidate vehicles to be assigned, we begin from searching the Q-tree in our moving object database. We traverse the Q-tree from the root to the leaves, and then link to the leaves of R*-tree by means of pointers.

As it is mentioned in Section 2, the Q-tree index is built as a logical partitioning of the topographical area we are working on. Therefore, we can also find the specific quadrant of a person on the Q-tree. We traverse the Q-tree from the root to the leaves, and find the leaf quadrant that accommodates person *p*. Since the leaf quadrant to be searched for candidate vehicles is found, we add all vehicles in this quadrant to our candidate vehicle set *V*. Then, we keep on searching over R*-tree using the pointers at the leaf level of Q-tree. Thus, we are able to access the MBRs of R*-tree without traversing R*-tree from the root to the leaves.

After all, there is still a probability of not finding any candidate vehicle for an individual person. We have to keep on searching the Q+Rtree in order to find at least one candidate vehicle for that person. To realize this, we jump to the neighbor quadrants of the current leaf quadrant on the Q-tree. Then, we repeat the same operations for all the neighbors. If there is still no candidate vehicles found, that person will unfortunately stay unassigned instead of being assigned to an unacceptably far vehicle. The flow diagram for the process above is in the following Figure 3.5.

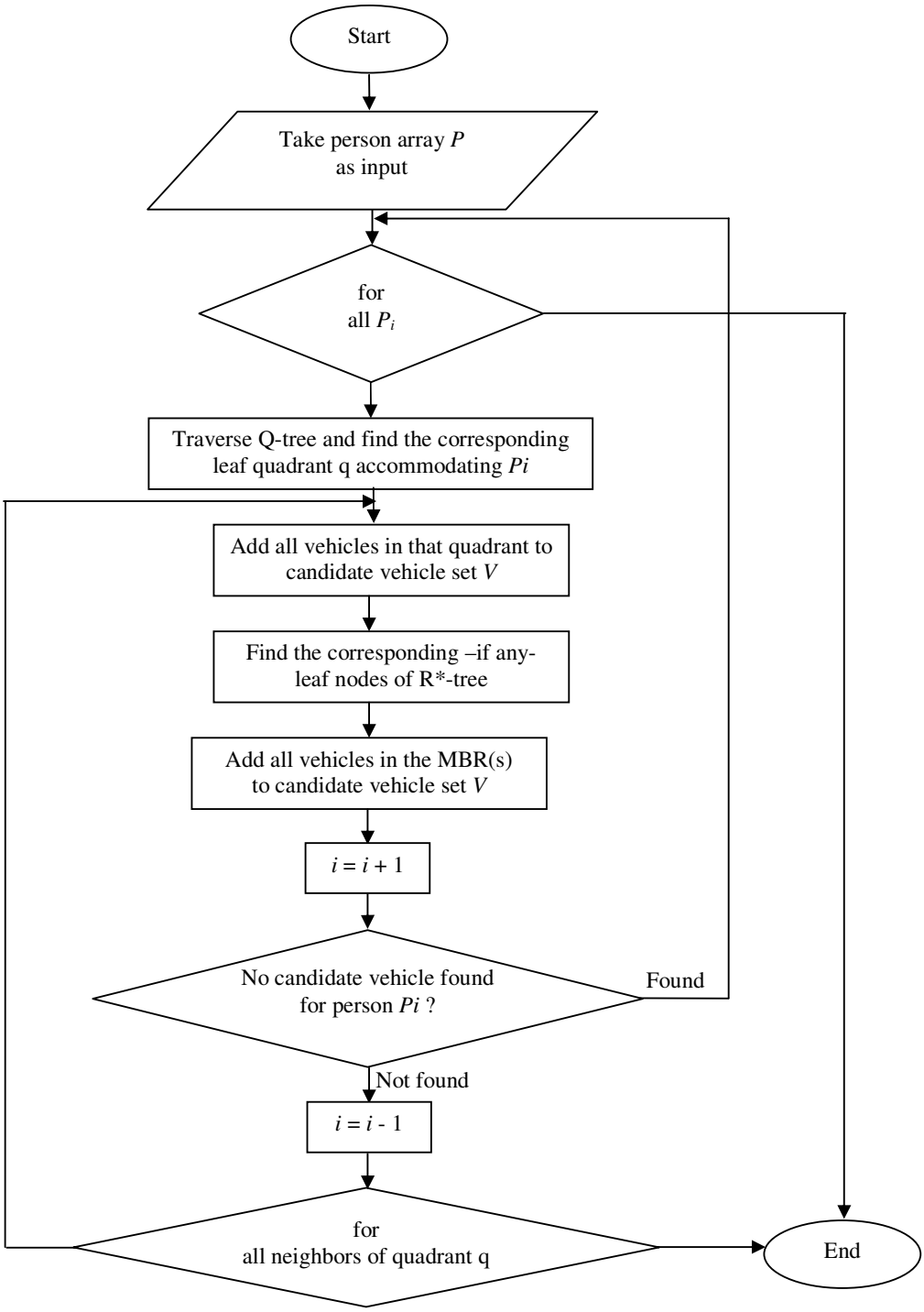


Figure 3.5. Flow diagram of finding candidate vehicles

3.2.3. Cost Computation

After determining the candidate vehicles to be assigned, we have to compute the path costs between each person and its candidate vehicles. This corresponds to computing the weights of the edges, namely *ARCS*, on the bipartite graph of the sets *P* and *V*. As we are working on a road network, it can be said that the cost of a path between a person *p* and a vehicle *v* is equivalent to the travel time between these two points. Furthermore, in a road network, travel time cannot be viewed as just the distance between two points because of some physical limitations, e.g. maximum speed restrictions on a road, narrowness of certain streets, traffic load, and etc.

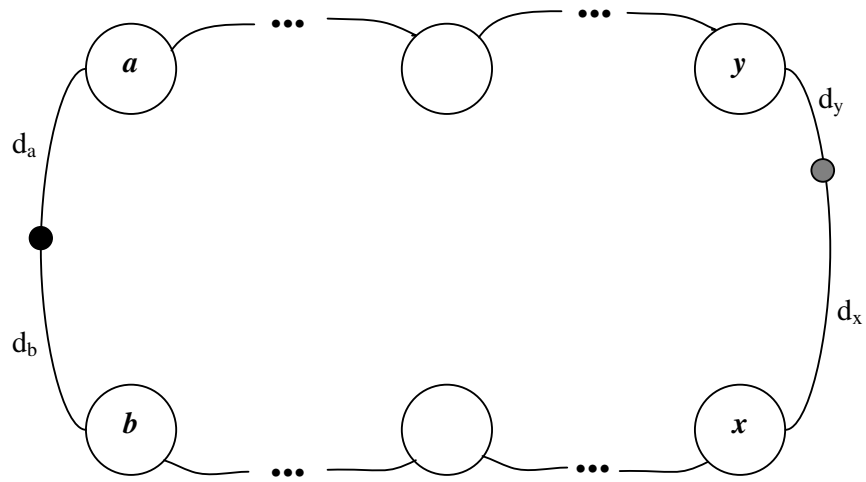
We base our study on these observations and make further observations as follows: Firstly, moving objects with a few exceptions follow certain paths such as a route on a highway system. Such paths on which moving objects move have physical limitations such as the number of lanes and/or route conditions. In addition, based on our everyday experience, we can expect that as the number of vehicles in a section of a route increases, the average speed of the vehicles decreases. This is supported by a report from the Texas Transportation Institute [6]. The report observed that as the daily traffic volume (the average number of vehicles observed in a day at a location) increases, the average speed of vehicles decreases. This leads to an increase in travel time between two points. Consequently, when calculating the path cost between a person *p* and a vehicle *v*, we place higher cost values for such cases.

Initially, we assume that the digital map of the topographical area we are working on is given. Thus, we also have the real distance values between the certain points of the road network. Briefly, the cost of a path (a, b) is determined by using both the real distance value and the average speed of vehicles on that specific path as follows:

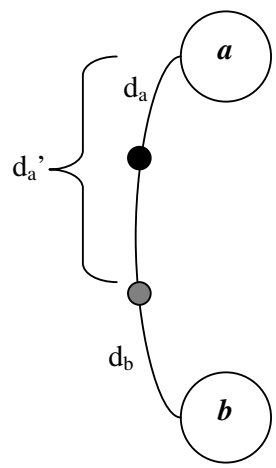
$$Cost_{ab} = (\ \ distance\ of\ (a,b\ )\ )\ \ /\ \ (average\ speed\ on\ (a,b))$$

(Average speed value is determined by using past behavior of vehicles on the related specific path at certain periods of time in a day.)

