ALGORITHMS FOR MULTI-POINT RANGE QUERY AND REVERSE NEAREST NEIGHBOUR SEARCH

NG HOONG KEE
(M. IT, UKM)
(B. IT, USQ)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgements

I would like to take this opportunity to extend my sincerest, heartfelt gratitude to the two great gentlemen of my life, my research supervisor Associate Professor Dr. Leong Hon Wai and my father Ng Hock Wai. They provided great and undying support while I was pursuing this degree. No words of thanks in this world can express enough how I feel. To Prof. Leong, I thank you for being my bright guiding star and a source of inspiration, particularly the invaluable advice and teachings. I cherish all the memories that we spent together all these years discussing research in your office or chit-chatting in the canteens. To dad Hock Wai, I thank you for being my pillar of strength and a source of unquestionable love, encouragement and comfort.

Equally, I accord great admiration to my beloved mother Poh Pei, whose support was always wonderful. I would also like to express sincere thanks to sisters Sook Fong and Sook Mei, as well as my wife Mee Yee for their continual encouragements, enthusiasm, and undefeated patience. Thanks also to Yin Fung for being distracting and noisy but cute.

Next, a word of recognition and commendation is accorded to all members of Prof. Leong's Research Allocation & Scheduling (RAS) research group, whom I have had great pleasure to meet and hold many a discussion on research and everyday topics. Particularly, a motion of thanks goes to David Ong Tat-Wee, Foo Hee Meng, Ho Ngai Lam, Dr. Ning Kang, Dr. Li Shuai Cheng, Dr. Kal Ng Yen Kaow, Chong Ket Fah, Melvin Zhang Zhiyong, Ye Nan, Max Tan Huiyi and Sriganesh Srihari for being very kind to me and incredibly helpful. To all the unmentioned RAS members and other NUS staff and students whom I've had the good fortune to meet, I assure you that you will be remembered and I will treasure all the time we've spent together.

Last but not least, I express my sincerest appreciation to the National University of Singapore for awarding a research scholarship to me so that I could realise my dreams of pursuing this higher degree. I am also grateful for the many knowledgeable, wonderful and helpful professors and lecturers that have taught me in NUS. May this beloved alma mater flourish in many more years to come.

Table of Contents

Summary

This research delves into two major areas of database research, namely (i) spatial database queries specifically for transportation and routing, and (ii) the reverse nearest neighbour (RNN) queries. Novel algorithms are introduced in both areas which outperforms the current state-of-the-art methods for the same types of queries.

Firstly, this research work focuses on a type of proximity query called the multi-point range query (MPRQ). We showed that MPRQ is a natural extension to standard range queries and can be deployed in a wide range of applications, from real-life traveller information systems to computational biology problems. Motivation for MPRQ comes from the need to solve this type of query in a real-life traveller information system (the Route ADvisory System (RADS) application, as well as its cousin web service Earth@sg Route Advisory Service at http://www.earthsg.com/ras). We researched various techniques used to solve MPRQ and discovered three approaches, presented their algorithms and analysed each of them in detail. Extensive, in-depth experiments were carried out to understand the MPRQ in a wide variety of problem parameters and MPRQ performs well in all of them against the conventional technique for solving MPRQ, i.e. the repeated range query (RRQ), used in proximity query systems today. Naturally, we extended MPRQ for external memory because in the real world, almost all applications deal with data that can never fit into internal memory. MPRQ also outperforms spatial join approaches for answering similar queries, such as the Slot Index Spatial Join (SISJ).

Secondly, this thesis lent contribution to RNN queries in the form of a hierarchical, novel data structure to find exact RNN results in metric space. The data structure is called RNN-C tree, making use of *k*NN graphs and inherent data clustering to find RNN. The RNN query is related to the nearest neighbour (NN) queries but is much harder to solve. Besides the RNN-C tree, we also presented several algorithms based on the grid file to find approximate RNN results, but is much faster. In some time-critical applications, sometimes approximate results are a good tradeoff between accuracy and response time. To the best of our knowledge, ours is also the first attempt to adapt the grid file data structure for solving RNN queries. As RNN is related to NN, the grid file becomes a natural choice as it can return NN results efficiently.

List of Tables

List of Figures

Chapter 1 Introduction

Wayfinding is a human need. In the past 20 years, an Internet boom has led to practical applications such as map viewing and driving route planning to be available on-line. These applications typically obtain a traveller's location and other desired preferences as input and return, after searching an underlying spatial database, the best available route to reach a destination. Most of them also provide many other services, most commonly the ability to show what is near the computed travelling route. These services have brought real-time information on-demand to reality.

In a transportation network scenario, public transportations such as buses and subways are modelled. In addition, extra services such as private vehicles routing and taxis routing (independent of a pre-determined route which is the case for buses), real-time traffic dispersal, searching of POIs such as public buildings, amenities and parks, are provided. Typically, a user is able to specify some preferences like reducing travelling costs, travelling time, or preference for certain roads. All these are made possible by advances in technologies such as the Global Positioning System (GPS) that can pinpoint a traveller's world coordinates to reasonable accuracy and mature third generation (3G) mobile devices that can be fitted into a car or be carried around (like PDAs and cellular phones). In the reports released by the U.S. Department of Commerce [DoC98, DoC01], 35% of the GPS units sold in the market is for car navigation, 22% for consumers' (private) use, 16% for survey/mapping (geographic information system related), 13% for tracking or machine control, the rest accounted for by OEM, aviation, marine and military use. By the year 2008, sales of civilian GPS reached US$28 billion.

In the telecommunications sector, location-based services (LBS) have long been touted as the next killer application for the wireless industry. Faced with growing subscribers equipped with GPS-enabled cellular phones and PDAs, there is a rush to develop commercially viable new applications like mobile yellow pages, safety calls and roadside assistance, location-based street and business directory search, traffic alerts, location-based games, personal navigation and tracking services. These are the kind of applications that many large corporations and government agencies will invest a great amount of money into. Despite the economic slowdown several years ago, Allied Business Intelligence has projected that the worldwide mobile data revenue will reach US$43 billion by 2014. Many researchers are funded by grants from their local transportation boards, municipal councils, state governments or private companies to carry out research aimed at modelling route queries, improving routing/searching algorithms, inventing efficient transportation models, expediting spatial operations and information retrieval (e.g. spatial join, closest pairs queries [Corr02], *k*NN-related queries), and so on.

One such recent work is the Route Advisory System (RADS) by [Lao99, FLLL99, TaLe04] which modelled the transportation network in Singapore and presented an algorithm that gives an optimal route based on multiple criteria tradeoffs (time against cost against number of transits) on multiple transport modes combination such as bus, subway and short walking. In addition to *route planning*, RADS is able to perform a *proximity query* that computes the points of interest (POI) and events that occur along the planned route that coincide with the time the traveller reaches that particular point in the route. It is not uncommon for a traveller to make a stop along the route to run an errand or simply to participate in some activities of interest such as exhibitions or sale.

1.1 Overview of Proximity Query

Let us define a typical route from point A to point B. To be a little more precise, the route comprises of a list of *k* segments of straight lines, where two consecutive segments are joined at a stop and there are *k*-1 stops. A value *d* representing the maximum distance of walking from any of the stops is given. We can roughly model the query as in Figure 1. In the remainder of this thesis, we shall refer to this type of user query as a proximity query. A mathematical definition of proximity query is found in Section 2.5.

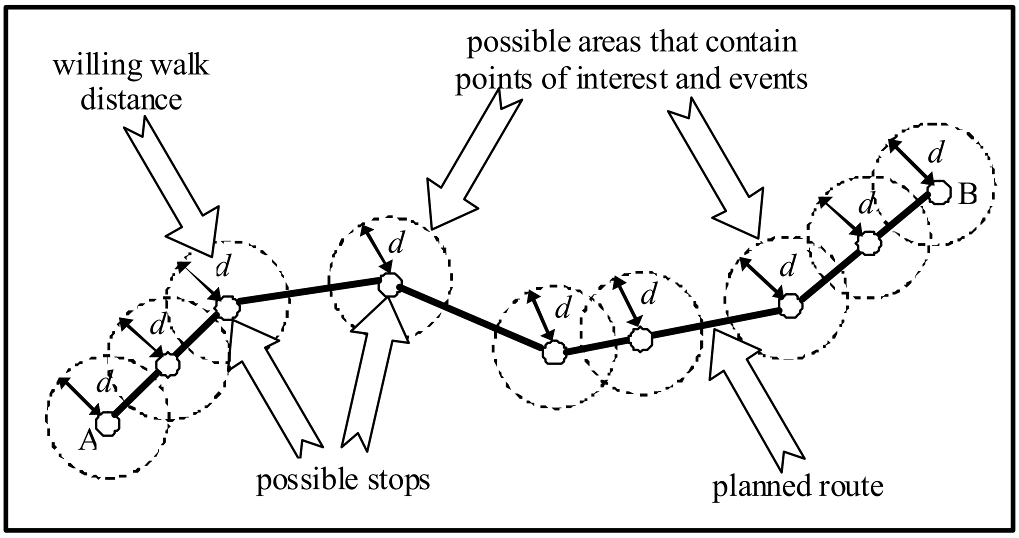


Figure 1. Proximity query modelled from a user scenario

The POIs that match the user query are divided into two types, namely static events and dynamic events. Static events are found at any one location permanently, e.g. buildings, lakes, bus terminals, parks, petrol stations and other establishments. Dynamic events usually occur at any location for a momentary period of time. They are characterised by a starting and ending time, or a daily recurring time window, e.g. a sale, blood donation drive, national day parade, musical concerts, etc.

The first part of this research was initiated as a natural extension to the RADS. RADS is a prototype software [FLLL99] that allows optimum trip planning for commuters with respect to one or more criteria combination of travelling cost, travelling time or transit mode. The first two criteria are self-explanatory. For transit mode, it means the switching of modes of transport in a single journey. This usually incurs waiting time for the next mode of transport to arrive at the stop, which is viewed as a penalty. The current RADS uses map and route data from the city of Singapore, but it can be easily suited to just about any cities in the world on availability of data.

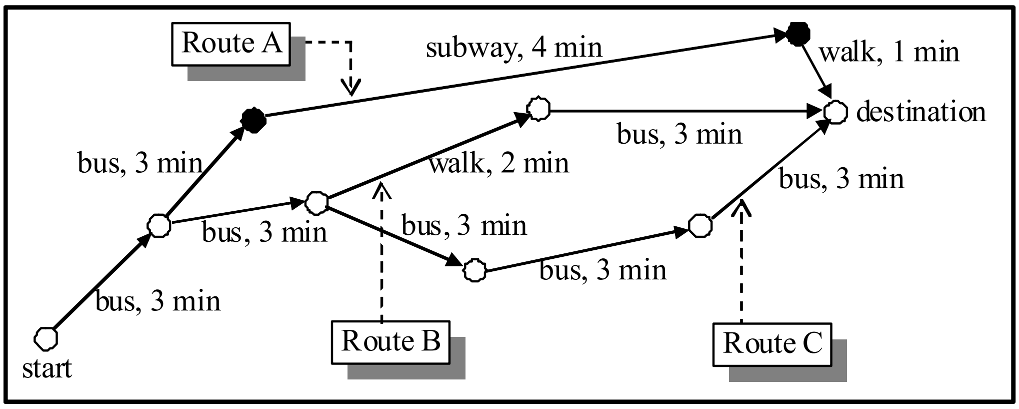


Figure 2. An example of RADS route planning. Route A represents optimal travelling time while Route B represents optimal transit mode. In real life, there are many possible route combinations to travel from start point to destination point

In Singapore, there are two major modes of public transportation, namely buses and subway called Mass Rapid Transit (MRT). In Figure 2, we illustrate the capabilities of the route planning engine of RADS with the three necessary modes to move from a start point to destination point (the third one is walking, modelled with an acceptable walking distance constraint). According to the statistics released by the Department of Statistics of the Ministry of Trade and Industry (MTI), Singapore [MTI09], in 2008 the average daily ridership was approximately 3.085 million, 1.809 million and 0.907 million trips for buses, MRT/LRT and taxis respectively. These figures are huge as the population is 4.839 million for the same period. Public transportation is the major mode of transportation in many parts of the world. Consequently, RADS is useful to the general public as a tool for smarter journeys, making available all alternatives of a journey at all times; to the public transport providers, RADS can help provide the big picture of the average journey, and to help identify missing/inadequate bus lines, enhance existing bus lines or plan the location of new bus stops (through generating extensive use cases).

With respect to the proximity query shown in Figure 1, we define the problem of finding all POIs and events (results) for a given set of stops (query points) within a given constrained distance d (a circular region of radius d centred at a stop) from each and every stop as *multi-point range query*. This type of proximity query is central to many applications and is widely studied in the literature.

1.2 Motivation

Multi-point range query (MPRQ) has many applications. Besides transportation planning problem, it can be adopted in air traffic control, water/electric/gas utilities, telephone networks, urban management, sewer maintenance and irrigation canal management [LaTh92, VTST93, ShLi97]. For example, in the telephone network problem we can find out how many users of different categories (e.g. business, residential, industrial, etc.) is dependent on a given telephone network line (e.g. one manifestation could be a non-weighted directed acyclic graph (DAG) whose vertices represent the telephone poles) so as to help in identifying heavy dependency on or usage of a particular line and for telephone network connection redistribution.

As another example, MPRQ can be generalised to a bigger scenario where each query point represents a town or a city, and the search distance represents the availability of certain establishments (e.g. a certain petrol station) within the town or city area. Coupled with a time factor, it could model town-to-town or city-to-city drive, providing an advanced knowledge on the availability of a favoured petrol station in upcoming locations and the estimate of petrol remaining at the time of reaching those locations (with petrol consumption tracking). The possibility of deployment in so many applications motivated us to research the MPRQ.

In many web applications that provide route planning as well as proximity query, the current approach is still limited to only performing proximity query one at a time on sections of the map (segment by segment), usually demarcated by road junctions or stretches of an expressway, even if the whole route is already pre-determined for the traveller. This has inadvertently localised the proximity information available to the traveller, supposedly in favour of saving Internet bandwidth and computation power. We foresee such web applications to be more intelligent in the future in that they not only provide the proximity information as requested but provide them accurately and quickly. Thus, the need for MPRQ as an enabling technology.

Note that this is by no means an exhaustive application of MPRQ. As another example, if we model electricity poles carrying a stretch of connected electricity cable along a road, performing MPRQ with the electricity poles as the query points will result in the number of households that are possibly connected to these switches. In a "what-if" analysis, MPRQ can be used to determine the number of households affected if the electricity cable is damaged or shut down temporarily.

The methods and algorithms that our research delve into are motivated by the following observation: when a path comprising many query points is given, and the objective is to return all events (also called object candidates [KMNP99] or sites [SoRo01]) near to these query points, where the searching mechanism for all query points is identical and related, and the results of that proximity query must be clean of any duplicate points. In our approach, we do not use a slicing technique to sample the path as in [SoRo01]; instead we explored query optimisation as a means to improve query processing.

1.3 Research Objectives and Scope

Conventionally, proximity query is solved by breaking down the route into many smaller segments interconnected by stops and performing multiple searches on spatial indexes to locate objects that are near each of the stops. Recall that this approach helps save bandwidth and improve response time in route planning applications on the web. One problem of this method is that it might result in many duplicate results if the segments are close to one another.

Therefore, a more specific query technique suitable for optimised spatial proximity querying is needed.

This research aims to achieve several objectives. We wanted to understand real-life GIS applications and the way they offer proximity querying. We studied and evaluated a type of query that we call *multi-point range query* (MPRQ), which can potentially perform proximity queries in a more intuitive approach. Many factors that affect the efficiencies of a proximity query were scrutinised, for instance, identifying a data structure that can support MPRQ. We rediscovered KDTopDownPack, a hybrid R-tree bulk-loading algorithm of [GaLL98] and subsequently designed some experiments to measure the performance of various data structures that can be used to support MPRQ.

Another objective of this research is to propose better search algorithms that can work well for answering MPRQ. There are many issues we need to address in order to achieve this objective. For example, the way pruning should be performed on the data structure during a search, and how effective they can be. Since MPRQ is observed to have some distinct properties, intuitively the orthodox set of pruning rules applicable for the general tree data structures might be inadequate. As a result, we defined some pruning rules that are implemented on a basic search algorithm. Experiments showed that applying our pruning rules are indeed more effective than without using them in the traditional query. Along this line, we have researched three techniques for fast pruning of input query points.

Last but not least, it is interesting to adopt the results of this research, the MPRQ, to genuine wide-ranging applications where it will be really useful.

Naturally, the first target application that comes to mind is where range query is widely used, which is a traveller information system. MPRQ was implemented as an extension in RADS. A brave, second application for MPRQ was targeted for the computational biology domain where research momentum is picking up very quickly in the past decade. Together with the self-organising map (SOM), MPRQ is part of a approach to perform multiple sequences similarity search in the peptide/protein identification problem.