The difficulty is that with continuously moving data points and alternative paths between the points present, we must continuously recompute the shortest paths utilizing the cost values calculated as above. We foresee that computations for calculating the shortest distances at run-time become too expensive in the mobile setting. Instead, we use an existing shortest path algorithm, Dijkstra's Algorithm [8], before the cost computation process for each arc from *P* to *V*. Pre-computation of the shortest distances between any pair of nodes in the road network provides a noticeable speed up during the cost computation process [7].



(i)



(ii)

Figure 3.6. Computing weights on a road network

(i) *p* and *v* are on different edges

(ii) *p* and *v* are on the same edge on the road network

The Figure 3.6 may be a good example to understand how we compute a weight on the bipartite graph of *P* and *V*. Calculating the weight of an arc between person *p* and vehicle *v* is explained below with the parameters given in the Figure 3.6:

IF *p* and *v* are on different edges on the road network (Figure 3.6 case i)

$D1 = SP_{ax} + d_a + d_x$

$D2 = SP_{ay} + d_a + d_y$

$D3 = SP_{bx} + d_b + d_x$

$D4 = SP_{by} + d_b + d_y$

$COST_{pv} = \min(D1, D2, D3, D4)$

ELSE IF *p* and *v* are on the same edge (Figure 3.6 case ii),

$D1 = | d_a - d_a' |$

$x = d_{ab} - D1$

$D2 = SP_{ax} + x$

$COST_{pv} = \min(D1, D2)$

where,

SP_{ij} : The cost of shortest path between I and j

d_{ab} **:** The cost of edge between a and b

3.2.4. Matching Algorithm

In this thesis, we propose an assignment query for MOD, where all people requesting service at a time are known in advance. This means that the person-set *P* is known at the beginning, and does not change online during the assignment process. The algorithm for the assignment query, *ASSIGN_OBJ*, works offline. We intended to make our offline algorithm's runtime significantly faster than the people's arrival rate. Thus, we approximate the online setting as well (see [37, 38] for online algorithms).

We have to give near real-time responses to user queries to provide an efficient solution. This is why the data objects, vehicles, will be still moving during the assignment job. Unfortunately, finding a minimal-cost matching for a general bipartite graph has time complexity $O(N^3)$ [11, 16, 33, 34]. Thus, we realized that exact solutions having such a time complexity would be infeasible in our moving object database application. Using a faster approximate algorithm for the assignment query seems to be a reasonable choice. A survey of approximation algorithms for combinatorial optimization problems is given in [27, 36].

An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time [18]. For instance, let c be the cost of solution produced by approximate algorithm and c^* be the cost of optimal solution. For our minimal cost bipartite matching problem, we are interested in finding a solution in the set of feasible solutions such that (c / c^*) be as small as possible.

Approximation algorithms for the weighted matching problem have been used in practice already for a long time. This is because many real world problems require graphs of such large sizes that the running time of the fastest available weighted matching algorithm is too costly. On the other hand, approximation algorithms for the weighted matching problem are very fast, and nevertheless produce very good results even if these results are not optimal. Their good running times are one of the main motivations for using them [18]. Furthermore, approximation algorithms proposed for weighted matching in general graphs fits our problem. This is because weighted bipartite matching is a subset of this general problem. Since bipartite graphs are useful for modelling matching problems, bipartite matching may give better results compared to matchings on general graphs.

The quality of an approximation algorithm for the weighted matching problem is measured by its so-called *performance ratio*. An approximation algorithm has a performance ratio of c, if for all graphs it finds a matching with a weight of at least c times the weight of an optimal solution [27].

In literature, we have found several linear time approximation algorithms to solve maximal weighted matching problem. We have chosen the best one and easily adapt it to our problem in order to obtain minimal cost matching.

Preis [19] presented a linear time approximation algorithm for the weighted matching problem with a performance ratio of ½ . He combined the advantages of the the greedy algorithm in Figure 3.7 and the maximal matching algorithm in one algorithm.

```
Greedy Matching ( G=(V: vertices, E: edges), w: E → R+ )
M : = 0
while E ≠ 0 do begin
    let e be the heaviest edge in E
    add e to M
    remove e and all edges incident to e from E
end

where,
    M is a matching in graph G(V,E).
```

Figure 3.7. The greedy algorithm for finding maximum weight matchings

Drake and Hougardy, in [17], proposed another linear time approximation algorithm with a performance ratio of ½ . The main idea is to grow two matchings simultaneously and return the heavier of both as the result. They also find the two matchings in a greedy way. Their algorithm and its analysis are simpler than that of Preis [19].

To the best of our knowledge, the most promising approximation algorithm suitable for MOD is proposed by Drake and Hougardy [18], which runs in linear time and has a performance ratio of ⅔. The main idea of their algorithm is to start with a maximal matching M and to increase its weight by local changes. These local changes which they call short augmentations add in each step at most two new edges to M while up to four edges of M are removed.

We adapted the idea in [18] to our minimal-cost weighted bipartite matching problem. This algorithm has a linear time complexity and gives a good performance ratio of ⅔ that is suitable for our MOD application. As the algorithm in [18] is proposed for the maximal matching, we inverted the weights on our bipartite graph and used these inverted weights as the input of this maximal matching algorithm. If, for instance, we have a cost of 25 on an edge of the bipartite graph, we invert this value as 1/25 and use it in the maximal matching algorithm. This inverted value may be considered as the *profit* of a customer to be assigned to a taxi-cab. Profit values closer to 1 are naturally more suitable for a

customer. Consequently, the algorithm becomes convenient to solve our minimal-cost matching problem by maximizing the total profit.

4. COMPARATIVE PERFORMANCE RESULTS

After presenting the proposed assignment query type and its implementation, we comment on the performance evaluation issues of our thesis in this section. Firstly, we summarize the approaches to increase overall performance of the system. Then, the detailed information about the simulation program is given. At the end, we discuss the comparative performance results to verify that our proposed system meets the needs of users in mobile environment.

We made several performance increasing suggestions for our moving object database. The important ones can be summarized as follows: Firstly, there is always a performance gain when updating and querying the Q+Rtree index. Since we begin searching from the Q-tree, we prevent unnecessary traversals over the R*-tree by means of links between the leaves of both tree structures. Another issue is about reconstructing the Q+Rtree index at certain time periods. For instance, the daily traffic load may change in different hours of a week day. Thus, taxi-cabs may wait for customers in different locations on the road network due to this traffic change. Reconstructing the Q+Rtree in morning and evening hours in a day may lead to a certain performance increase. Likewise, recomputing the shortest paths on the road network points out another performance gain. As it is mentioned before, we calculate shortest distances between two points with considering both real distances and traffic load. Also, as the daily traffic tends to increase in morning and evening hours, we recompute shortest paths to provide a better performance. In addition, using an approximate algorithm to find the shortest paths may fasten our system. This is why we are working with a road network of a whole city that corresponds to a dense graph in memory.

To analyze the performance results of the matching algorithm for the assignment query, we have written a simulation running on a computer having 3 GHz Pentium IV processor and 1024 MB of internal memory. The operating system, on which the simulation works on, is Microsoft Windows XP Professional Version 2002 with Service Pack 2. We have used the C++ programming language for the implementation and our

development environment is Microsoft Visual C++ 6.0. We have implemented the matching algorithm in [18], which finds an approximate maximal weighted matching in linear time. We have adapted this algorithm to our special case in order to find a minimal matching. To give comparative performance results, we have also implemented the Hungarian algorithm in [16] and a greedy algorithm. Hungarian algorithm gives an exact solution with a time complexity of $O(N^3)$. Greedy algorithm produces an approximate result for the minimal weighted bipartite matching.

4.1. Details of the Simulation Program

The simulation program has been written by the C++ programming language on MS Visual C++ 6.0 development environment. The program may be run for different cases by changing the parameters, e.g. number of people, number of vehicles, random seed for cost values. These parameters are used to evaluate the performance of the three matching algorithms used. A sample run of the simulation program for 10 people and 10 vehicles is illustrated below (using the greedy matching algorithm):

```
"C:\Documents and Settings\Ali Riza.ARK-PC.000\Desktop\tez-perf-analysis\...
Number of people: 10
Number of vehicles: 10

Cost matrix:
 [   72    71    61    38    67    83    96    41    10     3 ]
 [    5    57    65     9    67    47    30    90    53     4 ]
 [   97    45    98    30    82    90    88    16    71    96 ]
 [   59    67    42    16    20    92    11    11     5    68 ]
 [   85    81    67    10    95    43    23    68    52    98 ]
 [   92     9    84    85    86    41     9    57    67    65 ]
 [   84    95    98    24    46    22     1    98    41    17 ]
 [   41    93    56    91     1    41    81    29    79    72 ]
 [   28    20    22    38    41    96    89    65    53    44 ]
 [   19    32    31    71    99    65    36    58    53   100 ]

Assignment:
 [    0     0     0     0     0     0     0     0     0     1 ]
 [    1     0     0     0     0     0     0     0     0     0 ]
 [    0     0     0     0     0     0     0     1     0     0 ]
 [    0     0     0     0     0     0     0     0     1     0 ]
 [    0     0     0     1     0     0     0     0     0     0 ]
 [    0     0     0     0     0     0     1     0     0     0 ]
 [    0     0     0     0     0     1     0     0     0     0 ]
 [    0     0     0     0     1     0     0     0     0     0 ]
 [    0     1     0     0     0     0     0     0     0     0 ]
 [    0     0     1     0     0     0     0     0     0     0 ]


Total cost is 122.00
Press any key to continue
```

Figure 4.1. A sample run of the simulation program

The output of the simulation program may be also written to a file chosen by the user. This is because the number of people and vehicles may vary from 100 to 100,000 for different cases. In order to evaluate the performance of the matching algorithms, the choice of random seed is important. As the program produces random cost values at each run, the same random cost values should be used for each matching algorithm for the corresponding case. For instance, the cost matrix in Figure 4.1 should be also used for Hungarian algorithm and DH algorithm in [18] without any change. By substituting the same random seed in the program, we are able to produce the same cost matrix for each matching algorithm at each run of a chosen case.