The scope of the MPRQ research is narrowed down by a few assumptions: (i) the temporal aspects (time domain for dynamic events) of a proximity query is not considered, only static data is considered. Initial studies showed that temporal pruning first reduces the number of candidates by less than 5% on average whereas spatial pruning first gives a reduction of over 90% from the initial candidates set, (ii) the query algorithm is for $\Re^2$ space and the computation techniques based on L_2 Euclidean distance metric, (iii) query region is circular (using distance d as a radius), (iv) a 2-d query point represents the centroid of any polygonal objects on the map. Further computations are assumed to precisely confirm the correctness of a 2-d point result, (v) spatial objects on the map are adequately bounded by a minimum bounding rectangle (MBR). All the above assumptions hold for all MPRQ results presented in this thesis, unless otherwise stated.

It is argued that the road distance (L_1 Manhattan distance) is a better representative for determining the result of MPRQ, particularly in the case of transportation and road networks. We state that our method works for other distance metrics, as long as consistently applied. In general, we meant for

MPRQ to work in other scenarios too, such as in bioinformatics problems, where the edit distance might be more appropriate.

1.4 Contributions of Thesis

This thesis consists of three major contributions. Its principal contribution is the in-depth study of the multi-point range query for both internal and external memory cases, and the introduction of the MPRQ algorithm, an efficient algorithm for the processing of range query with multiple points as input. Instead of performing a range query for each and every point, MPRQ takes as input the whole set of points and perform the query once. MPRQ visits the spatial index only once by utilising smart pruning rules at every level of query processing within the spatial data structure, resulting in optimal I/Os. The key idea of MPRQ is about the efficient pruning of the input (of multiple points) with respect to each node encountered during the traversal of the spatial index, as well as optimising the results returned (for example, a large enough search distance will cover an intermediate level node which means all nodes and eventually leaf objects under it becomes the results) to decrease unnecessary computations in obvious cases. Several techniques have been developed for pruning of the input. Empirical results show that MPRQ can significantly improve query processing time both in internal and external memory [NgLH04, NgLe04].

Secondly, this thesis lent a huge contribution to the reverse nearest neighbour problem (RNN). The RNN query is a proven non-trivial problem no less than nearest neighbour (NN) queries. Although related to NN, the RNN results cannot be derived from NN's. RNN queries are categorised into those that find exact results and those that find estimated results. A novel, hierarchical data structure to find exact RNN results in metric space was presented. The data structure is called RNN-C tree, making use of *k*NN graphs and inherent data clustering to find RNN. Besides the RNN-C tree, we also presented several algorithms based on the grid file to find approximate RNN results, but is much faster. These algorithms are collectively called RNN-Grid. As RNN is related to NN, the grid file [NiHS84] becomes a natural choice as it can return NN results efficiently. Empirical results show that RNN-Grid is faster than other RNN algorithms in the same category, yet it can achieve higher recall. As for RNN-C tree, to the best of our knowledge, it is one of only two available RNN algorithms that can solve RNN in general metric space. Compared to its competitor, RNN-C tree is 1.5 times faster and does one order of magnitude less distance computation, which is central to pruning rules.

The third contribution of this thesis is two successful applications of MPRQ in traveller information system and computational biology research. We had successfully adopted MPRQ as a natural extension to the query processing in RADS. Based on the pre-planned multi-criteria, multi-modal route that a RADS user obtained as input, MPRQ is able to efficiently return all the POIs in the map within the vicinity of the route. We had also successfully adapted the MPRQ algorithm for performing similarity sequences queries by coupling it with a trained self-organising map (SOM) [Koho01]. This is a novel approach in two ways: (a) the SOM is mostly used for clustering analysis and visual representation of sequences for detecting similarities [BeGe01, MMSG04, ASKK06]. Researchers mostly view a trained SOM as the end result for spotting sequences similarity (using it manually by visual), and almost never exploiting it for further uses (post-trained SOM uses). To the best of our knowledge, post-trained SOMs were only adopted in image retrieval applications for large image databases [ZhZh95] but they have never been used in sequences similarity problem; (b) by applying MPRQ on the SOM, we are able to perform a single similarity query not just for a single input sequence, but rather a series of input sequences simultaneously and obtain results that are similar to the input sequences as a whole.

1.5 Organisation of Thesis

This thesis is divided into 2 parts: Part I focuses on MPRQ and spans Chapters 2, 3 and 4; whilst Part II focuses on RNN and is covered in Chapters 5, 6 and 7. A brief outline of this thesis is as follows: Chapter 2 summarises the relevant literature regarding data partitioning, query results filtering methods, data structures and discusses the MPRQ framework. Chapter 3 presents techniques for algorithms, experimental results and analysis of MPRQ in internal memory. Chapter 4 presents the extension of the internal memory MPRQ algorithms to external memory, introducing two more algorithms, with experimental results and analysis. It also covers a comprehensive look into the performance of MPRQ in external memory against relevant spatial join algorithms that can possibly be used to solve MPRQ.

Chapter 5 summarises the relevant literature for related approaches to solving the reverse nearest neighbour (RNN) problem. This chapter also features some statistical analysis on the parameters used by RNN-Grid to estimate results, as well as on the bounds of RNN-C tree height. Chapter 6 explores the RNN and presents four algorithms in the RNN-Grid approach for solving RNN with estimated results. Chapter 7 subsequently describes a data structure we call the RNN-C tree for solving RNN with exact results.

Finally, Chapter 8 concludes with some proposed extensions to this research and future work, for both MPRQ and RNN problems. Appendix A briefly describes a piece of research work this author has published, i.e. applications of MPRQ in problems from the computational biology domain, with emphasis on the peptide identification problem.

PART I

Multi-Point Range Query

Chapter 2 MPRQ and Related Work

Many applications that provide route-related services have an underlying database that does not change very frequently, as we do not expect bus stops and subway stations to be relocated all the time, if at all. Such databases are termed static. In contrast, databases that are subject to frequent updates are said to be dynamic. Usually, we query a spatial database to look for only subsets of objects that fit the conditions of our queries. This is called a region query. A special case of a region with zero area is called a point query. In order to facilitate searching of the database efficiently, suitable data structures are used to store the objects in the database based on the knowledge of the data being static or dynamic, and their distribution in space. Since geographical objects relate to each other primarily based on their relative position to one another, we term this as spatial indexing.

Data structures and spatial indexing are just two aspects of a spatial query. [Knut98] listed the three typical queries: point query, to find a point data with exact attribute; range query, to find all point data that exist in a given region; and boolean query, which answers the existence of point data satisfying point query or range query. Recent advances in geographical applications created the need for many operators for spatial searching, including intersection, enclosure, adjacency, spatial join and nearest neighbour queries [LuOo93, GaGü98].

In many scientific, geographic and engineering applications, the storage and efficient retrieval of multi-dimensional data is extremely crucial.

Traditional one-dimensional data structures such as B-trees [BaMc72] or hash tables do not provide the answer to storing polygons, squares and rectangles. A number of data structures have been designed to cater for multi-dimensional data, such as the two-dimensional index R-tree [Gutt84] and high-dimensional indexes such as M-tree [CiPZ97] or iDistance [YOTJ01, JOTY05]. In performing proximity queries, we need to implement an indexing scheme that is most suitable for organising the data points so as to effectively prune away most unnecessary results. We describe several methods in the literature.

2.1 Space Partitioning and Data Partitioning

A data structure used for indexing can be divided into two categories: space partitioning (SP) and data partitioning (DP).

In SP, search space in the problem domain (usually Euclidean space in planes, in general $\Re^d$ in hyperplanes) is divided into two or more disjoint (non-overlapping) subset space so that during query, data can be found in exactly one of the subset space. SP schemes are usually hierarchical in nature, and a smaller piece of subset space can be recursively space-partitioned to become smaller non-overlapping space at a lower level. The space is organised as multiple levels of a tree, and the tree is termed an SP-based indexing data structure.

On the other hand, if the search space in the problem domain is divided into two or more disjoint subset space based on the positions of data points, such schemes are called DP. Similar to SP-based index, DP-based index structures are also mostly hierarchical. The structure of a DP-based index is highly dependent on the order in which the data points are presented (insertion order) as well as their positions when the index is constructed.

2.2 Coarse Filtering and Fine Filtering

One common strategy in query processing involves the use of coarse and fine filters [NiWi97], which is also called filter-and-refine technique [SeKr98, SCRF99] or geometric filtering and exact geometry processing [KrSB93]. In terms of spatial query processing, the trend to use a two-level processing is relatively new.

Firstly, approximate geometric techniques such as the minimal orthogonal bounding rectangle of an extended spatial object is used to quickly and cheaply filter out as many objects as possible. This coarse filter is usually easy to perform and cheap on computational time and cost [NiWi97]. The overall running time of the whole spatial query is very much influenced by the success of the implementation of a coarse filter. This is because in the subsequent fine filter, or refine process, exact geometry is applied on every remaining candidate objects to eliminate false positive results. This process is extremely expensive as heavy computation is not uncommon to eliminate large candidate objects as they may have tens or hundreds of dimension (a typical polygon representing an accurate, complex real-world object typically has 1000 or more edges).

2.3 Point-Region Quadtrees

The quadtree [FiBe74] is a well-known class of DP-based hierarchical data structure for storing data points. Data points are assigned into one of four quadrants in the tree, based on their coordinates in relation to points already inserted into the tree. There are always four child nodes to each internal node, and each internal node contains a data point (its coordinates). [Same89] described PR quadtree (point region quadtree), an extension that associates each quadrant with a relative data point region where data points are stored only at the leaf nodes. The structure of the quadtree encourages sub-dividing of the data space, even when two points are actually very close by and therefore have a great chance of answering a range query.

In order to save time and space in sub-dividing the space into four sub-regions (where three of them will be empty), some form of bucket methods were proposed [Knot71, Oren82, MaHN84]. A bucket is a presumably short linked list which holds data points that are close to each other in space. The size of the bucket is determined by a certain threshold; if f is the fanout size of the quadtree, the bucket size is usually between f and $2f$. When a query reaches the leaf node which contains a bucket, all the points in the bucket are compared sequentially. An example of PR quadtree is illustrated in Figure 3.

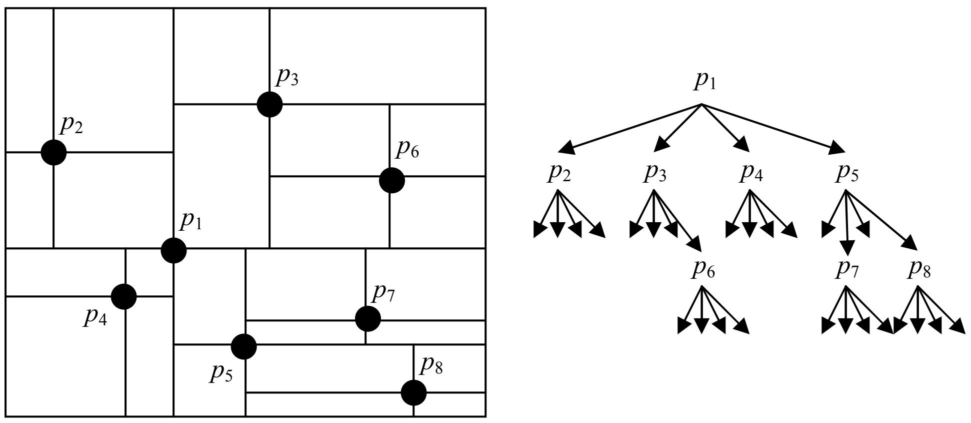


Figure 3. An point-region quadtree and the data points it represents. The data points are organised hierarchically in the order they appear, causing space to be decomposed w.r.t. data points

The PR quadtree was invented to overcome some of the drawbacks of using fixed grid cells structure. When data points are not uniformly distributed, many cells in the fixed grid will be empty, which is not efficient in terms of memory usage and utilisation. PR quadtree is a combination of the fixed grid method and binary search tree which can handle non-uniform data well.

2.4 R-trees

The R-tree was introduced by [Gutt84] and has since become a popular data structure for spatial searching. One reason is that, apart from its elegant generalisation from B-tree for storing multi-dimensional objects, the R-tree is capable of storing a myriad of complex objects such as lines, polygons in addition to mere points. Like the B-tree, R-tree is a hierarchical, height-balanced on-line data structure where all the leaf nodes are on the same level (or differ by at most 1). Each internal node of the R-tree has the form (MBR, *ptr*) where MBR is the minimum bounding rectangle that encompasses all the MBRs of its child nodes in space (the MBR enclosure property).

An MBR is characterised by a set of minimum and maximum coordinates defining a rectangle whose sides are parallel to the coordinate axis. Using the MBR instead of exact geometrical representation, any complex object is reduced to two points that define the most important feature of that object (i.e. its position and extension). The root node of an R-tree has an MBR that is the minimum rectangle of all the objects in the search space. Each leaf node of the R-tree also has the form (MBR, *ptr*) where the pointer points to an object being stored, rather than to another node. An internal node can have more than one child whose MBR overlaps and possibly covers a particular object. Therefore, in order to search for that object, it is compulsory to traverse all the children nodes involved. Due to this inefficiency, the R^+-tree was invented by [SeRF87] which eliminated overlapping altogether.

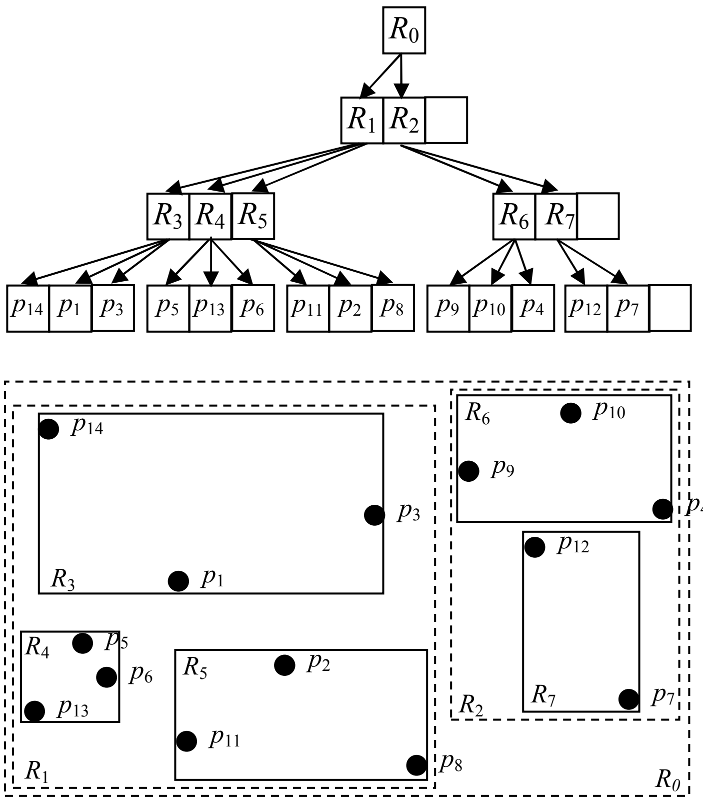


Figure 4. An example of a bulk-loaded R-tree. The R-tree is built from bottom up

An R-tree node has to be split when an object is inserted into a leaf node that is full. The splitting causes its immediate parent node to have one more child, and if the parent is full, it is also split. This process propagates up the tree until it hits a node that is not full or the root is split. [Gutt84] introduced three node splitting heuristics called exponential, quadratic and linear split. Many other splitting strategies were reported that minimised the overlapping area after the split [BKSS90, KaFa94, AnTa97].

The R*-tree [BKSS90] is a variant of the R-tree which is different in overflow handling and splitting policies. To handle an overflow node, it removes some rectangles from the overflowed node and re-inserts them from the root of the tree in the hope that they would be accommodated by some other non-full nodes.

The data structures discussed so far are all on-line data structures. They generally could have up to 73% node utilisation [AnSa96]. Their node utilisations and tree structures are compromised by the ability to insert or delete rectangle data dynamically. If we have *a priori* knowledge of the data before the data structure is built, we could possibly produce a fully packed R-tree that greatly facilitates searching. This method of constructing a spatial index is called bulk-loading.

Hilbert-Sort R-tree

[KaFa93, KaFa94] proposed the Hilbert-Sort (called HilbertPack in this thesis) R-tree which imposes a linear ordering based on the mapping of the Peano-Hilbert fractal curve [Hilb91], a space-filling curve as shown in Figure 5(a). The idea of space filling curves is to group similar data together, in this case the MBRs. The centre points of the MBRs are sorted based on their distance from the origin, measured along the Hilbert curve. This determines the linear order in which they are placed into the nodes of the R-tree.

The R-tree is built bottom-up starting from the leaf level (external nodes pointing to spatial data), resulting in a tree that is fully packed except, of course, for the last node at every level of the tree. Under the Hilbert curve, objects with close linear order number are also spatially close (although the reverse is not true). Query processing is proven more efficient than other dynamic versions of R-trees (e.g. R*-tree) of up to 36%. The structure of HilbertPack R-tree is adapted from B*-tree, where the keys refer to the Hilbert value of the data MBRs. Figure 5(b) reveals that some MBRs of HilbertPack at higher levels are very large, which will have an adverse impact on query processing as confirmed in our experiments.

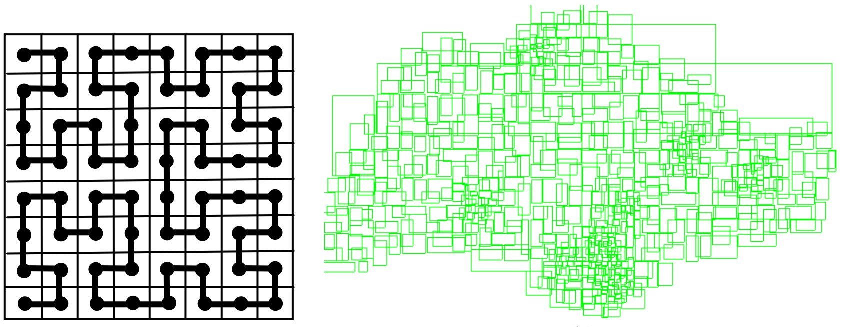

(a) (b)

Figure 5. An example of applying Peano-Hilbert space filling curve to (a) an 8×8 grid in 2-d, and (b) the SG dataset

Sort-Tile-Recursive R-tree

Sort-Tile-Recursive (called STRPack in this thesis) is a bulk-loading algorithm for the R-tree [LeEL97]. The basic idea for the STR algorithm is to *tile* the data space using $\sqrt{r/n}$ vertical slices so that each slice contains enough rectangles to pack roughly $\sqrt{r/n}$ nodes, where r is the number of rectangles and n is the cardinality. The centroids of rectangles are used as reference points. Rectangles are *sorted* by x-coordinates and partitioned into $\lceil \sqrt{r/n} \rceil$ vertical slices each containing $\sqrt{r}$ rectangles. The process is *recursively* repeated but now with rectangles sorted by their y-coordinates. Figure 6 reveals that most MBRs of STRPack are elongated, which will also have an adverse impact on query processing. The authors claim that STRPack outperforms HilbertPack for mildly skewed or uniform data.

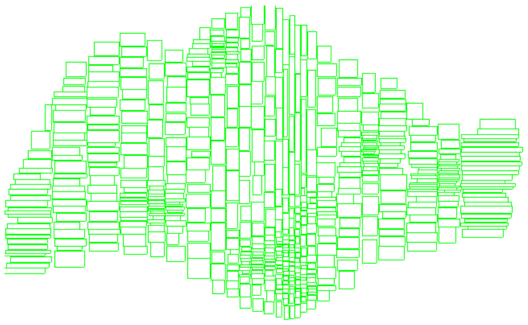


Figure 6. MBRs of the R-tree of the SG dataset constructed with STRPack with cardinality $n = 32$

Top-down Greedy Split R-tree

Top-down Greedy Split (TGS) is another bulk-loading algorithm proposed by [GaLL98]. TGS is motivated by the two key ideas: (i) it minimises the top levels first since the potential for cost reduction is higher, while (ii) considering all partitions induced by guillotine cuts such that resulting sub-trees are fully packed. TGS is an aggressive approach to greedily construct the various sub-trees of the R-tree. It recursively applies a basic split step which partitions a set of r rectangles into two subsets by a cut orthogonal to an axis. A cut must meet the condition that minimises the cost of some objective function $f(r_1, r_2)$ where r_1 and r_2 are MBRs of two ensuing partitions, and one subset must result in a fully-packed sub-tree. The recursion is applied to both subsets until there is one subset per child.

Two major disadvantages are that TGS is difficult to implement and it requires a relatively much larger loading time. This led us to discover an algorithm modified from TGS which has similar performance but fast, which we call KDTopDownPack.

2.5 Proximity Queries

We use the term *proximity query* to describe a type of spatial query that is unorthodox in the sense that consideration is given to the multi-point input for each instance of the range query. We view the multi-point input and the combined results that we obtained from the query as *one* proximity range query. The points that form the input to the proximity query are given in a list or array, in addition to a given search distance. The objective is to perform range query efficiently and report all the points (or objects) that lie within the range of the distance from the set of query points. Mathematically, for the general range query, given a finite set of points $P = \{p_1, p_2, \ldots, p_n\} \subseteq \Re^2$ and a circular region $R \subseteq \Re^2$, find the set of points $Q = P \cap R$.

At present, research interests are focused on addressing the *k*-nearest neighbour (*k*NN) queries. It has become a hot topic in the database research community and also is addressed by the computational geometry research community because it is useful in numerous applications such as data mining and knowledge discovery, multimedia database, pattern recognition, urban management and CAD/CAM systems.

In short, the general *k*NN problem is defined as given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n objects, and a query point q, find a subset $S' \subseteq S$ of size $k \leq n$ such that for any $p_1 \in S'$ and $p_2 \in S - S'$, $dist(q, p_1) \leq dist(q, p_2)$. Various techniques and algorithms were proposed for performing this type of queries in low-dimension, which is also the focus of this research. For example, [RoKV95] proposed a branch-and-bound method to answer 1NN queries (BB-NN) and then generalised them for finding *k*NN. The BB-NN algorithm was based on two metrics for ordering the NN search, and three pruning rules when visiting nodes during the search. Figure 7 illustrates. The various metrics and the concept of distance are detailed in Chapter 3.

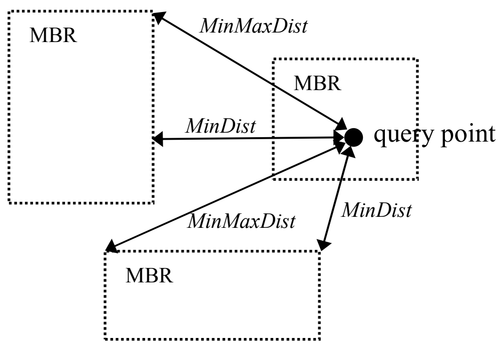


Figure 7. The concept of MinDist, and MinMaxDist as used by [RoKV95] for branch-and-bound *k*-nearest neighbour search

Later, [PaMa96] extended this work using a multi-disk multi-processor architecture, deriving the parallel nearest neighbour (P-NN) method. Since the BB-NN is a sequential algorithm, the P-NN algorithm generally outperforms it as the value of *k* increases, with as much as 60% improvements for large values of *k* (e.g. *k* = 400).

Only very recently, research focus on spatial queries has started to address the problem of *k*NN for a moving query point (*k*-NNMP), which is useful for applications in transportation and logistics where a continuously moving car wants to track where the nearest petrol stations are [SoRo01]. This problem is different from the MPRQ in that the problem addresses the need to know the *k*NN of a moving query point at any one time along its path.

2.6 Variants of Multiple Range Queries

Query scheduling for multiple range queries was studied by [PaMa98]. Based on the idea that the performance of multiple queries can be improved if they share common data (subsequent nearby queries retrieve a lot of the same data), the authors presented an algorithm that sort its queries (of rectilinear rectangles), group them together so that they are spatially close, and finally pass them for processing. Results were shown for R-trees built on Hilbert-curve sorted objects. Although the queries seem similar to the MPRQ, the main differences are (i) they are doing inter-query optimization, while MPRQ is a single query, (ii) the combined results obtained by joining the queries raised another issue which is the separation of results; extra processing needed to determine which objects belong to a specific range query. MPRQ generates cumulative results that answer the query as a whole.

There are many variants of the multiple queries problem. One such recent work is the group NN queries [PSTM04], where two sets of points P (database) and Q (multiple input) are given and the aim is to find a point p from P that minimizes the sum of distances $|pq_i|$ for all $q_i \in Q$. In [ZMPT04], for the same sets of points, the aim is to find the nearest neighbour from P for each and every point in Q. Three algorithms were described. The first is multiple NN (MNN) which is similar to RRQ in this thesis, except that the latter returns all points, instead of the nearest, w.r.t. the query points in Q. This approach is straightforward and already proven to be very slow in both [ZMPT04] and this thesis (Section 3.2.8). The second is batched NN (BNN) which is designed for cases where Q cannot fit in memory. BNN breaks all points in Q into arbitrary groups (bounded by two thresholds, max number of points per group and MBR size of the grouped points) to be processed together against P. The third approach is hash-based NN (HANN) where the points in P and Q are hashed to a grid and subsequently loaded pairs $<H_Q,H_P>$ ($H_P \in P$, $H_Q \in Q$) of buckets covering the same region are searched for each point in H_A its NN in H_B (with consideration for points near grid borders that might have NN in an adjacent region).

2.7 MPRQ Terminologies

This thesis deals with numerous issues regarding proximity queries, particularly a type of spatial query we call multi-point range queries (MPRQ), as well as its optimisation. The planned route in Figure 8 is returned by most route planning systems as it is the core functionality of a routing engine. Terminologies used throughout this thesis will be defined below.

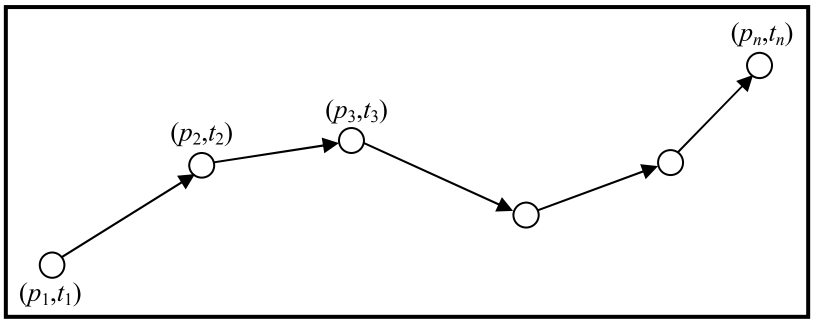


Figure 8. A planned route consisting of a series of directed segments joined by nodes, each node/point representing a possible stop. A node is also associated to a time when that node is reached

Definition (Path): *Given a start point s and an end point z, a path is defined as any sequence of directed, non-cyclic, connected points from s to z represented as* $(s, p_1, p_2, \ldots, p_k, z)$ *and consists of* $(k+2)$ *nodes.*

Definition (Planned route): *A planned route P is a path that is also associated to a corresponding sequence of arrival time T at each point when the path is traversed. T is represented as* (t_s, t_1, t_2, ..., t_k, t_z) *where* t_s *is termed the start time. The route size of P, denoted |P| is equal to the number of points in P. A planned route is usually optimal w.r.t. some user-specified criterion such as time, cost or |P|.*

With a planned route *P* returned by the routing algorithm, we perform proximity query on the set of points. To find all the POIs along the path, the conventional technique is to perform range query |*P*| times of the radius *d*, and returning the union of the search results set *R*. Mathematically, it can be written as $R = \bigcup_{p_i \in P} \text{Query}(p_i, d)$ where Query(p_i, *d*) is a nearest-neighbour query that returns all the nearest neighbours of distance *d* from point p_i. We call this straightforward technique repeated range queries (RRQ). This technique works when the search regions do not overlap, as shown in Figure 9. However, this is actually not a common occurrence in most real-life situations.

Definition (Incidental event): *Given a path* $P = (p_1, p_2, ..., p_n)$*, a distance d and the proximity query result set R, an event e incidental to P is a dynamic or static point-of-interest (POI) that is found in the spatial database which satisfies:* $dist(e, p_i) \leq d, \exists p_i \in P$*. An event can be incidental to more than one intermediate point in P (i.e.* $p_2 ... p_{n-1}$*).*

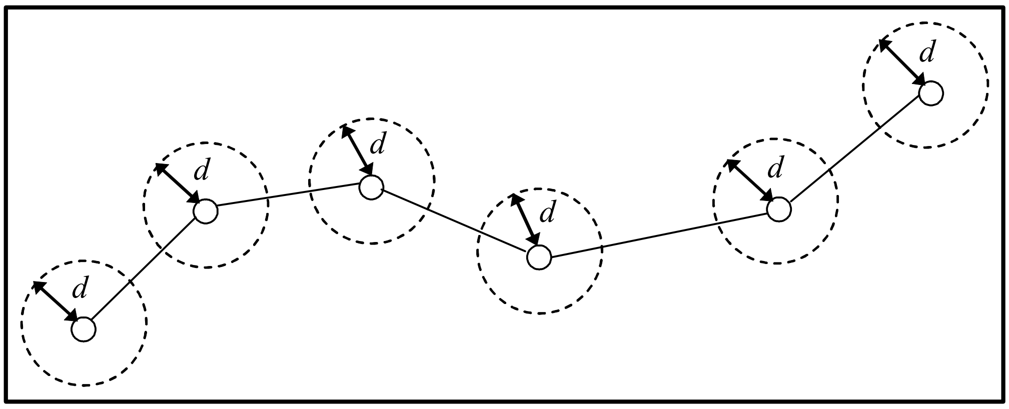


Figure 9. Conventional technique for performing proximity queries on a planned route *P*. MPRQ is broken down into smaller queries with each being executed sequentially and the results combined

As the search region *d* is enlarged, the conventional method becomes very inefficient because the combined results contain many duplicate events and some queries become almost redundant. This is evident in Figure 10. In a transportation network setting, route *P* can be a bus route while the nodes in *P* can be the bus stops that the bus calls at during the journey. On average, for a city that heavily relies on public transportations, bus stops are built within 200-300 metres of one another. Almost all the time in most queries there are some number of duplicates results. Therefore, we strive to perform the proximity query just once, using techniques to effectively remove duplicate results and efficiently execute the query.

Definition (Multi-point range query): *Given a planned route P and a distance d, using a single query, find all the events incidental to all the intermediate points in P and return the non-duplicate results set R.*

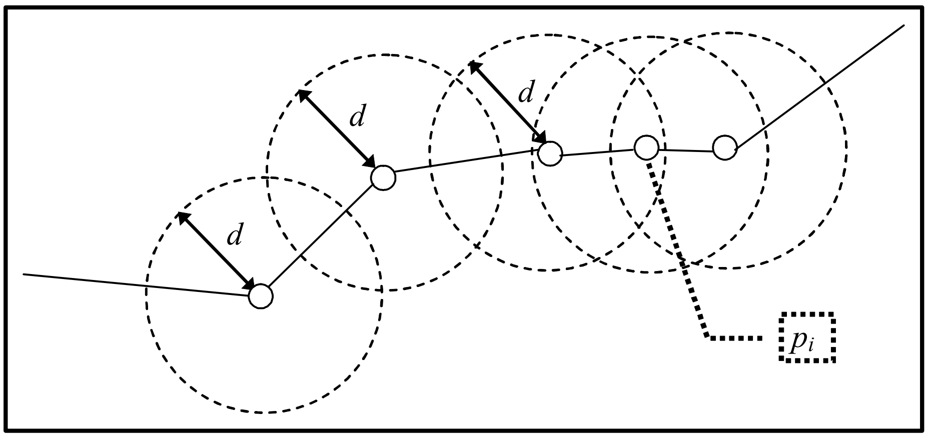


Figure 10. Performing queries on some route P gives many duplicate results; some queries like the one performed on point p_i even become almost redundant

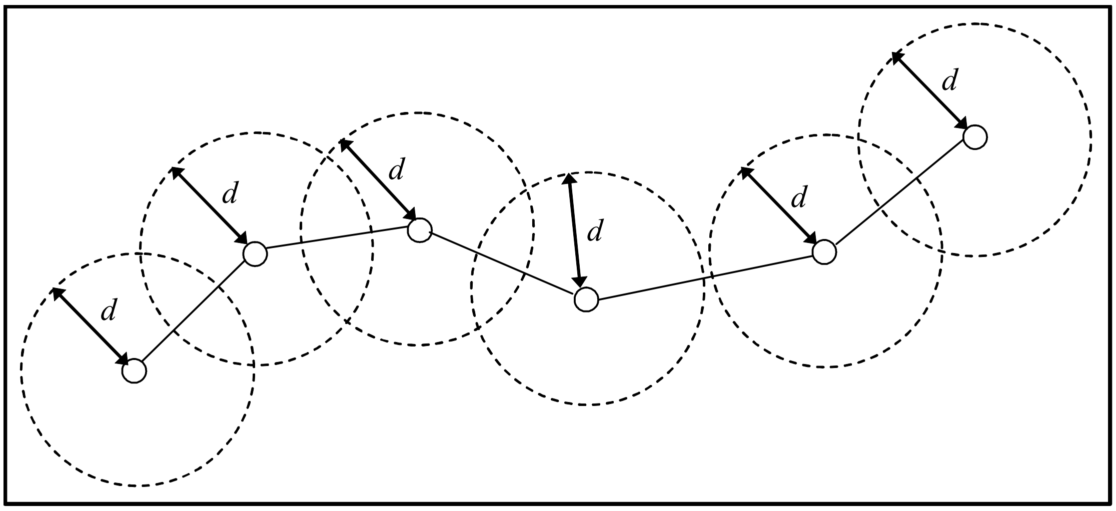


Figure 11. Performing multi-point range query on the planned route P. We are interested in all the non-duplicate incidental events that are within a distance d from all nodes in P

2.8 MPRQ Formal Problem Definition and Framework

The formal definition of the MPRQ is presented in this section. Firstly, the constants and variables are defined. This is followed by giving the definition and the constraints of the MPRQ.