4.2. Methodology

As iti is mentioned in the previous section, we have chosen the DH (Drake-Hougardy, [18]) algorithm for the matching algorithm in our thesis. We have also employed the Hungarian algorithm in [11] and a greedy approach to compare with the performance results of the DH algorithm. We have worked on 100 to 100,000 people and vehicles in order to demonstrate the scalability.

We have focused on two performance metrics, which are execution time and total cost. Execution time of the matching algorithm is an important issue in the mobile setting. While the program runs, people and vehicles tends to move physically. After the matching is found by the algorithm, these people and vehicles may be too far away from their old positions. The algorithm should work as fast as possible to meet the needs of a user. Thus, we compare the execution times of the three algorithms whether they are satisfactory or not in the mobile setting. Another important performance issue is the total cost after the assignment of people and vehicles. As DH algorithm performs an approximate weighted matching, the total cost of the assignment should be within reasonable boundaries.

On the other hand, we have tested the DH algorithm with different type cost matrices and different number of people and vehicles. This means that the cost matrix may have values from a large interval or small interval. Values from a large interval correspond to the case that the majority of people and vehicles distributed relatively far away from each other in the geographical area. Values from a small interval correspond to the case where

people and vehicles are located relatively close to each other. In addition, the number of people and vehicles to be assigned may not be equal. To evaluate the performance of the three algorithm for this case, we have run the simulation where **M (number of people) = 2*N (number of vehicles)**, and vice versa.

Lastly, we focused on the DH algorithm's sensitivity to the number of vehicles. We observed the execution time and total cost values of the DH algorithm, where the number of people is constant and the number of vehicles increases step by step. We employed four cases for the constant value: 100, 1,000, 10,000 and 100,000 people. We investigated the change of execution time and total cost values for these cases in order to simulate the behaviour of the proposed system in different situations. This is why the daily traffic load in a road network may differ in different hours in a day, or different days in a week.

4.3. Performance Results

In this section, time complexity and efficiency results of the chosen matching algorithms are given for the cases mentioned above. Firstly, execution time and total cost changes are illustrated for each algorithm. Results for randomly chosen cost matrices and different number of people and vehicles are given. At the end, the DH algorithm's sensitivity to the number of vehicles is discussed with resulting graphs.

4.3.1. Results for Randomly Chosen Costs

According to the physical distribution of people and vehicles on the geographical area, the travel time between a vehicle v_i and a person p_j may change. If the vehicles and people are distributed relatively far away from each other, the cost of travelling from v_i to p_j will be costly in terms of travel time. Thus, the total cost of the assignment will increase. In the other case, people and vehicles may be located relatively near to each other. This means that the travel time from a vehicle v_i to a person p_j will decrease considerably. Also, the execution times of the matching algorithms are expected to differ for both cases. In this section, the resulting graphs from our simulation program are given for the two cases.

4.3.1.1. Results for Relatively Large Cost Values

Large cost values correspond to people and vehicles distant from each other on the geographical area. In other words, most of the candidate vehicles for a person will be far away. The Figure 4.2 illustrates the execution times of DH, Hungarian, and greey algorithms. The number of people and vehicles are equal to each other in each step. For instance, the value 50,000 in the x-axis corresponds to 50,000 people and 50,000 vehicles are included in the simulation.

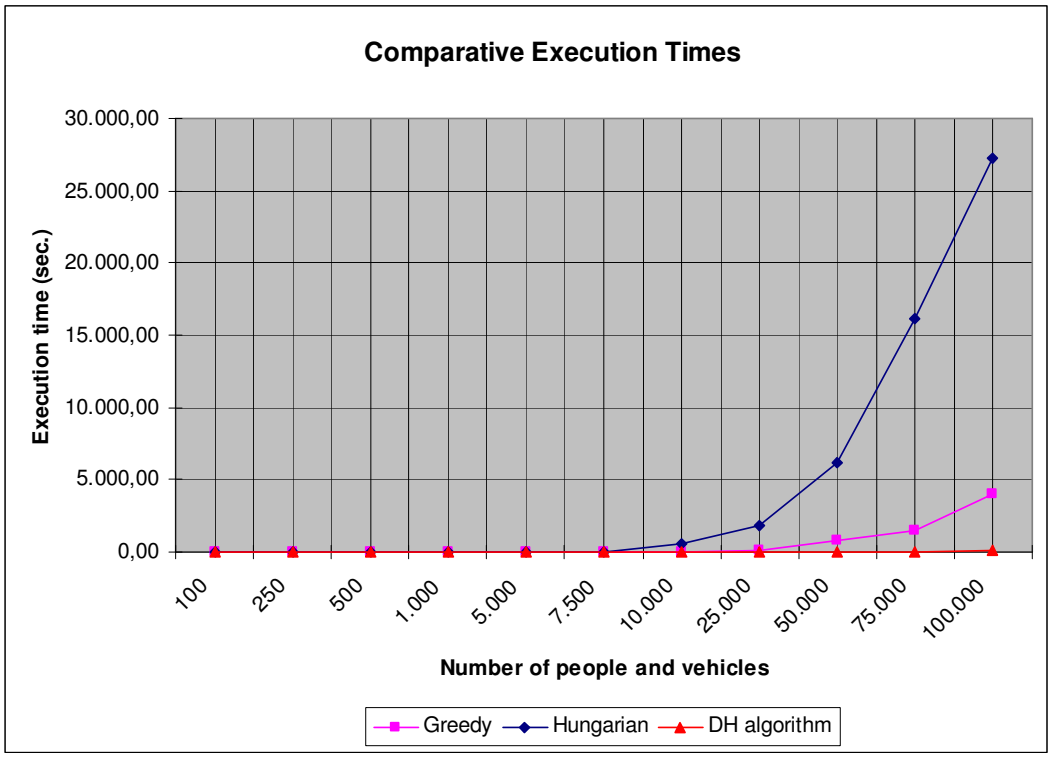


Figure 4.2. Time complexity of matching algorithms for large cost values

It is obvious that the execution time of Hungarian algorithm is the worse, as its time complexity tends to $O(N^3)$. DH algorithm outperforms the other ones as the number of people and vehicles approaches to 100,000. Thus, DH algorithm becomes convenient for a mobile environment as it has a linear-time complexity. Execution time of DH algorithm is shown in Figure 4.3 for the same case:

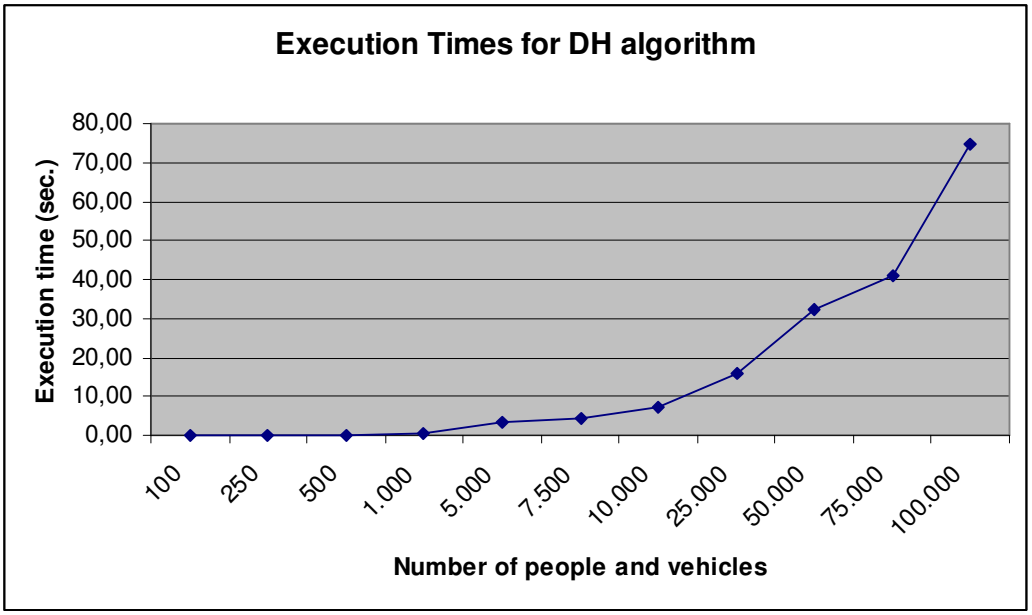


Figure 4.3. Time complexity of DH algorithm for large cost values

After comparing the execution times of the matching algorithms, changes in total costs are evaluated. As Hungarian algorithm performs an exact solution for the problem, minimal total cost value is guaranteed. It is also known that DH algorithm has a performance ratio of ⅔. Thus, we expect that DH algorithm will produce %33 larger total cost value when compared to Hungarian algorithm. Greedy algorithm produces total cost values relatively close to the Hungarian algorithm. Resulting graph from our simulation program is given in Figure 4.4.