Let d be any search distance where $d > 0$, N be a spatial database of 2-d points, $P = (p_1, p_2, \ldots, p_n)$ be a planned non-empty route with n-1 segments where each $p_i \in \Re^2$ forms the segment from p_i to p_{i+1}, $1 \leq i < n$ and $P \not\subseteq N$.

Find the set of results $R = \{p_i \in P \mid dist(e, p_i) \leq d, \forall e \in N\}$. The set R implies two observations: (i) the size of the results set being at most equal to the size of the spatial database (i.e. no duplicates are allowed); (ii) any event reported in the results set R will be within the distance d from some point in P.

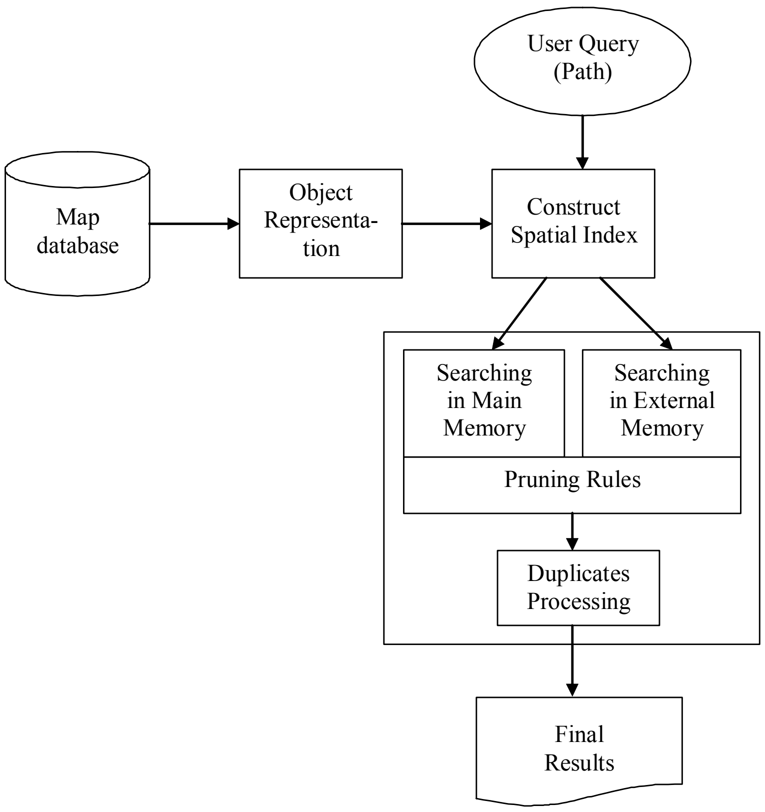


Figure 12. The multi-point range query framework depicts various areas that this research addresses, among others constructing the spatial index, proximity query pruning rules and duplicates processing

Figure 12 depicts the multi-point range query framework upon which implementations for this research is based. The user query is in the form of a set of segments forming an acyclic path. The final results is a set of object references (or pointers) of all valid non-repeating objects that answer the query. In the scope of research, the MPRQ is constrained to use an averaged midpoint (centroid) to approximate any polygonal spatial objects for query processing.

MPRQ not only performs the query but also filters off duplicate points and cleanly return only the results set of unique points. In other words, the results given by MPRQ do not include duplicates by default. On the contrary, RRQ cannot perform duplicates removal as each query point is processed independently of each other, sequentially. When all the results are obtained, the combined results must be post-processed for duplicates removal. It is already too late as the costs to obtain all results have been incurred.

In our proximity query, the distance or metric used in calculating the proximity of any POI from the planned route is based on the L_2 metric, i.e. the Euclidean distance $dist(x, y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ for our case in 2-d space. It follows that all the results must satisfy the following four conditions for a metric to hold true: for any three points *x*, *y* and *z*,

- $dist(x, y) \geq 0$: distance is a nonnegative number
- $dist(x, y) = 0 \Leftrightarrow x = y$: distance of an object to itself is 0, i.e. identity
- $dist(x, y) = dist(y, x)$: distance is symmetric
- $dist(x, y) \leq dist(x, z) + dist(z, y)$: distance observes the triangle inequality principle

It is possible to use any Minkowski metric, but for consistency with most other research works in the literature [RoKV95, PaMa96, MaMo01] we chose the Euclidean metric. We shall emphasise that our algorithms are, without loss of generalisation, valid for other metrics of any order of *p* (*p*-norm distance) in *k*-dimensional space which is defined as

$L_p = \left(\sum_{i=1}^{k} | x_i - y_i |^p \right)^{1/p}$ for $1 \leq p \leq \infty$, and $L_p = \max_{1 \leq i \leq k} | x_i - y_i |$ for $p = \infty$.

Chapter 3 Main Memory Algorithms for MPRQ

In this chapter, we formally present the study of multi-point range query (MPRQ) for internal memory. This is because MPRQ is believed to work very well for small route and small database. Recall that in Section 2.7, we presented a problem scenario and definitions that describe a typical route and its connection to proximity query. The concept of MPRQ is subsequently formulated mathematically in Section 2.8. Moreover, implementation of MPRQ in this chapter follows the framework we defined for solving MPRQ also found in the same section.

Algorithms and techniques for solving MPRQ are presented in Section 3.1, followed by extensive experimentations described in Section 3.2. In the next chapter, MPRQ is investigated for cases where the database is stored in external memory.

3.1 MPRQ Algorithms

As our approach, we used a depth-first search strategy aided by various pruning rule techniques that would prematurely halt the search in an intermediate MBR (when certain conditions are met) and retrace its steps backwards. Technically, it can be termed as a branch-and-bound technique. The similarity is that we would still avoid many branches and their sub-trees altogether when the pruning condition matches. The major difference is that we do not keep track of an objective value and use it to terminate the search in a branch. Instead, pruning rules tell us when to shrink our query region size (temporarily remove some input search points at different levels of the R-tree) and when to know that all subsequent objects under an MBR qualifies as the result (thereby reducing unnecessary computations).

Our pruning rules show promising results. Applying them on the MPRQ produces improvement in query processing time of up to 94.2% on average from the standard query processing time in which only the basic pruning rule is activated (i.e. avoid going into MBRs not intersecting with the query regions), which is the standard branch-and-bound technique.

3.1.1 Preliminaries

Definition (Node colour): Consider a planned route $P = \{p_1, p_2, \ldots, p_n\}$, a node R in the R-tree and $C(p, d)$ as the query region centred at point p, with radius d. Then,

- R is said to be *white* w.r.t. P iff $R \cap C(p_j, d) = \varnothing$ *for all* $p_j \in P$,
- R is said to be *black* w.r.t. P iff $R \cap C(p_j, d) = R$ *for some* $p_j \in P$,
- R is said to be *grey* w.r.t. P otherwise.

MPRQ will prune off the white nodes and search only the grey and black nodes in the R-tree. Note that these are natural extensions of the normal range query. This pruning rule is called NodeOut.

Definition (Query point colour): Consider a query path $P = \{p_1, p_2, \ldots, p_n\}$ and a node R in the R-tree. Then,

- p_j is said to be *white* w.r.t. R iff $R \cap C(p_j, d) = \varnothing$,
- p_j is said to be *black* w.r.t. R iff $R \cap C(p_j, d) \neq \varnothing$.

The MPRQ algorithm will *also* prune off the white query points. This is where it enjoys the advantage of simultaneous pruning. This pruning rule is called PointOut.

3.1.2 Algorithm 1: RRQ

A straightforward approach to answer the multi-point range query is to apply the standard range query (RQ) to each and every point in *P* and combining the results, i.e. MPRQ(*P*, *d*) = $\bigcup_{p \in P} RQ(p, d)$. We call this repeated range query (RRQ). Since a call to RQ is independent of one another, each query will search the spatial index once. This is repeated as many times as the size of our planned route (|*P*|) to retrieve all results. A post-processing step is usually needed to eliminate the duplicates that result from overlapping of query regions (Figure 10). Apparently, this method is extremely expensive even when the whole spatial index resides in internal memory.

This method is used by many web-based proximity query applications that disguise their weaknesses behind multiple, separate web pages for two reasons: (i) web pages displaying partial results are more intuitive to navigate and digest (reducing information overload) and reduces loading time; (ii) web pages are used to break up multiple points query using standard RQ as it is easier to implement, and they might not have an efficient algorithm to retrieve results given multiple query points.

```
RRQSearch(R, P, d, Obj)
// Input:  MBR R, a query path P, a search distance d
// Output: Obj - set of objects within distance d of some
//         point in P
begin
  R ← R-tree.root
  for each $p_i$ in P do
    RQSearch(R, $p_i$, d, Obj);
  endfor
end; {procedure RRQSearch}


RQSearch(R, p, d, Obj)
begin
  if (R is a leaf-node) then
    Process objects in R wrt point p;
  else
    for each $R_c$ of node R do
      RQSearch($R_c$, p, d, Obj);
    endfor
  endif
end; {procedure RQSearch}
```

Figure 13. Algorithm for implementation of RRQ

3.1.3 Algorithm 2: MPRQ-MinMax

We introduce a combination of techniques called pruning rules that solve multi-point range query and make it possible to sustain good performance even when simultaneously dealing with a large set of query points. Recall that the R-tree data structure organises its nodes in a hierarchical manner where each node stores the minimum bounding rectangle (MBR) that contains child nodes of which no area is outside the MBR of their parent. Besides the MBR of a node, the pruning rules include two metrics computed to aid the pruning process, called MinDist and MaxDist, which are described below.

Definition (MinDist): *Given a point p, and a node in the tree, MinDist(node, p) is the smallest possible distance between p and any points contained by the node. If p is located outside the MBR of the node, we measure the distance of point p from the nearest boundary or the nearest vertex (corner) of the node's*

MBR. If p is located within the MBR of the node, MinDist is defined as having zero distance.

Lemma 1. For any point p, MBR R, search distance d and $C(p, d)$ is the circle with centre p and radius d, $\forall q \in R$, $dist(p, q) > d$ if and only if $R \cap C(p, d) = \varnothing$.

Proof. Since all objects q are bounded by R and the minimum distance between R and p is $> d$, it follows that the distance of any $q \in R$ will be $> d$. This is the condition for MinDist. ■

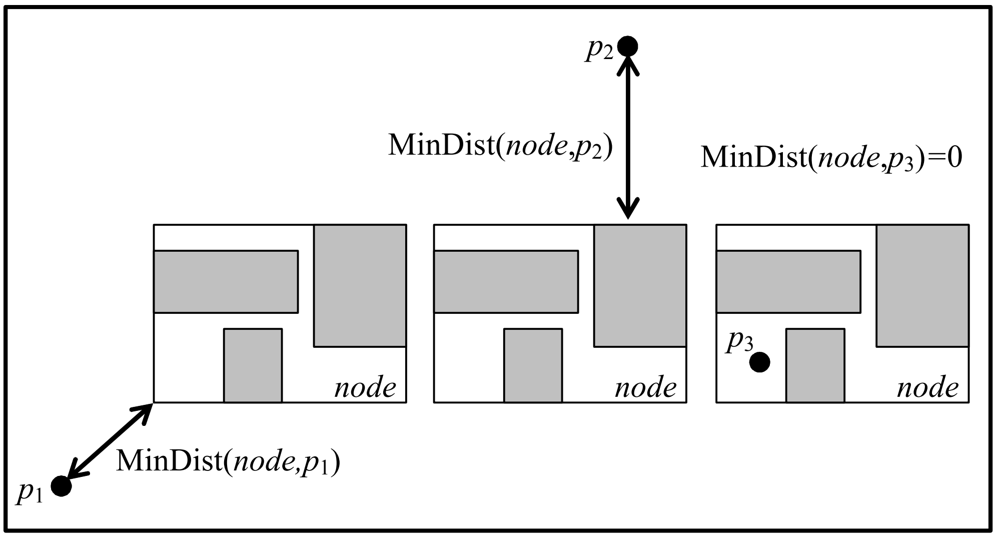


Figure 14. Different cases of MinDist. We illustrate the case where the point lies outside a node (MBR) and within a node

Definition (MaxDist): *Given a point p, and a node in the tree, MaxDist(node, p) is the maximum distance between p and any points contained by the node. We measure the distance of point p from the furthest vertex (corner) of the node's MBR.*

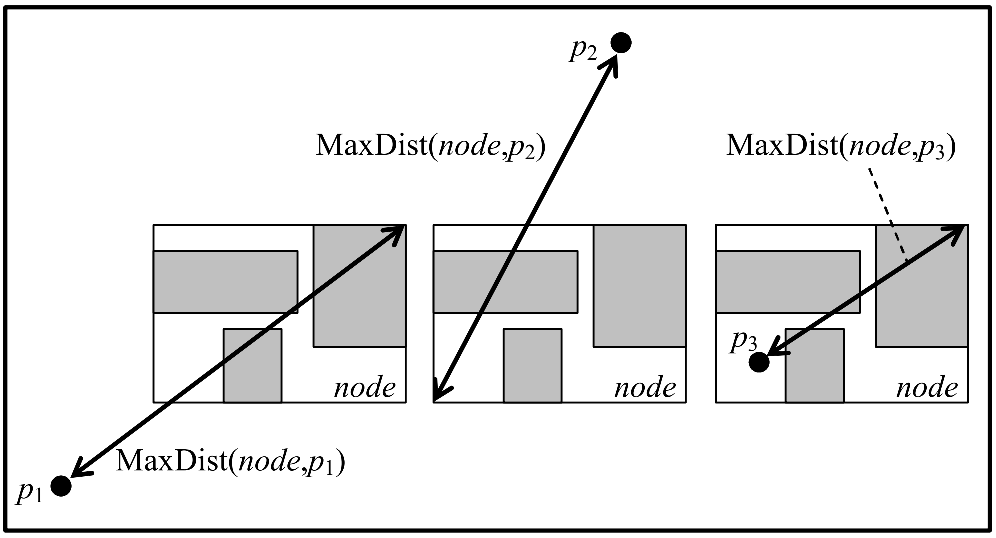


Figure 15. Different cases of MaxDist. The MaxDist is still defined when point *p* lies within a node

The concept of MinDist and MaxDist is illustrated in Figure 14 and Figure 15. These two metrics computed for each query point during query capture its suitability for pruning in the pruning process as a whole. Computing squares is less costly than computing square roots. Thus, in actual implementation, to achieve computational efficiency, the square root function is not used for MaxDist and the last case of MinDist. Instead, the first to third cases of MinDist are squared to keep the metric consistent for comparisons. Mathematically, MinDist and MaxDist are computed as follows. Given,

$$node(x_1, y_1, x_2, y_2),\ p(x, y),\ i \in \{x_1, x_2\}, j \in \{y_1, y_2\}$$

$$\text{MinDist}(node, p) = \begin{cases} 0 & if\ (x_1 \le x \le x_2)\ and\ (y_1 \le y \le y_2) \\ \min(|x - i|) & if\ (y_1 \le y \le y_2)\ and\ ((x \le x_1)\ or\ (x \ge x_2)) \\ \min(|y - j|) & if\ (x_1 \le x \le x_2)\ and\ ((y \le y_1)\ or\ (y \ge y_2)) \\ \min(\sqrt{(x-i)^2 + (y-j)^2}) & otherwise \end{cases}$$

$$\text{MaxDist}(node, p) = \max(\sqrt{(x-i)^2 + (y-j)^2})$$

The computation of MinDist(*node*, *p*) involves more case analysis and was described to great depth in [RoKV95, Chan01]. For computing MaxDist(*node*, *p*), it is very obvious that the maximum distance must occur at one of the vertices (corners) of the MBR and so we shall restrict our considerations to only the vertices of an MBR. This is more closely illustrated in Figure 16.

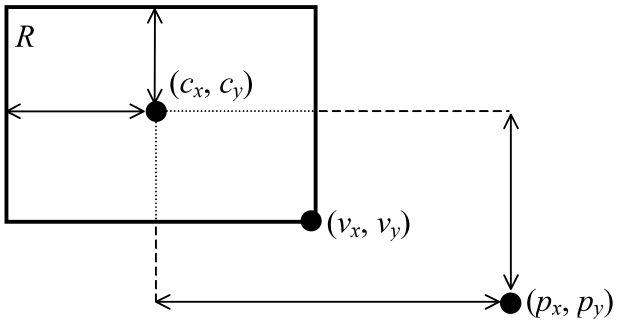


Figure 16. Calculating MaxDist(*node*, *p*) using the point *p*, the centroid *c* and a corner vertex *v* of rectangle *R*

Let $p = (p_x, p_y)$ and let $c = (c_x, c_y)$ be the centroid of the MBR *R*, and $v = (v_x, v_y)$ be any corner vertex of *R*. Then,

$$\text{MaxDist}(R, p) = \sqrt{(|\,p_x - c_x\,| + |\,v_x - c_x\,|)^2 + (|\,p_y - c_y\,| + |\,v_y - c_y\,|)^2}$$

The following lemma forms the basis of MPRQ-MinMax. Lemma 2 postulates that any node satisfying the condition is grey, thus requiring further investigation (downward traversal).

Lemma 2. For any point *p*, MBR *R*, and any object $q \in R$, *R* is grey if and only if $\text{MinDist}(R, p) \leq dist(p, q) < \text{MaxDist}(R, p)$.

Proof. The first part of the condition, $\text{MinDist}(R, p) \leq dist(p, q)$, follows directly from Lemma 1. This means the query region of *p* overlaps *R*. The second part of the condition implies that the largest distance of an object in *R* from p is smaller than R's distance from p. This means that the query region is not large enough to totally cover R. Therefore R is grey. ■

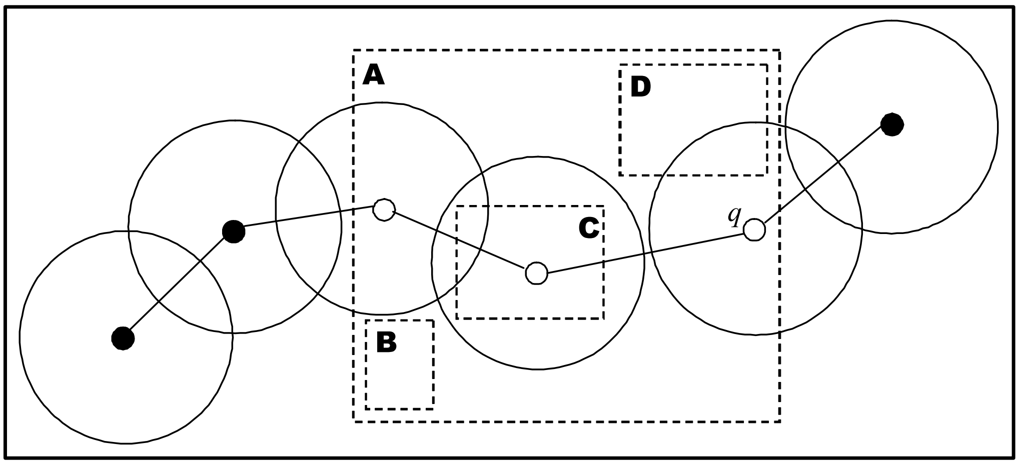


Figure 17. An example to illustrate the pruning rules NodeOut and NodeIn. In this scenario we have MBR A, which contains MBRs B, C and D. The planned route with all the search points and the circular query regions are shown. (Note that in actual case, the boundary of an MBR *tightly* bounds the boundary of its child MBRs)

Pruning Rule 1: NodeOut

In the example of Figure 17, during the index traversal, at a certain point down the R-tree tree we will find that the MBR of the current node (MBR A, which is not an external node yet) partially intersects the query regions. At this point, it contains several children labelled MBRs B, C and D. Some of the children's MBR do not intersect with any query regions. For example, MBR B (a white node) does not intersect with any query regions but MBRs C and D do. Therefore, we can safely ignore MBR B as well as all its children because they will not be among the potential results as their parent is already further away from the query region than allowed. However, MBR C (a black node) is totally contained in the search region and MBR D (a grey node) partially overlaps the search region, therefore we need to traverse down these two nodes in order to be sure.

Pruning MBR B can be done by establishing the condition that MinDist of MBR B to *each and every* query point is never smaller than the search distance, i.e. given planned route Q and search distance d, $\{q \mid q \in Q, \text{MinDist}(B, q) \leq d\} = \varnothing$.

To summarise, pruning rule NodeOut helps avoid traversing white nodes that do not overlap with any query regions. The rationale behind NodeOut is that in most proximity query instances, which are based on a planned route, the search engine can safely ignore all the MBRs that do not overlap with the search regions. In a very vast map, NodeOut quickly helps zoom into the query regions after several iterations of searching. It is imperative to note that NodeOut is achieved by the hierarchical R-tree data structure used to index the spatial data points, and any hierarchical data structures that use the concept of bounding boxes will also work.

Pruning Rule 2: NodeIn

In the example of Figure 17, we see that MBR C is totally contained by the query regions. It clearly shows that the circular search region, which is formed by the radius of search distance, completely encloses MBR C (hence, a black node). Therefore, we can be certain to recursively report all the results in all the children MBRs under MBR C without further MinDist/MaxDist computation and comparisons, right down to the leaf level (done by `FastReport(`R_c`)` in Figure 19). This can be determined as we compute MaxDist of MBR C and find that the MaxDist value is less than or equal to the search distance, i.e. given planned route *Q* and search distance *d*, $\exists q \in Q$, MaxDist(C, *q*) $\leq d$. As for MBR D, the condition MinDist(D, *q*) $\leq d <$ MaxDist(D, *q*) w.r.t. point *q* is true (Lemma 2). Hence it is a grey node and we have to traverse further down MBR D.

To summarise, pruning rule NodeIn helps improve query time by automatically reporting all the results under a node that is completely contained by a query region. The rationale behind NodeIn is that in an instance where the search distance is amply large (for example, modelling it as customer coverage between cities), we can achieve early termination of pruning rules checking and just return all results. This is actually the case in most multi-point range queries of a reasonably large given search distance. The query usually terminates halfway down the search tree, reporting all qualified events correctly.

Pruning Rule 3: PointOut

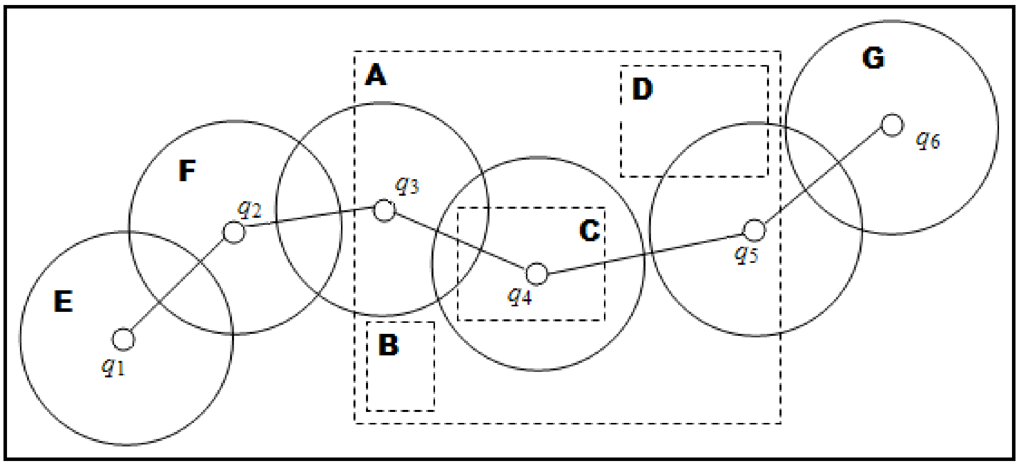


Figure 18. An example to illustrate the pruning rule PointOut. Additional labels are given to the two query regions to the left of MBR A (Regions E and F) and one query region to the right of MBR A (Region G)

In the example of Figure 18, suppose we have traversed down to MBR A. It is obvious that we should not consider the three query points q_1, q_2, q_6 (defined as white points) that define Query Regions E, F, G respectively because they do not overlap MBR A and therefore have no chance of hitting objects under MBR A. This pruning is guided by the computation that MinDist of MBR A from each of the three white points is already greater than the radius of their defined query region. The set of query points found at any level of the tree is always segmented into two mutually-exclusive sets, one in which the query points intersects\ the current MBR (the black points), and another in which they don't (the white points), i.e. for any two sets X and Y where Q is the set of query points, find $X \subseteq Q$, $Y \subseteq Q$ such that $\{x \mid x \in X, \text{MinDist}(\text{MBR}, x) \le d\}$, $\{y \mid y \in Y, \text{MinDist}(\text{MBR}, y) > d\}$, $X \cup Y = Q$ and $X \cap Y = \varnothing$.

Continuing the example, as we consider the children of MBR A, we subsequently prune away MBR B (rule NodeOut) and MBR C (rule NodeIn) and two more query points q_3, q_4 that do not overlap with MBR D (white points w.r.t. MBR D). The power of PointOut lies in that it can quickly shorten the query route length to only the remaining relevant query points w.r.t. the current MBR being investigated.

To summarise, pruning rule PointOut helped improve search time greatly by removing white query points that does not overlap the node being investigated. The rationale behind PointOut is that as the search progresses down the spatial index, the intermediate nodes cover less area than their parent nodes, and hence are representing a (sometimes significantly) smaller defined area. Therefore the chances of a lower level node covering some query points are reduced, and hence we can safely prune away those query points too.

```
MPRQSearch(R, P, d, Obj)
// Input:  MBR R, a query path P, a search distance d
// Output: Obj - set of objects within distance d of some
//         point in P
begin
  if (R is a leaf-node) then
    Process objects in R wrt path P;
  else
    for each R_c of node R do
      PointOut-Rule(R_c, P, d, P_new);    // pruning rule PointOut
      if (P_new <> empty) then
        if NodeIn-Rule(R_c, P, d) then  // pruning rule NodeIn
          FastReport(R_c);              // report all objects
        else MPRQSearch(R_c, P_new, d, Obj);
    endfor
  endif
end; {procedure MPRQSearch}
```

Figure 19. Algorithm for implementation of MPRQ

The algorithm combining the abovementioned pruning techniques is shown in Figure 19. In a nutshell, pruning rule NodeOut avoids traversing nodes that do not overlap with any query point at all. Pruning rule NodeIn reports all events under a node if a query region entirely encloses the node, terminating further search within that node branch immediately. Pruning rule PointOut considers only a subset of the query points when traversing the data structure, effectively pruning query points that are not in the vicinity of the node as we go deeper down the children of each node (as they focus on a smaller area of the map).

3.2 Experiments and Results

In all spatial queries, processing efficiency is the bottleneck. To improve the processing of proximity queries, two main directions can be pursued. Firstly, we could speed up the geometric algorithms in order to answer complex spatial queries efficiently. In MPRQ, there are a significant number of spatial overlay comparisons between MBRs and the query points. To cut down on the number of these comparisons, we introduced some pruning rules without loss of generality. Secondly, we could improve the retrieval time of spatial objects that are handled with spatial access methods (SAM). We experimented with various existing data structures such as quadtrees and R-trees. On top of the data structures, for the R-trees we implemented different node splitting heuristics and bulk-loading (offline packing) algorithms.

Empirically, we compared the performance of various kinds of data structures suitable for implementing the query engine using the MPRQ-MinMax algorithm (simply called MPRQ in experiment results in the remaining of this chapter). We also performed an in-depth study of the effect of applying the various combinations of our pruning rules. We compared the performance of MPRQ against the RRQ for answering proximity queries.

3.2.1 Datasets

The map database used in all experiments as well as the choice of datasets and the combination of experiment parameters are based on four factors, namely (i) the number of event points, (ii) the distribution of event points and the effect of clustering of event points, (iii) the search distance, and (iv) the modes and combination of different types of routes.

We used the RADS database as the underlying source of GIS data to work with. The RADS database, based on the map of Singapore, consists of a collection of geographical objects represented by a series of coordinates. The nature of the RADS database is briefly described in this section in order to more understand the kind of GIS data we used in our applications. The RADS database was widely used in [FLLL99, Lao99, Ho00, NgLH04, NgLe04, NgLe07] for experiments. Firstly, the database represents real-life data comprising the map of Singapore including definitions for landscapes, bus stops, subway stations, roads (partially) and buildings (partially). Secondly, because the RADS database consists of real-life data, we can expect the data distribution to be non-uniform. This provides an opportunity to conduct experiments on hot areas using real-life data.

Definition (Hot area): *Hot areas represent a concentration of activities that lead to a significant number of events within a span of a small area. We represent hot areas with clusters of different intensity, expressed in a percentage of the total events. Events not in the hot areas are randomly uniformly distributed.*

Thirdly, because the real-life datasets are based on Singapore, work can be carried out on a full dataset of one city totally in the main memory. Later, we could further scale up the implementation into external memory to apply the results on a larger city. Experiments were run with various parameters as listed in Table 1.

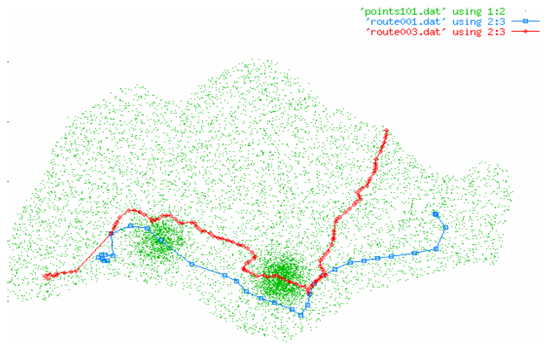
Table 1. The nature of the RADS database that became the primary database for internal main memory experimentations

Number of points (events)	10000, 20000, 40000, 80000, 160000
Clustering and distribution of points (hot area)	• 100% uniform points • 2 clusters (20%, 10%) + 70% uniform • 4 clusters (10% x 2, 5% × 2) + 70% uniform • 8 clusters (8% x 2, 4% × 6) + 60% uniform
Search distance	100m, 500m, 1000m, 5000m
Planned routes	R1, R2, R3, R4, R5, R6

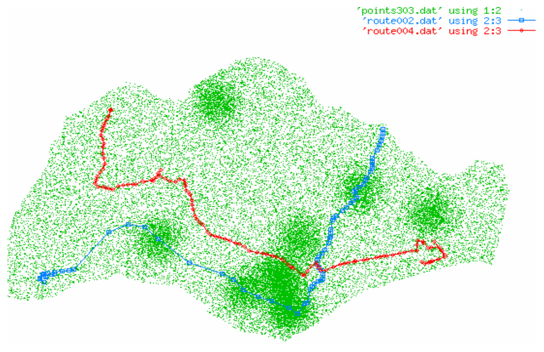
The number of points represents the events that will be available for search at any one time. The number of clusters (0, 2, 4 or 8) represents hot areas that have a concentration of activities. It is realistic to assume at any one time there will be a few places with a concentration of activities. The search distance is representative of short walking to a destination (100m or 500m), a connecting short drive or shuttle buses taken from boarding to alight point (1000m or 5000m).

The query routes (R1, R2, R3, R4, R5, R6) chosen are more subjective. The aim is to simulate different kinds of transportation that a typical traveller may take. R1 is a typical journey using the subway going from the west to the east of Singapore. The stops are generally far away from each other (ranging from 900m to 4800m). R2 is a journey of taking a bus, then switching to the subway, getting off at a hot area (clustered), and continuing the journey on a connecting bus again. The entire journey passes by four hot areas. R3 consists of a long route by buses that pass through a hot area and continuing northbound. R4 is a typical long journey by bus from one end of Singapore (northwest) to the other (southeast), with many stops which are very close to each other (approximately 400m). It passes in between two clusters. R5 and R6 are short journeys (less than 10 stops) which do not pass through any clusters, both at two different parts of the map. They are used as control and correctness measure. Figure 20 illustrates some of the different parameters.

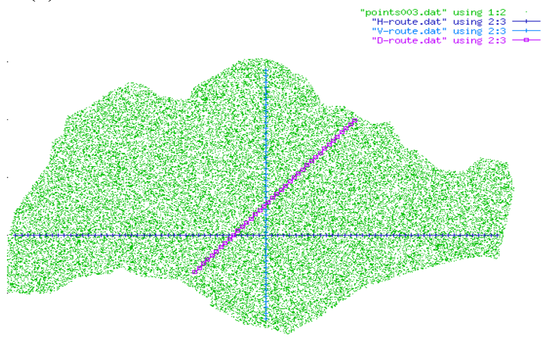
Synthetic query route. A uniformly spaced horizontal route (called *H-path*) with 80 query points (at regular interval 500m apart) is used across all experiments, i.e. |*H-path*| = 80. We also have a vertical query path *V-path* with 38 points and a diagonal query path *D-path* with 45 points. Figure 20(c) illustrates.