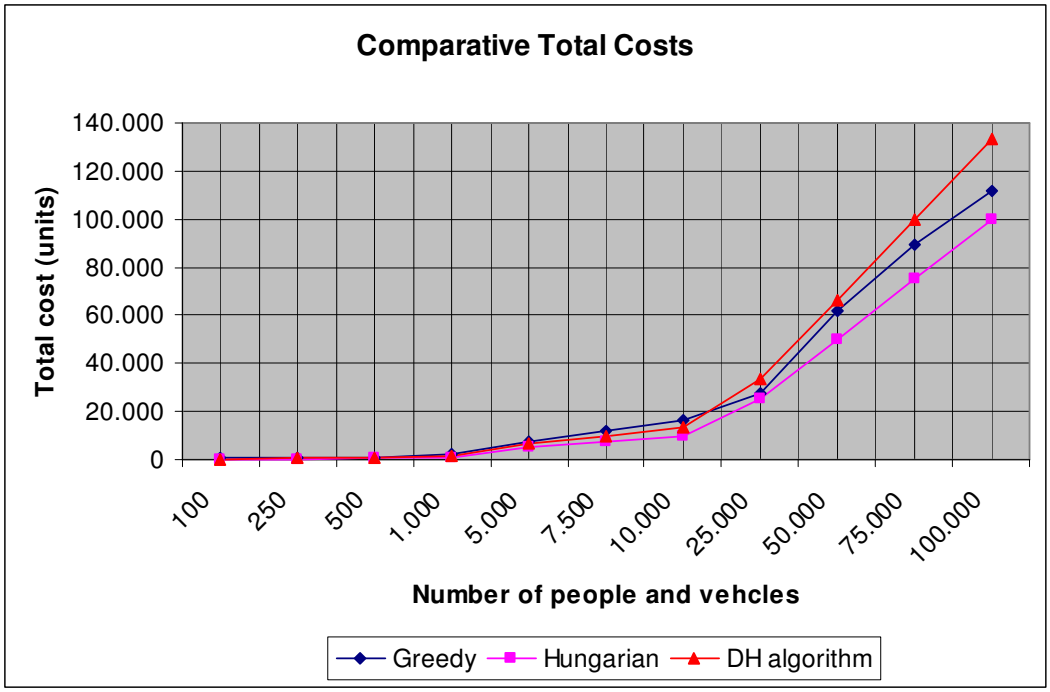


Figure 4.4. Total costs produced by the matching algorithms for large cost values

4.3.1.2. Results for Relatively Close Cost Values

If people and vehicles are located close to each other on the geographical area, the travel time between a person and its candidate vehicles will be relatively small. This situation affects the total cost of the assignment and execution time of the algorithms. The results are shown in Figure 4.5 and Figure 4.6.

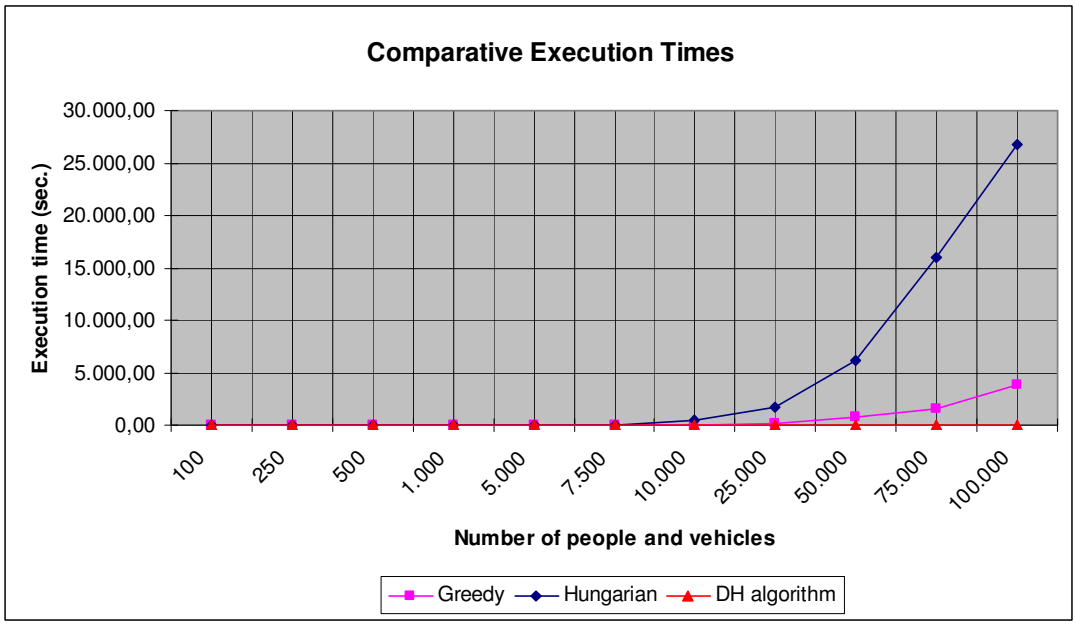


Figure 4.5. Time complexity of matching algorithms for close cost values

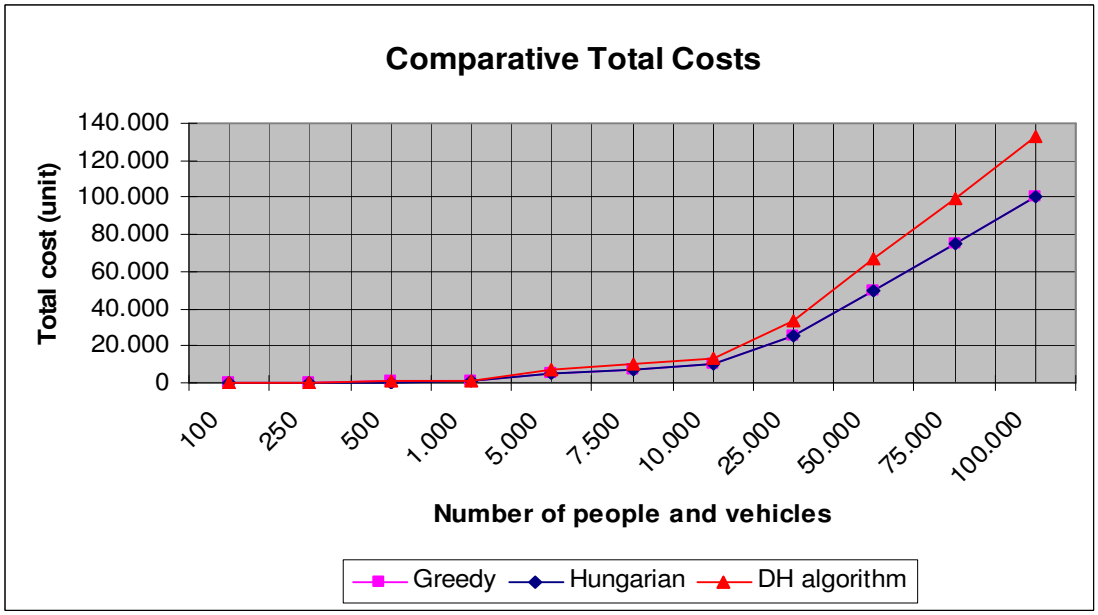


Figure 4.6. Total costs produced by the matching algorithms for close cost values

In Figure 4.5, it is seen that the execution times of the matching algorithms decreases at a small rate when compared to the case in Section 4.3.1.1. In Figure 4.6, total cost values produced by greedy algorithms tends to approach Hungarian algorithm differently from DH algorithm. DH algorithm still produces approximately 33% worse values than

Hungarian algorithm. It can be said that greedy algorithm may be convenient for small cities, where people and vehicles are distributed not too far away from each other. On the other hand, DH algorithm outperforms greedy algorithm in execution times. For big cities, DH algorithm is the best one to choose when the number of people and vehicles increase although they are located closer or far away from each other. It runs within approximately 70 seconds for about 100,000 vehicles and people. This is acceptable in the mobile setting.