(a)



(b)

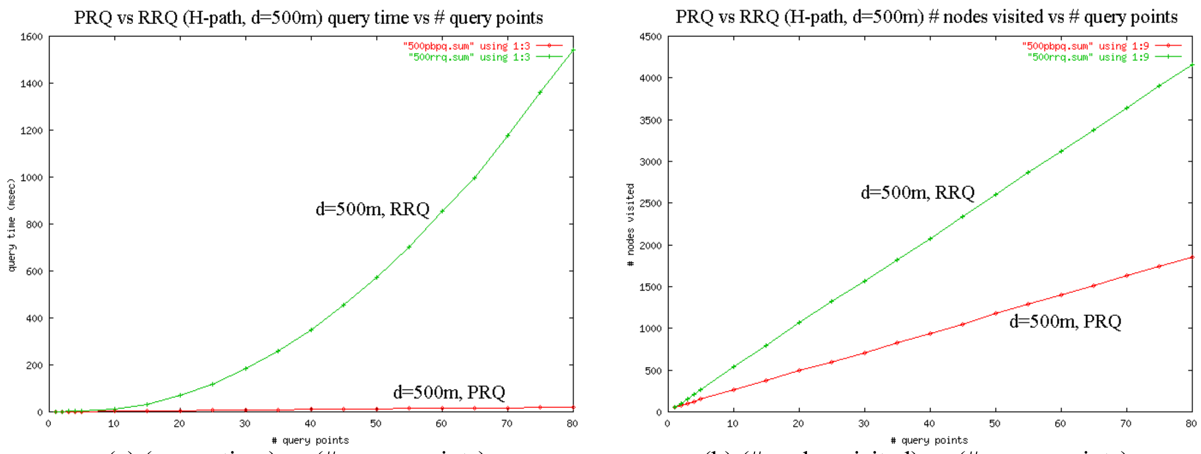


(c)

Figure 20. Graphical representation of the RADS database. The rough map of Singapore is formed by (a) 2 clusters (20%, 10%) + 70% uniform, (b) 8 clusters (8% × 2, 4% × 6) + 60% uniform, and (c) 100% uniform. The percentage specified is the percentage of total points used. In (a), we used two long planned routes, one consists of multiple bus stops and the other is an MRT journey, both passing through a clustered area. In (b), we see one planned route that misses the clustered area and the other goes through many clustered area. In (c), we see synthetic routes with regular intervals called *H-path*, *V-path* and *D-path*

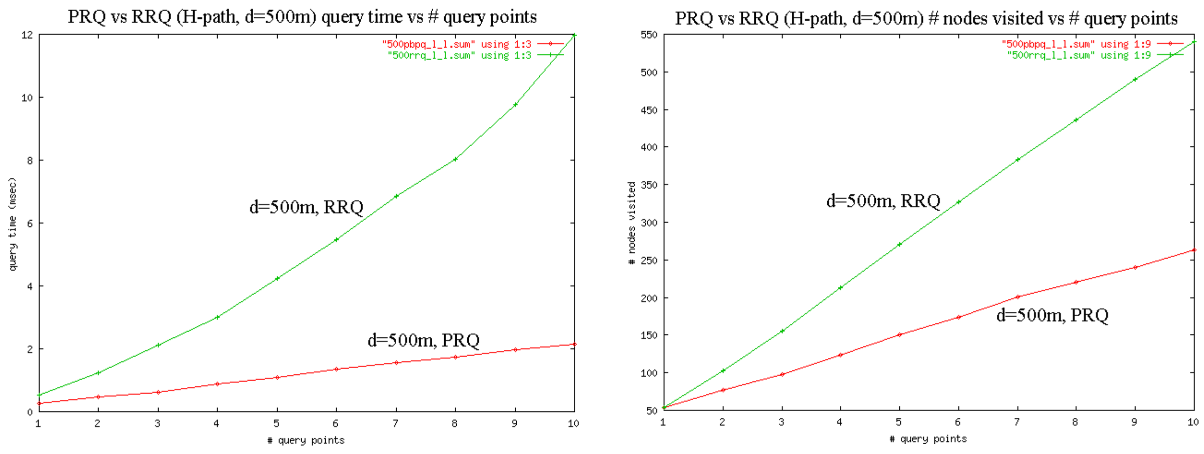
Real-life query route. For one set of experiments, 6 real-life routes (R1 to R6) that are computed by the multi-criteria, multi-modal shortest-path algorithm of [FLLL99, Lao99] are used. The paths contain 34, 78, 120, 123, 11 and 7 query points respectively. The paths exhibit many aspects of a real-life travel plan which can consist of taking buses (query points very near to each other – meaning overlapping is heavy), the subway (points far apart – less incidents of overlapping), and combinations of the two.

3.2.2 Effect of the Number of Query Points



(a) (query-time) vs (# query-points) (b) (# nodes-visited) vs (# query-points)

Figure 21. Comparison of MPRQ and RRQ for query route *H-path* and *d*=500m



(a) (query-time) vs (# query-points) (b) (# nodes-visited) vs (# query-points)

Figure 22. Zoom in on Figure 21 for 1-10 query points

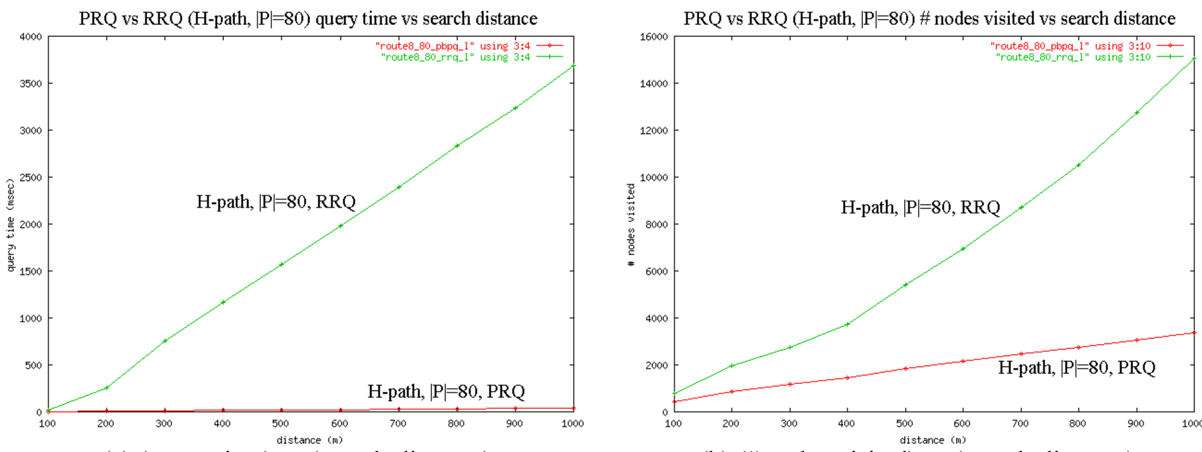
Very naturally, we first compare MPRQ and RRQ across varying number of points in the path (1-80) in the horizontal query route *H-path*. Figure 21 and Figure 22 show the results comparing both the query time and the number of nodes visited for MPRQ and RRQ. It is clear that MPRQ outperforms RRQ. For the Singapore dataset (Figure 21), the query time speed-up is 81 times for $|P|$ = 80; and 6.5 times for $|P|$ = 10. In general, the query time speed-up increases with the number of query points.

The reduction in the number of nodes visited for MPRQ versus RRQ is also significant. For the case of *H-path*, the number of nodes visited rises almost linearly with the number of query points for both MPRQ and RRQ. Figure 21(b) shows that, on average, the number of nodes visited by MPRQ is about 45% and 40.8% of that for RRQ.

In general, we expect MPRQ to perform better when the number of points in the query route *P* increases. Therefore, as a stringent test we have also zoomed into the cases where 1 ≤ |*P*| ≤ 10. The results are shown in Figure 22 and they confirm that MPRQ outperforms RRQ even when there are only two points in the query route *P*.

In addition, our results for the other two query routes *V-path* and *D-path* as well as the real-life NJ dataset (not shown here) also show identical trends with respect to performance comparison between MPRQ and RRQ. So, for the remainder of this study, it suffices to report on results for *H-path*.

3.2.3 Effect of the Search Distance



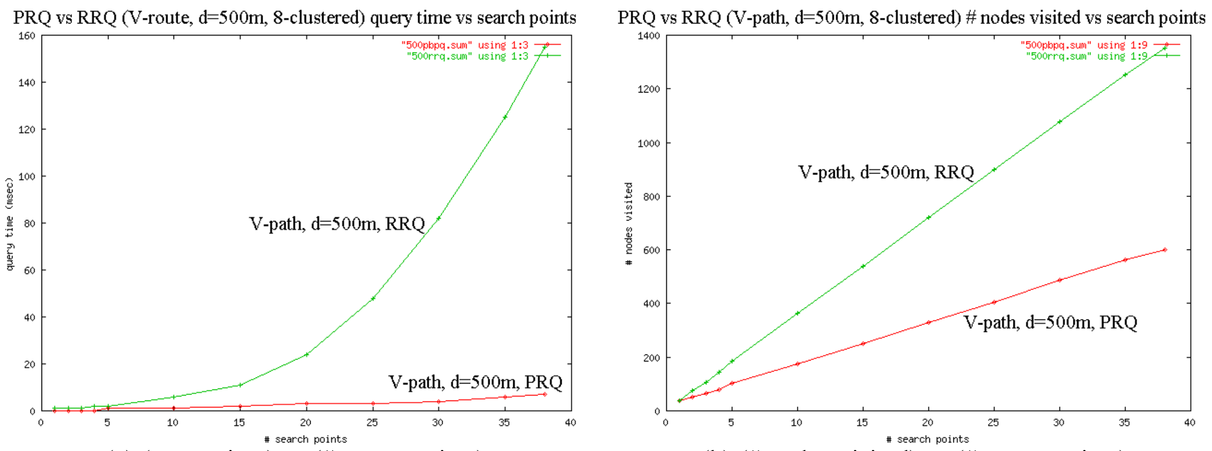
(a) (query-time) vs (search-distance)   (b) (# nodes-visited) vs (search-distance)

Figure 23. Comparison of MPRQ and RRQ for *H-path* with 80 points

We now compare MPRQ and RRQ across different search distances – Figure 23 for the Singapore dataset with |*P*| = 80. The results show that there is a significant speed-up in the query time when using MPRQ as compared to RRQ. In particular, Figure 23(a) shows that the speed-ups in query time (of MPRQ vs RRQ) are 37 times, 82 times, and 97 times for the search distance *d* = 200m, 500m, and 1000m, respectively. The distances represent no overlapping, moderate overlapping and heavy overlapping of query regions.

More stringent tests with very short query route (*H-path* with 1 ≤ |*P*| ≤ 5) and *d* = 3000m showed that the query time speed-up for MPRQ ranges from 2.82 times to 13.38 times. Also, the number of nodes visited for MPRQ (as a ratio of that for RRQ) ranges from 0.40 to 0.63.

3.2.4 Effect of Clustered Dataset



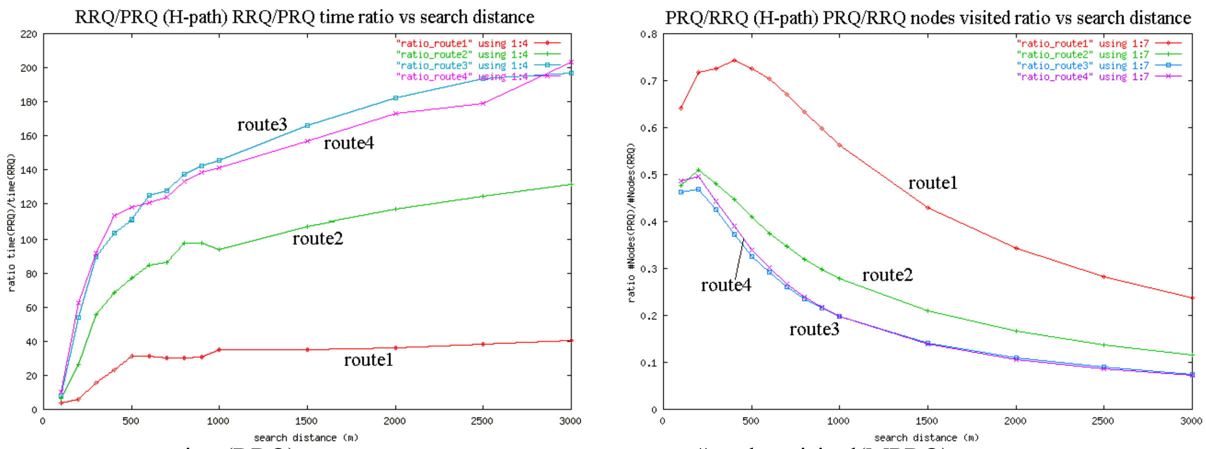
(a) (query-time) vs (# query-points) (b) (# nodes-visited) vs (# query-points)

Figure 24. Comparison of MPRQ and RRQ using clustered data, *V-path* and *d*=500m

We ran MPRQ and RRQ on the clustered datasets to observe the effect of clusters on proximity queries. It is not uncommon for a traveller to travel into and out of a hot area in a journey. For these runs (shown in Figure 24), we use the query route *V-path* that cuts across several clusters of points and *d* = 500m.

Again, it is clear that MPRQ significantly outperforms RRQ. Compared to random dataset, the curves are not as smooth – most likely due to the presence of clusters that cause variations in the results. We can conclude that MPRQ's superior performance holds even for clustered datasets.

3.2.5 Performance of Real-Life Routes



(a) $\dfrac{\text{query-time(RRQ)}}{\text{query-time(MPRQ)}}$ vs (search-distance) (b) $\dfrac{\text{\# nodes-visited(MPRQ)}}{\text{\# nodes-visited(RRQ)}}$ vs (search-distance)

Figure 25. Comparison of MPRQ and RRQ for real-life routes (route1-4)

The performances of the four real-life routes (route1-4) are shown in Figure 25 showing clear advantages of MPRQ over RRQ. We plot the ratio of query time for RRQ over MPRQ. In Figure 25(a), the query time speed-up for real-life routes is generally similar to those for the synthetic *H-path* (shown in Figure 23). The lines show that speed-up continues to rise as search distance increases. In Figure 25(b), the reduction in the number of nodes visited for MPRQ widens with the search distance.

3.2.6 Performance of Data Structures

Extensive experiments were performed using datasets described in the previous section on various data structures to measure the performance of data structures against MPRQ. Two metrics were used as measurement for the various data structures, namely query time per point and memory used per node. Both are calculated as follows.

$$\text{Average query time} = \frac{\text{Total query time}}{\text{Size of spatial database}}$$

$$\text{Average memory used} = \frac{\text{Total memory used}}{\text{Total nodes in tree}}$$

The PR quadtree was first investigated using varying bucket size and maximum tree depth. The bucket size determines the maximum objects stored at the leaf level before it overfills and be split into two. The larger the bucket, the better the utilisation. The maximum tree depth is imposed to prevent from getting a narrow, skewed and chain-like tree. Typically, an events-based GIS database can contain data points that share the same exact location (e.g. an exhibition event at a convention centre, or many different companies located inside a high-rise building). A bucket implementation effectively keeps the points (internally) together in the resulting tree node. Results are shown in Table 2 and Table 3.

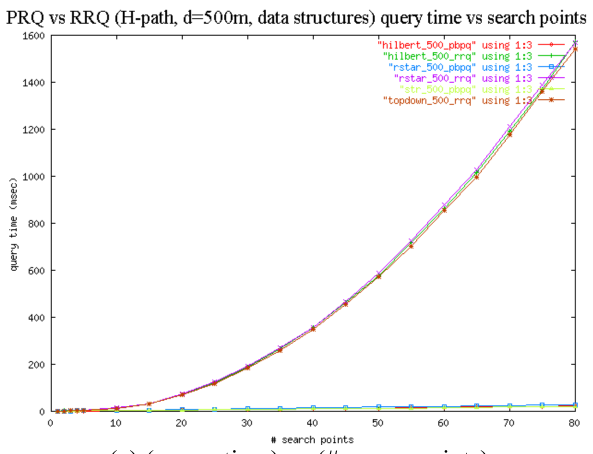
Table 2. The average search time in milliseconds of the PR quadtree implementation with various bucket sizes and maximum tree depths limited to various depth levels

Average Query Time (ms)		Maximum Tree Depth					
		9	12	15	18	21	24
PR Quadtree		0.419	0.321	0.309	0.318	0.360	0.388
Bucket Size	2	0.431	0.296	0.292	0.317	0.329	0.351
	4	0.279	0.274	0.291	0.299	0.292	0.294
	8	0.236	0.236	0.234	0.253	0.240	0.253
	16	0.228	0.208	0.228	0.227	0.209	0.206
	32	0.204	0.206	0.197	0.218	0.212	0.203
	64	0.183	0.184	0.197	0.189	0.198	0.201

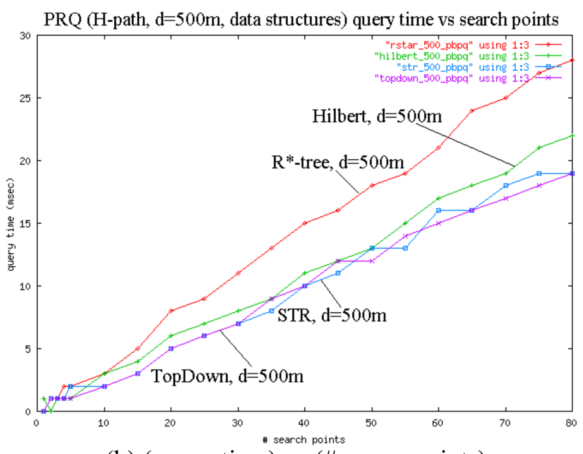
Table 3. The average memory used per node in bytes of the PR quadtree with various bucket sizes and maximum tree depths limited to various depth levels

Average Memory Used (bytes)		Maximum Tree Depth					
		9	12	15	18	21	24
PR Quadtree		40.2	48.2	62.0	81.7	104.8	128.5
Bucket Size	2	40.3	48.0	59.8	75.6	93.0	109.3
	4	47.1	59.5	73.0	83.9	91.8	96.9
	8	61.3	70.9	75.5	77.9	78.5	78.8
	16	66.2	67.2	64.5	65.0	67.7	67.7
	32	61.8	61.9	61.9	61.9	61.9	61.9
	64	59.6	59.6	59.6	59.6	59.6	59.6

For the PR quadtree alone without buckets, the average query time is best when a depth of 15 is used. The average memory used per node is increasing proportional to the tree depth. For the PR quadtree with varying bucket size, the search time improves as a larger bucket is used. The best time is achieved when the maximum depth is set at 15. Since the bucket is a linked list, the larger the bucket the longer it takes to search through it. Hence a bucket which is too large will adversely affect query time (the time saved from using the quadtree hierarchical structure cannot compensate for the time spent on searching buckets). As for memory used, there is more or less no difference when a larger bucket is used.