4.3.2. Results for Different Number of People and Vehicles

The number of people and vehicles to be assigned may change in a day. Number of people may be larger than candidate vehicles to be assigned, or vice versa. In this section, behaviour of the three matching algorithms is observed when the number of people and vehicles are different. In the first case, the number of people is twice the number of vehicles. In the second case, the number of vehicles is twice the number of vehicles. These cases are chosen to observe the average cases for the number of people and vehicle.

4.3.2.1. Results for Higher Number of People

If the number of available vehicles are too few at a time, then the number of candidate vehicles to be assigned will decrease. As an average case, candidate vehicles may be half of people need to be assigned. Different cases studied in our simulation are given in Table 4.1. Comparative execution times of DH, Hungarian and greedy algorithms are shown in Figure 4.7 regarding the cases in Table 4.1:

Table 4.1. Cases for higher number of people

Case	Number Of People	Number Of Vehicles
1	100	50
2	250	125
3	500	250
4	1.000	500
5	5.000	2.500
6	7.500	3.750
7	10.000	5.000
8	25.000	12.500
9	50.000	25.000
10	75.000	37.500
11	100.000	50.000

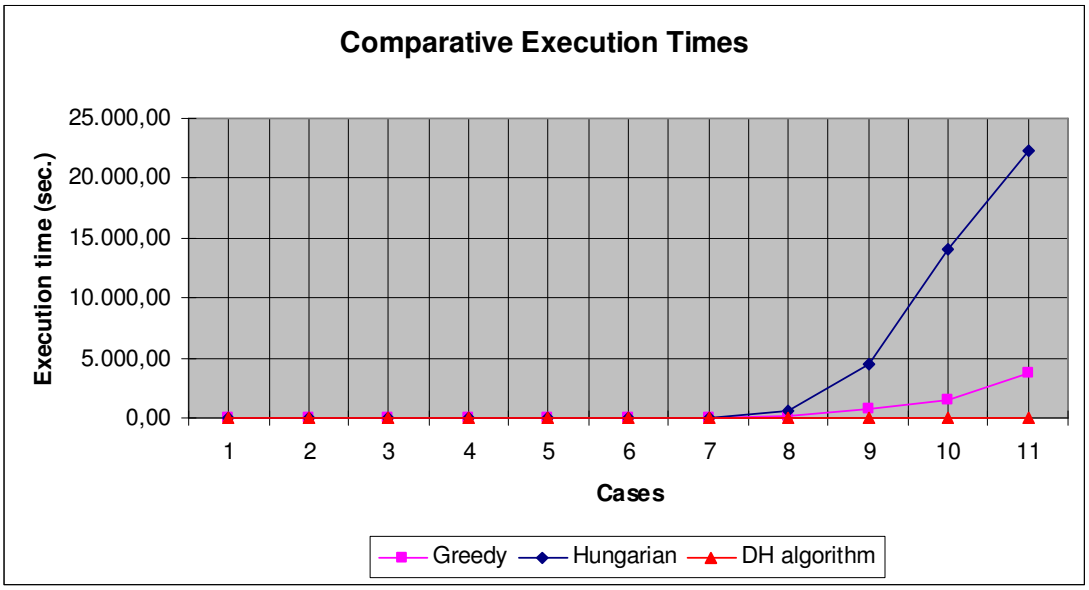


Figure 4.7. Time complexity of matching algorithms for cases in Table 4.1

As it is seen, execution times of the three matching algorithms decrease when compared with the cases in Section 4.3.1 and 4.3.2. This is why there are less number of vehicles to be assigned. Half of the people are not assigned to any vehicle. This means that the number of comparisons to choose candidate vehicles decreases to half. DH algorithm still outperforms Hungarian and greedy algorithms in execution time.

Total costs produced by the three matching algorithms are illustrated in Figure 4.8. Until the case 4 in Table 4.1, DH approximation algorithm is better than greedy algorithm. Greedy algorithm produces more close values to Hungarian algorithm for large number of people and vehicles. As DH algorithm has minimum execution times compared to the other two algorithms, it still seems the most promising one for our mobile environment.

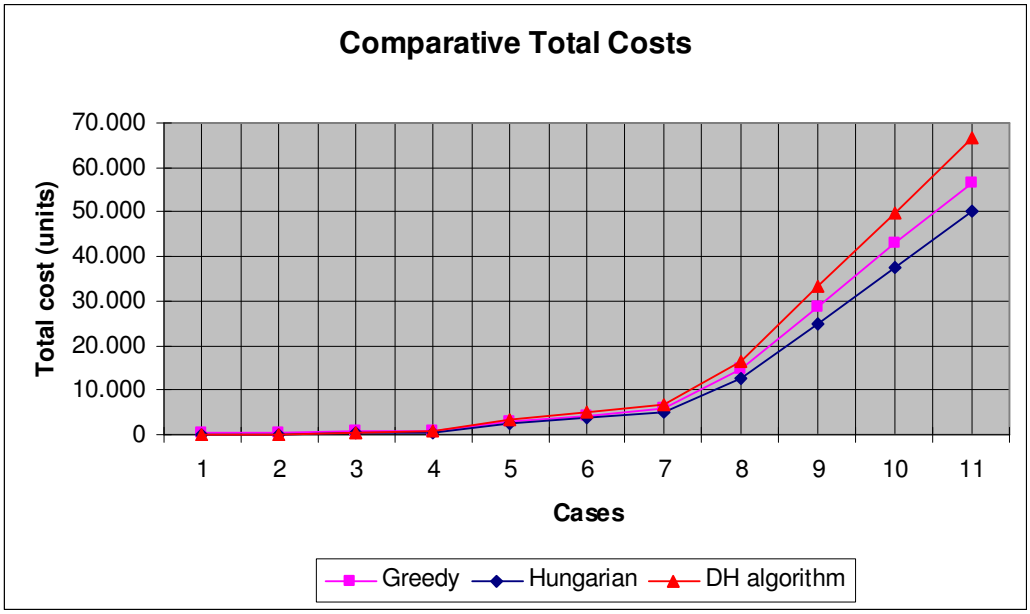


Figure 4.8. Total costs produced by the matching algorithms for cases in Table 4.1

4.3.2.2. Results for Higher Number of Vehicles

If the number of people is less than the number of vehicles in the system, there may be more vehicles to be chosen a candidate. It is expected that the execution times of the matching algorithms will increase considerably. As the number of candidate vehicles tend to be more than the number of people, comparisons to find the final assignment will icrease, too.

Table 4.2 lists the cases which are considered in the simulation. Figure 4.9 and 4.10 are the resulting graphs showing the execution times and total costs produced by the three matching algorithms, in turn.

Table 4.2. Cases for higher number of vehicles

Case	Number Of People	Number Of Vehicles
1	50	100
2	125	250
3	250	500
4	500	1.000
5	2.500	5.000
6	3.750	7.500
7	5.000	10.000
8	12.500	25.000
9	25.000	50.000
10	37.500	75.000
11	50.000	100.000