(a) (query-time) vs (# query-points)



(b) (query-time) vs (# query-points)

Figure 26. Different R-tree data structures: HilbertPack, R*-tree, STRPack and KDTopDownPack. (a) comparison of MPRQ and RRQ for *d*=500m, (b) showing MPRQ only for *d*=500m

Table 4. The average search time in milliseconds of various implementations of node splitting heuristics and R-tree bulk-loading algorithms with various bucket sizes

Average Query Time (ms)		Bucket Size				
		8	16	32	64	128
R-tree variants	QuadSplit	0.508	0.434	0.731	0.975	1.201
	LinearSplit	0.448	1.094	1.597	1.376	1.862
	NewLinearSplit	1.031	0.843	0.741	0.790	1.228
	R*-tree	0.418	0.564	0.569	0.718	1.122
	HilbertPack	0.410	0.563	0.775	1.215	1.468
	STRPack	0.354	0.414	0.565	0.750	1.004
	KDTopDownPack	0.112	0.136	0.174	0.252	0.384

As for the R-trees variants shown in Figure 26 (data shown in Table 4 for clarity), for the average query time, it is observed that as bucket size increases to 64, some on-line algorithms like NewLinearSplit [AnTa97] and R*-tree perform comparably with STRPack which is an off-line bulk-loading algorithm. Generally, STRPack performs very well at different bucket sizes compared to the rest except the KDTopDownPack. The latter is much faster than the former due to its design whereby it splits the search space to be disjoint at each level and further partition and pack each node down from the root level. At all levels, KDTopDownPack computes the axis major for any subset of rectangles it is about to organise, and uses the axis major to split its rectangles, resulting in well-divided, balanced area on its children. Our result for HilbertPack is also in line with [HKCL03] who conducted a performance study of main-memory R-tree variants.

Table 5. The average memory used per node in bytes of various implementations of node splitting heuristics and R-tree bulk-loading algorithms with various bucket sizes

Average Memory Used (bytes)		Bucket Size				
		8	16	32	64	128
R-tree variants	QuadSplit	101.3	94.0	90.4	88.6	87.3
	LinearSplit	101.4	94.3	90.8	89.5	89.6
	NewLinearSplit	96.8	89.6	86.6	85.5	85.2
	R*-tree	97.0	90.2	87.2	85.8	85.5
	HilbertPack	97.8	93.0	90.9	89.8	89.4
	STRPack	43.7	39.7	38.1	37.8	37.4
	KDTopDownPack	45.2	40.3	38.5	37.3	36.7

Table 5 shows the average memory used by the R-tree variants. Generally, average memory used decreases with larger bucket sizes until it stabilises. The STRPack and KDTopDownPack are close to each other on this term, while the rest use about 3 times greater memory. This is due to both trees being packed to the brim even from the start, exhibiting good utilisation of buckets. When taking average query time into account, the proposed KDTopDownPack is definitely the outstanding one. Because of this, we use it as the default data structure for MPRQ.

3.2.7 Effectiveness of Pruning Rules

Using our experiment datasets, we next investigate the effectiveness of the pruning rules that we introduced. We applied the pruning rules (described in Section 3.1.3) incrementally and ran separate rounds of experiments. For easy referencing, they are summarised as follows.

NodeOut. Avoid traversing nodes that do not overlap with any query point.

NodeIn. Report all events under a node if some query region contains it.

PointOut. Consider only a subset of query points while traversing the index.

We start by applying rule NodeOut to the R-tree as a baseline for our experiment, then apply NodeIn and PointOut incrementally and we measure the difference in performance in terms of time taken to answer an MPRQ. NodeOut was picked as the baseline because it is considered to be a pruning technique implicitly derived from using a hierarchical data structure alone. Table 6 summarises the effect of these pruning rules.

Table 6. The effectiveness of applying different pruning rule combinations. NodeOut was used as the baseline. The percentage value represents the time taken for answering the multi-point range query. In interpreting the results, we used the mean running time

	Percentage compared to applying pruning rule NodeOut as baseline measurement			
Pruning Rules	min	mean	median	max
NodeOut	100.0%	100.0%	100.0%	100.0%
NodeOut +NodeIn	45.3%	88.9%	93.3%	102.2%
NodeOut +PointOut	0.2%	8.1%	3.0%	41.2%
NodeOut +NodeIn+PointOut	0.2%	5.8%	2.8%	34.3%

We observed that by adding pruning rule NodeIn on top of NodeOut, we obtain a 11.1% decrease in query processing time. By applying PointOut, we obtain a further 83.1% decrease in processing time (only 5.8% of the original time). On the other hand, applying pruning rule PointOut on NodeOut results in 91.9% cut in query processing time, while applying NodeIn further results in another 2.3% reduction.

In general, we note that pruning rule PointOut helps in reducing more of the query processing time over pruning rule NodeIn. This is because a route might spread across a wide area (possibly the entire search space), and hence, eliminating irrelevant query points that do not affect the computation of overlapping operation greatly helps in reducing the processing time by cutting out a lot of branches that need not be traversed. On the other hand, pruning rule NodeIn only helps when the search distance is relatively larger than the areas covered by the nodes in a data structure, pruning thus takes place near to the leaf level (with smaller node coverage) of the tree.

3.2.8 MPRQ vs Traditional Query

It is interesting to know how well our pruning techniques perform in comparison to the traditional method, which is the RRQ. We conducted experiments to show its merits in performance (running with all three pruning rules activated) in contrast to performing query processing using traditional methods, i.e. performing the query point-by-point and combining the results (with duplicates removed in post-processing). Using MPRQ-MinMax, we only need to perform the search once with the all the points in the route as one input set (incorporating the pruning rules that we described).

Table 7. The average query time in milliseconds comparison of various implementations of node splitting heuristics and R-tree bulk-loading algorithms between the multi-point range query and the traditional repeated range query

Average Query Time (ms)		Query Type		Improvement
		MPRQ	RRQ	
R-tree variants	QuadSplit	700.375	5128.708	7.323
	LinearSplit	733.458	5061.958	6.901
	NewLinearSplit	693.396	5120.938	7.385
	HilbertPack	673.875	4932.604	7.320
	STRPack	640.500	4850.438	7.573
	KDTopDownPack	609.040	4811.710	7.900

Table 7 shows that MPRQ outperforms RRQ by as much as 8 times (when using KDTopDownPack). The reason behind this is because MPRQ can prune away the nodes that do not overlap the combined search regions of all search points at all. Once the MPRQ found a node that is contained in one of its query regions, it will immediately report everything under that node and not consider that branch anymore, saving valuable computational time.

On the contrary, RRQ traverses down the data structure once for each and every search point in the whole path sequentially, oblivious to neighbouring search points. The spatial index is traversed as many times as the input route size, with the possibility of traversing down the same nodes at the top of the tree each time (if two consecutive query points are close to each other). Even in the case of internal memory here, which does not incur disk

I/O costs which is 2 orders of magnitude higher, the RRQ loses to MPRQ. This is true despite MPRQ having to perform more computations than RRQ due to the complexity of its pruning rules.

3.3 Summary

Several spatial data structures used in indexing objects in isotropic search space were explored. These are data structures that support space decomposition, i.e. by dividing search space until it is small enough to accommodate an individual object. We implemented and investigated PR quadtree, kd-tree, R-tree and its many variants (different node splitting techniques and R-tree bulk-loading algorithms) on their performance in multi-point range queries.

Experimentation results showed that our KDTopDownPack bulk-loaded R-tree (refer to [Ho00] for details) outperforms other spatial data structures in terms of memory use and query time. For example, it outperforms the PR quadtree by one-third in query processing time with about one-fifth savings in memory used for the data structure. Compared to other R-tree variants, KDTopDownPack took only half the time to answer the same query although the memory used is slightly more than its closest rival the STRPack R-tree. Therefore, we decided to focus on KDTopDownPack R-tree as our base data structure in all MPRQ implementations.

MPRQ is a special type of query characterised by a series of points that represents a travelling route. We explored decomposition issues [KrSB93] with regard to multi-point range queries. We had looked into the issue of object decomposition (using pruning rules), where geometric tests are only applied to a representative of the object which is more efficient than testing for the whole object [KrHS91, NiWi97, SeKr98, SCRF99]. We arranged the events into regions before indexing them with a spatial data structure so that objects are organised with respect to their location as proposed by [SeKr90]. This forms the implementation of object and space decomposition. Our experimental results show that the three pruning rules combined are very effective in cutting down query processing time.

As a summary, this study has addressed many issues in dealing with the multi-point proximity range query, enough to develop an initial prototype incorporating the right data structure and pruning algorithms for a small database. We had expected MPRQ-MinMax to perform well for small route queries, and our empirical studies have proven it. In the next chapter, we explore MPRQ for the case of very large database involving external memory.

Chapter 4 External Memory Algorithms for MPRQ

Often in dealing with spatial databases, even the smallest dataset may be too large to reside in internal memory. Not many systems can afford the luxury of having very fast processors with lots of internal memory. In 2000, the cost of memory (dollar per MB) stands at US$0.30 compared to about US$40 back in 1990. Even so, there is the problem of not enough space and heat in packing too many transistors per square inch into a memory module. The largest memory module today for the personal computer is 4GB RAM module. New memory technologies with low power consumption called flash memory (such as CompactFlash, Secure Digital or MultiMedia Card), commonly used in PDAs and handheld devices, are also comparable in price to standard memory.

Even with such a drastic drop in memory cost, the idea of processing large amounts of data on external storage with a small amount of internal memory is still unfathomable. A typical GIS database size is in the range tens of terabytes (10^{12} bytes). There are 23×10^{9} billion indexed web pages in the world as of May 2010, according to Google. Due to the sheer size alone, it is inescapable for proximity querying applications to commonly deal with large amounts of data stored in secondary memory.

We know that the cost of disk accesses is relatively much more expensive than its internal memory cost counterpart even on a single workstation, let alone distributing the data components across the globe using wide area networks (WAN), which is the common practice in today's globalised world. It is not uncommon to perform spatial joins of spatial databases from different spatial data centres located in different geographical locations worldwide.

4.1 External Memory Experimentation Systems

In experimentation, the research world is divided and there are not any canonical programming methods or platforms in which research on disk-based algorithms are done. As disk access activities such as data-to-disk mapping, actual I/O calls, data buffering, caching and I/O timing and accounting are not standard to any programming language (C, C++ or otherwise) or operating system, it is generally very difficult to implement an experiment that conforms to the design of a sound disk-based experimentation framework. We do not wish to mix the algorithm part of our application with the presentation logic and the disk-access logic in one program. This would greatly increase coupling, which is not desirable.

Table 8. Different software components widely used for research in the performance of external (secondary) memory data structures and algorithms

Packages	Comments
MPI	General message-passing routines; supports only C (not C++) and Fortran
PVM	Comprehensive message-passing routines, widely used in educational and commercial applications
TPIE	Created to support parallel I/O systems research, many research papers around
LEDA-SM	Relatively new, provides basic external data structures like lists, stacks, queues, arrays
STXXL	An implementation of the C++ standard template library for out-of-core computations

We researched some of the high-level packages supporting disk access that are freely available for research. They are generally divided into two categories, (i) those that facilitate computer to computer communication via message packets; and (ii) those that provide a templated data structure that simulates disk access and its fundamental methods (e.g. insertion and deletion for a disk-based data structure implementation).

In Table 8, the first two packages, MPI (Message Passing Interface) [GrLS94, MaDo94] and PVM (Parallel Virtual Machine) [Sund90, GBDJ94] belongs to the message passing category. TPIE (Transparent Parallel I/O Environment), LEDA-SM (Library of Efficient Data Structure for Secondary Memory) [CMAB98, CrMe99] and STXXL (Standard Template Library for Extra Large datasets) [DeKS05] belongs to the access-oriented library for I/O implementation. After investigating, we chose TPIE for MPRQ.

TPIE (Transparent Parallel I/O Environment) is a framework-oriented approach for development of I/O codes. TPIE [Veng94] is a set of templated classes and functions that facilitates the implementation of external memory algorithms. The whole process of reading data, processing and writing them back to disk is abstracted out by TPIE into a continuous process where the program is fed data mapped from an outside source (physical disk drives), reads into and writes data from it. The underlying details of how I/O is performed on a particular machine/platform are handled by TPIE, as well as the associated accounting such as time and memory used and I/O operations performed. Each disk D is a simulated stream of objects that resides on disk as a file. Continual support by the developer with release of newer stable versions and ease of deployment make TPIE a good choice.

4.2 Porting MPRQ to Disks

To run new experiments on external memory with the previous sets of MPRQ codes, some modifications are in store to implement the chosen disk-based programming library, the TPIE. First of all, a realistic disk block size *B* is chosen. This parameter is often called the page size in the literature. Each I/O will consist of a read/write operation of size *B* that we want to simulate a true logical block of disk access in the underlying operating system. This is done by modifying the `BTE_MMB_LOGICAL_BLOCKSIZE_FACTOR` value inside the app_config.h file. The AMI (Access Method Interface) is one of the three components of TPIE and is the only one that we have to interact with. The settings for our implementation are as follows.

`BTE_IMP_MMB` – we used memory mapped block transfer engine, where each disk *D* is implemented as one ordinary file in the Unix file system. This paradigm transparently maps the currently accessed block of a file to internal memory. When a node is outside the mapped block, the current block is unmapped (saved to the disk if modified) and a new one (a node that is requested) is mapped from the disk. This is equivalent to one I/O operation. Logical disk block size factor = 4 – we used the LDBS of 4 * O/S block size. We ran experiments on Linux, whose file system's default block size is 4096 bytes, the size of one inode (the basic building block at file system level).

When a node is accessed, its disk_index is retrieved and a seek is made to the disk location before a read or write operation takes place. A seek outside the current logical block mapped into memory will cause an unmap operation of the current block and a map operation to the new block. So if the next node to access is physically nearby, a read/write will not generate an I/O operation, as it is done within the current block.

Each node *R* has a block id which consists of the bucket size (fan out) number of R_c (*R*'s children), each with their own block id. Largely, the handling of block ids is synonymous to the handling of pointers in main memory. In the MPRQ algorithm, the spatial data structure is implemented with a `struct node` which contains up to *f* (fan out) pointers to other `struct node` objects, at the leaf level in which they are cast to point to real spatial data points. In the MPRQ algorithm modified for disk, henceforth we refer to as MPRQ-Disk, nodes are stored in blocks that contain links (block ids) to other nodes (blocks).

In previous experiments in internal memory, three metrics were used as a measurement for the various data structures, namely the average query time, the number of nodes visited during query and memory used per node. When focus is shifted to disk-based accesses, we are interested in the amount of disk I/Os that each data structure uses during operation. This figure should be minimised and optimal to the data on disk with respect to locality. Since the cost of a disk I/O operation is several orders of magnitude greater than a memory access, our algorithm running time is clearly dominated by it. In the literature, disk I/Os are commonly used for upper and lower bound analysis of algorithm performance.

Measurements are obtained from the statistics provided by TPIE at the end of each experiment. TPIE provides statistics on the number of reads, writes, maps, unmaps and seeks. The number of I/Os performed by an algorithm is given by TPIE's count of the number of map operations performed, which is fully documented. This is also the measurement used in the research done in [AHVV99].

Two issues to look at are locality of reference and the amount of data to access in a single block of I/O operation. The data exchange between the internal memory and the external memory is one logical block at a time. Locality of reference means access to all data inside this block takes about the same time as accessing a single item within the block, because a chunk of data is read or written in one I/O. The question is how large a chunk (logical block) should be used. Many researchers go by LDBS = 4 [KrSB93] because that is the default block size in the Windows and Linux file system platform, the size of one cluster in FAT32 up to 8GB per partition.