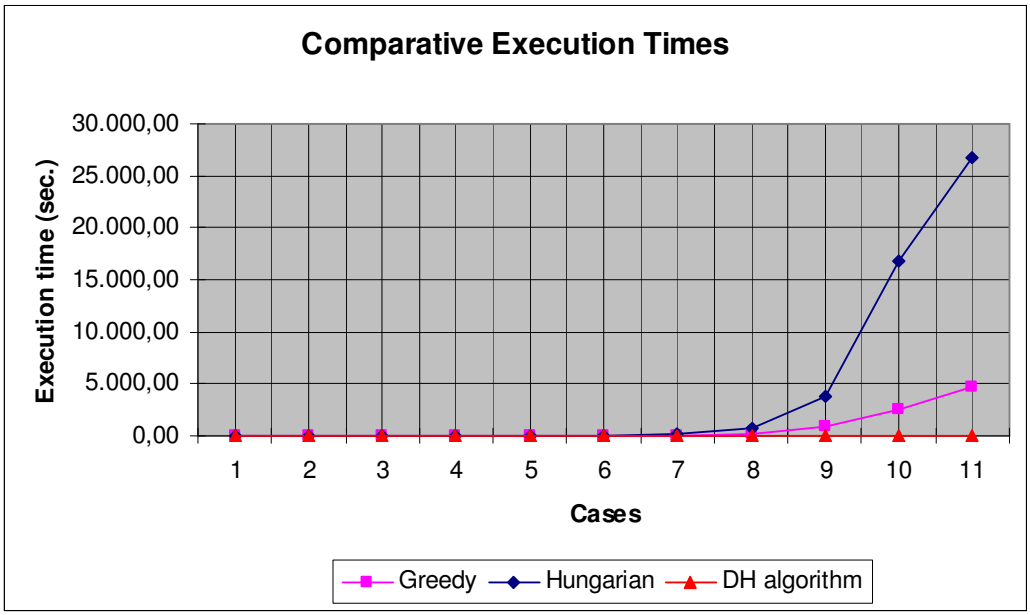


Figure 4.9. Time complexity of matching algorithms for cases in Table 4.2

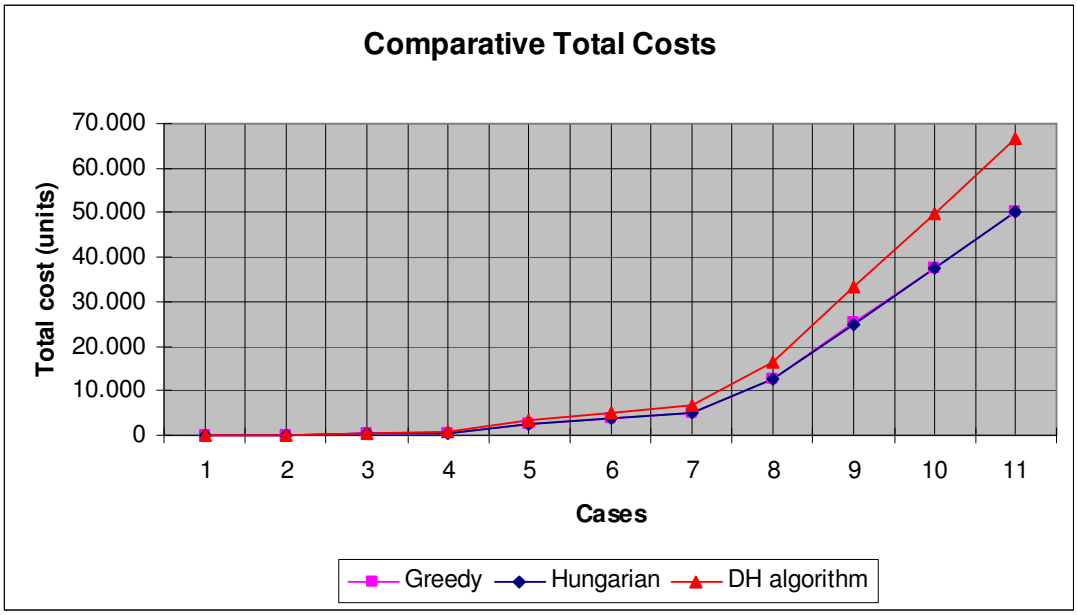


Figure 4.10. Total costs produced by the matching algorithms for cases in Table 4.2

As it is expected, execution times of the three algorithms tend to increase. In Figure 4.10, it is seen that greedy algorithm produces almost the same total cost values with the Hungarian algorithm. Until case 6 in Table 4.2, greedy algorithm and DH algorithm has near execution times. We can conclude that greedy algorithm may be convenient until 3,750 people and 7,500 vehicles. After case 6, DH algorithm outperforms greedy algorithm in execution time, and becomes more convenient for users in a mobile environment.

4.3.3. Results for DH Algorithm's Sensitivity to Number of Vehicles

In the previous sections, we have shown that DH algorithm is the most promising matching algorithm for our mobile environment application. In this section, DH algorithm's sensitivity to the number of vehicles in the system is discussed. We keep the number of people at a constant value (each color represents a constant number of people) while the number of vehicles is increased from 100 to 100,000. To evaluate the changes of execution time and total cost, we employed four constant values for the number of people: 100, 1,000, 10,000 and 100,000.

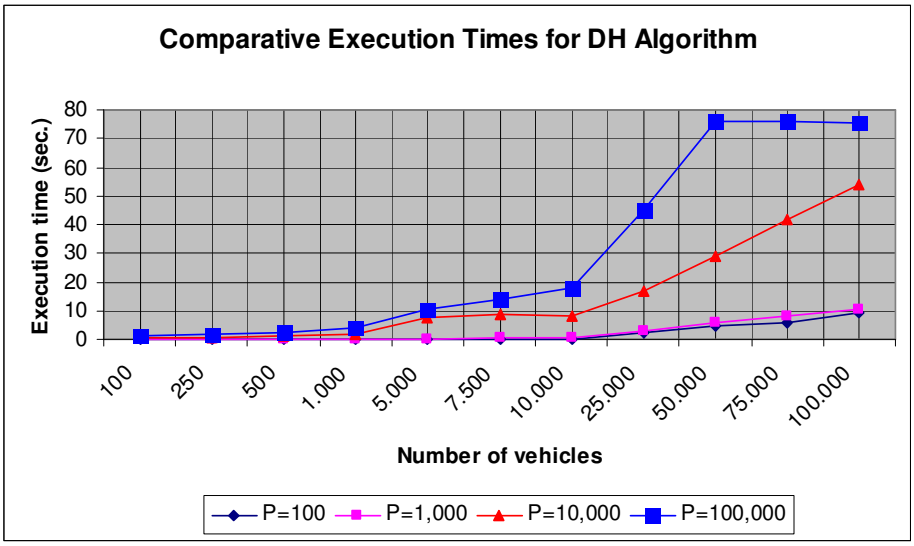


Figure 4.11. Comparative execution times of DH algoritm for constant number of people

In Figure 4.11, P is used for the number of people. For P=100 and P=1,000, DH algorithm has apprioximately same execution times. For P=10,000 and P=100,000, DH algorithm becomes sensitive to the number of vehicles and execution time of the algorithm increases seriously.

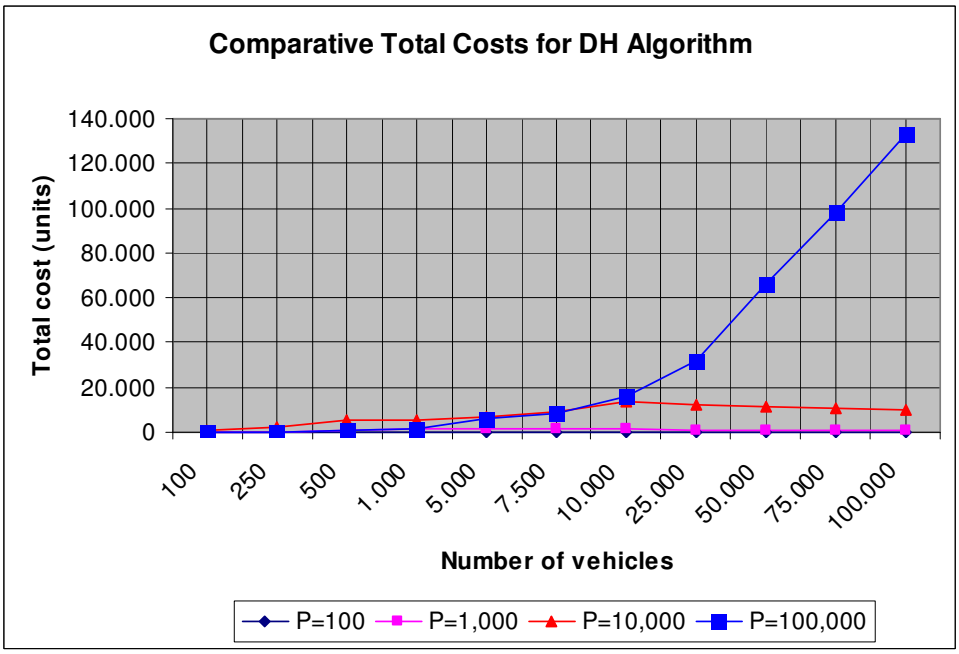


Figure 4.12. Comparative total costs produced by DH algorithm for constant number of people

In Figure 4.12, total costs produced by DH algorithm for different constant values of number of people are shown. For P=100 and P=1,000, DH algorithm produces approximately same results. For P=10,000 and P=100,000, we can see that total cost increases considerably and so, the algorithm becomes sensitive to the number of vehicles.

In Figure 4.13, total cost changes of DH algorithm for P=10,000 above is given.

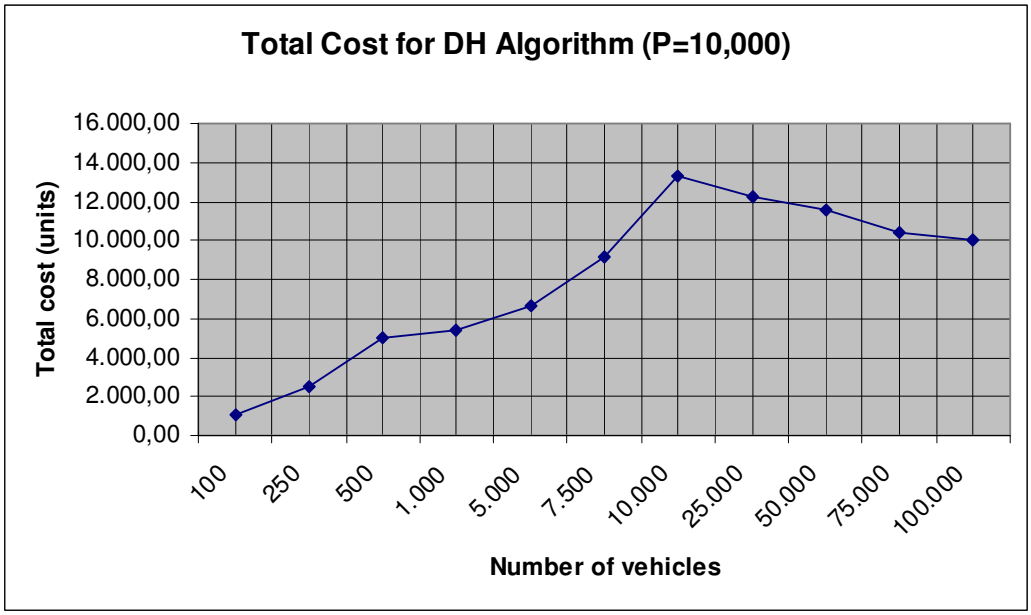


Figure 4.13. Total costs produced by DH algorithm for 10,000 people

In the Figure 4.13 above, total cost values tend to increase until the number of vehicles reaches 10,000. After the number of vehicles increase from 10,000 to 100,000, total cost values begin to decrease. This can be explained as follows: If the number of vehicles becomes greater than the number of vehicles, then the number of candidate vehicles is to increase. As the number of candidate vehicle increases, the algorithm has more alternative cost values to find a more efficient solution.

5. CONCLUSION

Considering the great application demands for moving object databases, to be able to efficiently locate and answer queries related to the position of these objects in time became very important. More specifically, modern applications, such as Mobile Computing and Geographic Information Systems, make the development of research within this field inevitable. In this thesis study, we proposed a novel operator as a query type for moving object databases to solve the well-known assignment problem that is also called weighted bipartite matching. We also described the details of a possible implementation of this query type. We focused on the case that query objects are stationary and data objects are mobile. Our goal was to find a matching that minimizes the total cost.

After realizing that classical solutions for the minimal-cost bipartite matching tends to have a time complexity of $O(N^3)$, we decided to use a linear time algorithm to implement the assignment query. We have chosen the best known approximation algorithm proposed by Drake and Hougardy in [18], which runs in linear time and has a performance ratio of $2/3$. Since this algorithm was proposed for maximal weighted matching, we had to adapt this algorithm to our minimal cost weighted matching problem. During the implementation of the assignment query, we needed to calculate shortest paths between data points for cost computation module. As calculating the shortest paths in real time is too expensive in a MOD application, we chose to utilize pre-computed shortest paths, and then just use them in the algorithm.

Furthermore, we presented a combined tree index structure, called Q+Rtree, to efficiently process the location updates coming from data objects. We also utilized the Q+Rtree structure in the algorithm of the assignment query in order to find the candidate vehicles (data objects) to be assigned as fast as possible. Consequently, we achieved in defining an assignment query type for MOD by producing the result within an acceptable period of time in the mobile setting.

This thesis is an initial study to define a new operator as a query type for moving object databases. The matching algorithm to assign the data objects and query objects

could be improved to find an optimal solution. As we are working in mobile environment, obligation of providing a resonable response time should not be neglected.

APPENDIX A: PARTITIONING A REGION WITH A Q-TREE INDEX

Q-tree index separates the space into four sections in each step.

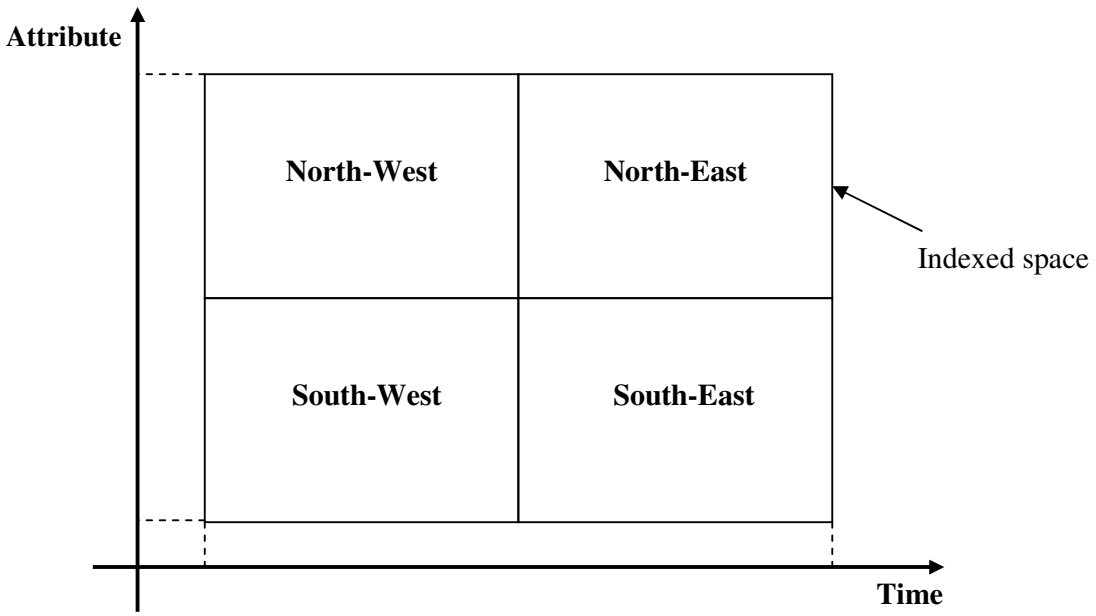
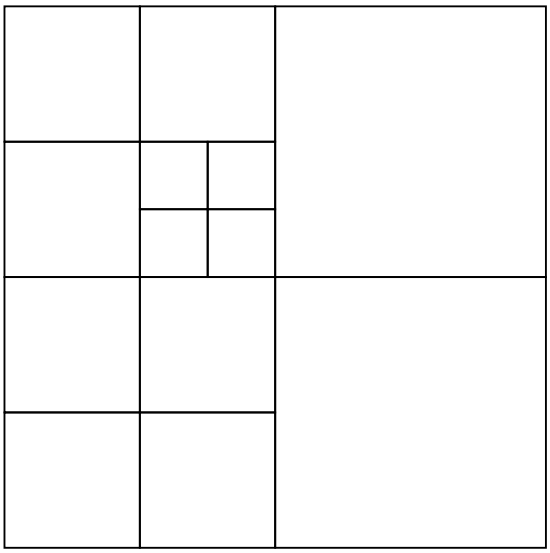


Figure A.1. Indexing space with a Q-tree index

APPENDIX B: REPRESENTING A REGION WITH A Q-TREE INDEX

Q-tree representation of a geographical region is illustrated below:



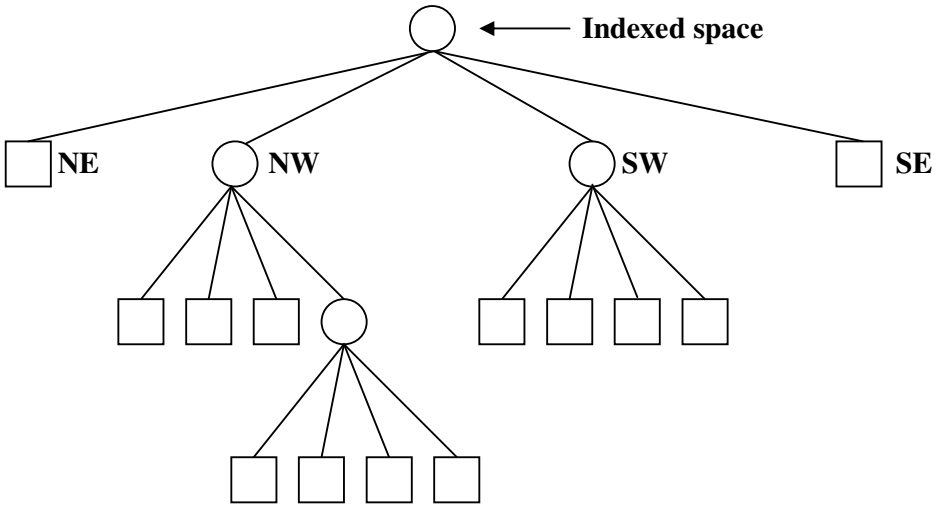


Figure A.2. Q-tree representation of a geographical region

APPENDIX C: OVERVIEW OF THE WHOLE SYSTEM

The following figure is overview of the whole system to process the proposed assignment query. Mobile users (e.g. people and vehicles) send location updates to MOD by receiving their current position from GPS. Mobile Service Provider is used by mobile users to request the service from our application. It provides an interface between mobile users and application. Our assignment query application also responds the assignment result throughout Mobile Service Provider.

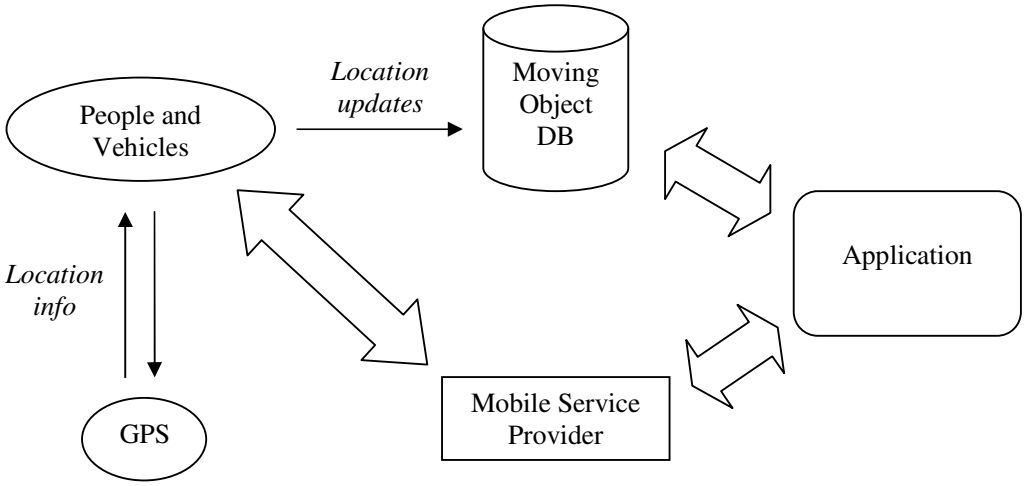


Figure A.3. Overview of the whole system

REFERENCES

1. Cheng, R., Y. Xia, S. Prabhakar and R. Shah, "Change Tolerant Indexing for Constantly Evolving Data," *icde*, pp. 391-402, 21st International Conference on Data Engineering (ICDE'05), 2005.

2. Xia, Y. and S. Prabhakar, "Q+R-tree: Efficient Indexing for Moving Object Databases", Proceedings 8th International Conference on Database Systems for Advanced Applications (DASFAA'03), pp.175-182, Kyoto, Japan, 2003.

3. Katayama, N. and S. Satoh, "The R*-tree: An efficient and robust access method for points and rectangles", In Proc. Of SIGMOD, 1997.

4. Kwon, D. and S. Lee "Indexing the Current Positions of Moving objects Using the Lazy Update R-tree", Third International Conference on Mobile Data Management, January, Singapore, p. 113, 2002.

5. Tayeb, J., O. Ulusoy and O. Wolfson, "A Quadtree-Based Dynamic Attribute Indexing Method", The Computer Journal, Vol. 41, No. 3, pp. 185-200, 1998.

6. Kyoung-Sook K., H. So-Young and L. Ki-Joune, "Arrival Time Dependent Shortest Path by On-Road Routing in Mobile Ad-Hoc Network", W2GIS 2004, pp. 242-253, 2004.

7. Jensen, C. S., J. Kolarvr, T. B. Pedersen and I. Timko, "Nearest Neighbor Queries in Road Networks", In ACMGIS 2003, New Orleans, Louisiana, USA., 2003.

8. Dijkstra, E. W., A note on two problems in connexion with graphs. Numer. Math. 1:269-71, 1959.

9. Harvey, N., R. Ladner, L. Lov'asz and T. Tamir, "Semi-matchings for Bipartite Graphs and Load Balancing", Proc. 8th WADS, 2003, pp. 294-306, 2003.

10. Saltenis, S., C. S. Jensen, S. T. Leutenegger and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects", ACM SIGMOD International Conference on Management of Data, pp. 331-342, 2003.

11. Buss, S. R. and P. N. Yianilos, "Linear and O(n log n) Time Minimum-Cost Matching Algorithms for Quasi-Convex Tours", SIAM J. Of Computing, 27(1):170–201, 1998.

12. Lawler, E., Combinatorial Optimization: Networks and Matroids, Holt Rinehart and Winston, 1976.

13. Roussopoulos, N., S. Kelley and F. Vincent, "Nearest neighbor queries", Proceedings of ACM Sigmod , May 1995.

14. Cai, Y., K. A. Hua and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects", Mobile Data Management, MDM, 2004.

15. Guttman, A., "R-trees: A dynamic index structure for spatial searching", Proc. of the ACM SIGMOD Int'l. Conf., 1984.

16. Kuhn, H. W., The Hungarian method for the assignment problem, Naval Research Logistics Quarterly 2, pp. 83-97, 1955.

17. Drake, D. E. and S. Hougardy, "A Simple Approximation Algorithm for the Weighted Matching Problem", Information Processing Letters 85, pp. 211-213, 2003.

18. Vinkemeier, D. E. D. and S. Hougardy, "A Linear Time Approximation Algorithm for Weighted Matchings in Graphs", ACM Transactions on Algorithms 1, 2005.

19. Preis, R., "Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs", Symposium on Theoretical Aspects of Computer Science (STACS) 1999, C. Meinel, S. Tison (eds.), LNCS 1563, Springer 1999, pp. 259-269.

20. Agarwal, P. K., L. Arge and J. Erickson, “Indexing Moving Points”, Proc. of the PODS Conf., 2000.

21. Wolfson, O., A. P. Sistla, S. Chamberlain and Y. Yesha, “Updating and Querying Databases that Track Mobile Units”, Distributed and Parallel Databases 7(3): pp. 257–387, 1999.

22. Kollios, G., D. Gunopulos and V. J. Tsotras. “On Indexing Mobile Objects”, Proc. of the PODS Conf., pp. 261–272, 1999.

23. Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, ISBN 0-262-03293-7, Section 26.3: Maximum bipartite matching, pp. 664–669, 2001.

24. Karp, R., U.Vazirani and V.Vazirani. "An Optimal Algorithm for On-Line Bipartite Matching ", Proceedings of the 22nd ACM STOC, pp. 352-358, 1990.

25. Gunther, O. and J. Bilmes, "Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation," IEEE Transactions on Knowledge and Data Engineering, vol. 03, no. 3, pp. 342-356, Sept., 1991.

26. R-tree, Wikipedia, http://en.wikipedia.org/wiki/R-tree, 2006.

27. Shmoys, D. B., Computing near-optimal solutions to combinatorial optimization problems, Technical report, Ithaca, NY 14853, 1995.

28. Benetis, R., C. Jensen., G. Karciauskas and S. Saltenis, “Nearest Neighbor and Reverse Neighbor Queries for Moving Objects”, IDEAS, 2002.

29. Marsit, N., A. Hameurlain, Z. Mammeri and F. Morvan, "Query Processing in Mobile Environments: A Survey and Open Problems," dfma, pp. 150-157, First International

Conference on Distributed Frameworks for Multimedia Applications (DFMA'05), 2005.

30. Goldberg, A. V. and R. E. Tarjan, A new approach to the maximum-flow problem, Journal of the Association for Computing Machinery, 35(4):921--940, October 1988.

31. Arge, L., K. H. Hinrichs, J. Vahrenhold and J. S. Vitter, “Efficient bulk operations on dynamic r-trees”, Algorithmica, 33(1):104–128, May 2002.

32. Pfoser, D., C. S. Jensen, and Y. Theodoridis, Novel approaches in query processing for moving objects. Proceedings of the 26th International Conference on Very Large Databases(VLDB), September 2000.

33. Kuhn, H. W., "Variants of the Hungarian method for assignment problems", Naval Research Logistic Quarterly, 3: 253-258, 1956.

34. Munkres, J., "Algorithms for the Assignment and Transportation Problems", Journal of the Society of Industrial and Applied Mathematics, 5(1):32-38, March 1957.

35. Ahuja, R., T. Magnanti and J. Orlin, Network Flows, Prentice Hall, 1993.

36. Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, ISBN 0-262-03293-7. Chapter 35: Approximation Algorithms, pp.1022–1056, 2001.

37. Mehta, A., A. Saberi and U. Vazirani, “Adwords and Generalized Online Matching”, FOCS, 2005.

38. Borodin, A. and R. El-Yaniv, Online Computation and Competitive Analysis, Cambridge University Press, 1998.