In our study of MPRQ in internal memory, the issue of performance was largely dominated by the effect of pruning of the nodes visited during the tree traversal. The performance speed-up of the MPRQ algorithm is more or less directly proportional to the number of nodes visited during the query process. However, when porting MPRQ to disk, several different performance issues needed to be studied – for instance, as disk block reads are typically much slower than internal memory access, the number of disk I/Os becomes the critical factor in performance. Since each disk block contains nodes of R-tree, issues such as disk block size and disk buffering greatly affect the performance of MPRQ-Disk.

For the disk case, given the size of spatial database N, the size of disk block B, at any node R in the R-tree, MPRQ-Disk now incurs O(m*B) time for each node, where m is the number of query points and $f \leq B$ is the fan out of node R. The former, m, is mostly internal CPU computation where the pruning rules of NodeIn and PointOut take place. Since disk accesses are generally 2-3 orders of magnitude slower than CPU computations, B becomes dominant and it contributes to the bulk of the query time and the total number of I/Os. In comparison, the RRQ has a processing time of O(f) per node in internal memory, but O(B) on disk. In general, RRQ-Disk answers MPRQ in $O(m^*(\log_B N + k/B))$ using bulk-loaded R-trees (such as KDTopDownPack) which guarantees a bounded height of O(log n), where k is the number of results found. There is also a post-processing cost of O(K log K) to remove duplicates, where $K = \sum_{i=1}^{m} k_i$. In comparison, MPRQ-Disk answers MPRQ in $O(\log_B N + k/B)$.

```
MPRQ-Disk-Search(Bid, P, d, Obj)
// Input:  Disk block ID Bid, query set P, search distance d
// Output: Obj - set of objects within distance d of
//         some point in P
begin
  Access block Bid for node R;
  if (R is a leaf-node) then
    Process objects in R wrt path P;
  else
    for each Rc of node R do
      PointOut-Rule(Rc, P, d, Pnew);    // Pruning rule PointOut
      if (Pnew <> empty) then
        if NodeIn-Rule(Rc, P, d) then // Pruning rule NodeIn
          FastReport(Rc.Bid);    // report all objects under Rc
        else MPRQ-Disk-Search(Rc.Bid, Pnew, d, Obj);
    endfor
  endif
end; {procedure MPRQ-Disk-Search}
```
Figure 27. Algorithm for MPRQ-Disk

4.3 MPRQ Algorithms

4.3.1 Algorithm 3: MPRQ-Sorted Path

With a larger database, we need to have more efficient pruning methods. The sorted path approach, which we shall call MPRQ-SP [NgLe04], sorts the input query route along the major axis and takes advantage of the fact that the query points are always sorted to quickly prune the path with respect to the MBR being processed. This technique involves making slight changes to the MPRQSearch algorithm of Figure 19 and is easy to implement. The main difference is in the way the two pruning rules, PointOut and NodeIn, are implemented. Hence, only the corresponding algorithms for these two pruning rules are presented. Other algorithms that are needed to provide supporting roles but are common, such as sorting and binary search, are omitted.

MPRQ-SP prepares the planned route *P* for ease of pruning by sorting all the query points according to its major axis before the query begins. While traversing down the R-tree, to eliminate all the white points in *P* w.r.t. a node, it is now possible to quickly find the cut-off left and right end of *P* to extract a shorter sorted sub-path for passing down to that child node. In general, instead of evaluating *n* points for any given node and path, it now suffice just to evaluate O(log *n*) points, providing substantial savings especially when the size of input query is very large.

MPRQ-SP has three major steps: determine the axis major of the input path *P*, rearrange (sort) all the points in *P* along the axis major, and finally begin query with the sorted path *P*. The first step is straightforward, and it involves scanning *P* to determine whether the path *P* is more horizontally or vertically inclined. This process can be achieved in O(*n*) time. Then all the points in *P* are sorted according to the axis major of *P*, i.e. the longer of the two axes. Sorting takes O(*n* log *n*) time. The second step is akin to mapping all the points to a one-dimensional structure for easy search. The first two steps are the pre-processing steps for MPRQ-SP.

Figure 28 illustrates the example where all the points are sorted using their x-values or y-values, depending on their axis major. The intuition to use the axis major stemmed from the fact that if we cut *P* along the axis major, more points can be pruned easily as *P* spreads out more along the axis major compared to the axis minor. Refer to Table 9 for the running time of MPRQ-SP.

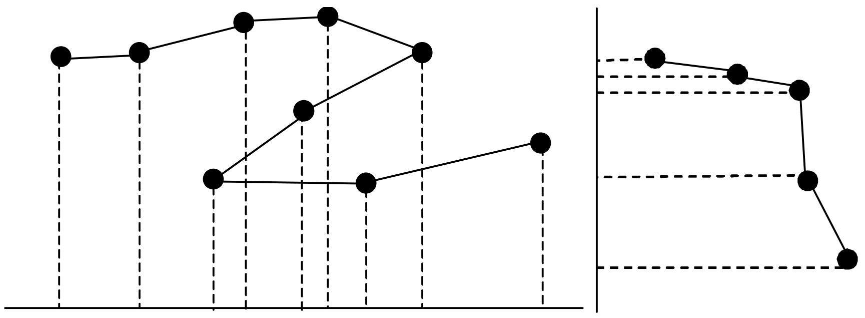


Figure 28. Sorting the query points in route *P* along the axis major

The third final step is the searching. We present the PointOut and NodeIn algorithms for MPRQ-SP in Figure 30 and Figure 31 respectively. The difference for these two algorithms is that they both accept a sorted input route *P* rather than any input route *P*. The algorithms make use of binary search to locate along the sorted path the cut-off points for extracting a shorter pruned route (defined as black points) which is by default also sorted. Hence, no more sorting is necessary throughout this step to maintain a sorted sub-path. Two binary search routines appeared in the PointOut pruning rule, `left_bsearch` and `right_bsearch`.

It is noted that they are modified versions of the standard binary search routine which, without loss of generality, also give us the cut-off point in O(log *n*) time. Their names suggest that they are either left- or right-biased. A direction-biased binary search, say right-biased, in this case means the routine is able to proceed towards searching right as long as the pivot is equal to or less than the target value but stops when the pivot is greater than the target value. The rationale behind biased binary search is to handle points that map to the same location along the axis major. In the example of Figure 29, the right-biased binary search will terminate and return the index to point p_i but not p_j since the latter is greater than the right side of the MBR R'. The same principle applies to the left-biased version.

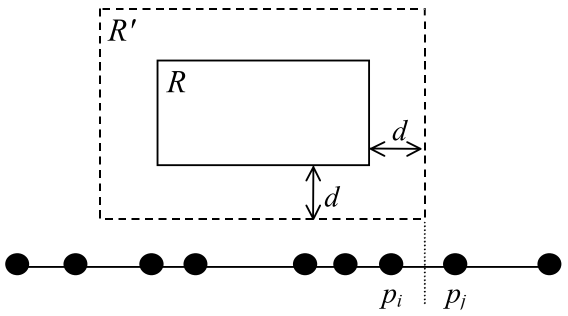


Figure 29. `right_bsearch` returns the point on path *P* along the sorted axis that is less than or equal to the right edge of the "augmented" MBR *R′*

```
MPRQ-SP-PointOut(R, P, d, P_new)
// Input:  MBR R, a sorted query path P, a search distance d
// Output: P_new - sorted query subpath of P
begin
  lo ← left_bsearch(R_sorted-axis.lower - d, P)
  hi ← right_bsearch(R_sorted-axis.upper + d, P)

  forall points pt in P[lo, hi] do
    if R_non-sorted-axis.lower-d ≤ pt_non-sorted-axis ≤
       R_non-sorted-axis.upper+d then
      P_new ← P_new ∪ {pt}
    endif
 endfor
end; {procedure MPRQ-SP-PointOut}
```

Figure 30. Algorithm for the MPRQ-SP PointOut pruning rule

```
MPRQ-SP-NodeIn(R, P, d)
// Input:  MBR R, a sorted query subpath P, a search distance d
// Output: Obj - set of objects within distance d of some
//         point in P
begin
  midpt ← (R_sorted-axis.lower + R_sorted-axis.upper) / 2
  mid ← right_bsearch(midpt, P)

  if GetMaxDist(P[mid], R) ≤ d then
    return true
  elseif GetMaxDist(P[mid+1], R) ≤ d then
    return true
  else
    return false
  endif
end; {procedure MPRQ-SP-NodeIn}
```

Figure 31. Algorithm for the MPRQ-SP NodeIn pruning rule

In Figure 31, the NodeIn pruning rule uses at most two MaxDist computations because the fact that the query route is sorted allows us to ignore all the query points in *P* except the ones closest to the center of the MBR. This guarantees a constant time NodeIn processing with respect to the size of the route. By NodeIn rule, in order to determine if a given MBR *R* is black, we need to show MaxDist(R, p) $\leq d$. The goal is to find the $\min_{p_i \in P}\{\text{MaxDist}(R, p_i)\}$ that gives the smallest *d*. We start by dividing *R* into four quadrants with the centre *C* of *R*, and it follows that all the points that lie in one quadrant will produce MaxDist when paired with the opposite diagonal corner of *R*. This is illustrated in Figure 32. Since all the points p_i are sorted along an axis, say the x-axis, the point that would give the smallest *d* would be nearest to *C*. Therefore, we can utilize the right-biased binary search tree to give us the point p_l to the left of *C*. To cover the right half of *R*, we can immediately derive the next point p_r on our sorted path. Suppose MaxDist(R, p_l) > d and MaxDist(R, p_r) > d, because the points in *P* are not sorted on the other axis, there may exists a point $p_{i\,(i \neq l \neq}$

$_{r)}$ that may give MaxDist(R, p_i) < d. Even so, we are content that the NodeIn pruning rule will be invoked at one level lower down the R-tree.

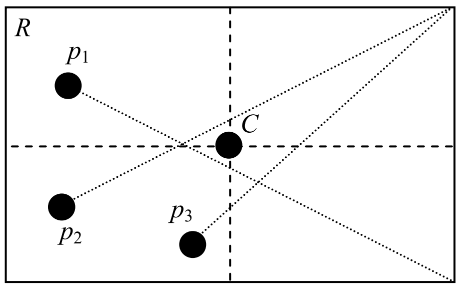


Figure 32. The MaxDist(R, p) is given by the distance of p to the opposite diagonal corner of MBR R from the quadrant where p lies. The quadrant where p lies is determined by the centre C of MBR R

4.3.2 Algorithm 4: MPRQ-Rectangle Intersection

The rectangle intersection approach, MPRQ-RI [NgLe04] for short, transforms the input query route into a set of rectangles to be solved as the two-set rectangle intersection problem, the set of child MBRs of the investigated node being the other set of rectangles.

In this approach, the key idea is to transform all the points in P into a collection of rectangles, say R_1, and find all the intersections between them and the current collection of child rectangles of the current MBR being investigated. The intuition for this approach is that once we reach a certain MBR, all its children are already visible so we actually could make use of all of them at the same time.

Rectangle intersection problem is a well-defined research problem [PrSh85], which is defined as given a collection of N orthogonal rectangles, report all the intersecting pairs. The standard approach to solve this problem is by plane sweeping, i.e. scanning a sweep line horizontally across the plane, inserting or deleting a rectangle's left edge into an event point schedule as the sweep line enters or leaves a rectangle respectively. When a new rectangle is encountered, we perform an interval query of all the current intervals in the schedule with the new rectangle's left edge interval, and report all intersections. The running time is O(N log N + k) where k is the number of rectangle pairs reported, with pre-processing time of O(N log N) to prepare the sweep schedule and the space complexity is O(N). Other approaches to the rectangle intersection problem exists, such as by using the divide-and-conquer method [GüSh87].

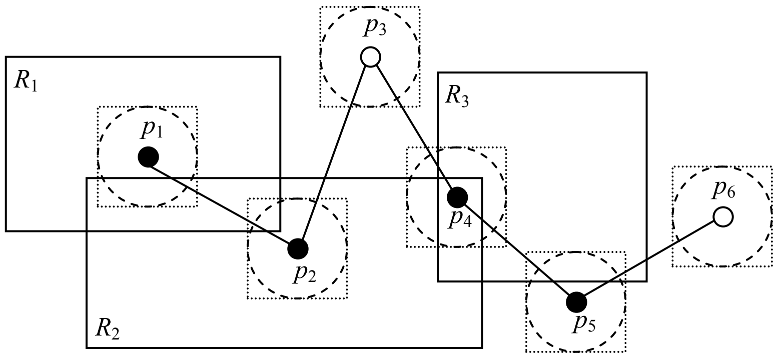


Figure 33. Transforming the PointOut rule into a rectangle intersection problem. Given two sets of orthogonal rectangles, find all overlapping that occurs between them

We present a simple yet elegant algorithm for processing MPRQ. As we need to single out all the black query points for each grey child node of the current MBR, one not so obvious technique is to transform all the query points into a set of rectangles. This is accomplished by extending length d in all four directions parallel to the axes from a query point, augmenting it to cover the circular radius of its search distance d. By doing so, we have approximated the circular query regions with rectangular query regions. This gives us an extra cover area of $d^2(4-\pi)$ for each query point for our coarse filter, and in some instances a white query point will be included in the pruned path for a rectangle because their corners overlap.

In the example of Figure 33, there are two distinct sets of rectangles. The first is a collection of child MBRs for a given node. We have O(m) rectangles in this set, m being the bucket size (degree fanout) of the R-tree. The second is a collection of n rectangles, each representing a query point in the route. Using our PointOut rule, n is the total number of black points with respect to the current node, therefore it varies (and become smaller) as we traverse down the R-tree.

Our approach using the rectangle intersection problem is derived from the general rectangle intersection problem. Instead of having one set of rectangles, we have two disjoint sets of rectangles and we want to report all the intersecting pairs between the two sets. The main objective is that we do not want to report the intersecting rectangles within the same set, but rather *across* the two sets. Simply put, we have: Given two sets of rectangles R_1 and R_2, find all pairs $r_1 \in R_1$ and $r_2 \in R_2$ such that $r_1 \cap r_2 \neq \emptyset$.

We implemented the algorithm of Figure 34 using the interval sets [Will85, MeNä95]. At each node, we first insert the interval of all the child rectangles, which is equal to the bucket size m. Inserting m intervals takes O($m \log^2 m$) time. Following that, we query the interval of all the points in the route P of n points, each query taking O($\log^2 m + k$) time, where k is the intersecting rectangles pairs found and to query a path of size n takes n ($\log^2 m + k$) for each node. Refer to Table 9 for the running time of MPRQ-RI.

```
MPRQ-RI-PointOut(R, P, d, P_new)
// Input:  MBR R, a sorted query path P, a search distance d
// Output: P_new() - array of sorted query subpath of P
begin
  forall r in R do
    insert interval [r.x_1,r.x_2] into rect
  endfor

  forall pt in P do
    result ← interval [pt.x - d, pt.x + d] ∩ rect
    forall i in result do
      if (i.y_1 - d ≤ pt.y ≤ i.y_2 + d) then
        j ← index of rectangle R at interval i
        P_new(j) ← P_new(j) ∪ {pt}
      endif
    endfor
  endfor
end; {procedure MPRQ-RI-PointOut}
```

Figure 34. Algorithm for the MPRQ-RI PointOut pruning rule

4.3.3 Running Time

Table 9 summarises the asymptotic running time of the four approaches that we had discussed. The amount of processing needed per node is also given. RRQ incurs a constant amount of processing as the query path is static. MPRQ has varied node processing time depending on the length of the query path at each level of the search tree. The approaches differ in the method used in path pruning and they all use O(*N*) space, where *N* is the size of the spatial database. We use *k* to denote the number of results returned.

Table 9. Various approaches to answering the multi-point range query, the amount of processing done per node and total running time. *N* is the size of the spatial database, *m* is the cardinality of node, *n* is the size of input query path, *k* is the size of the results, and *t* is the amount of processing per node

Approach	Amount of Processing per Node (*n* is length of path when entering node)	Running Time
RRQ	$m * n$	$n * N \log N + k \log k$
MPRQ MinMax	$\sum_{i=1}^{m}(n + k_i)$, $k_i \le n$	$t * N \log N$
MPRQ Sorted Path	$\sum_{i=1}^{m}(2 \log n + k_i)$, $k_i \le n$	$n + n \log n + t * N \log N$
MPRQ Rectangle Intersection	$m \log^2 m + n (\log^2 m + k)$	$t * N \log N$

4.4 Experimental Setup

4.4.1 Datasets

A very popular choice of GIS database for research is the Topologically Integrated Geographic Encoding and Referencing (TIGER) system first introduced in 1990 by the US Census Bureau [TIGER02, Doli01]. The TIGER/Line files comprise a digital database of geographic features, such as roads, railroads, rivers, lakes, even political boundaries and census statistical boundaries, covering the entire United States. The database contains information about these features such as their location in latitude and longitude, the name, the type of feature, address ranges for most streets, the geographical relationship to other features, and other related information.

Many research works [APRS98, AHVV99, LeEL97, RoKV95, PaMa96] use the TIGER/Line database for experiments because it serves the purpose of uniformity for benchmarking results, is comprised of real-life data and is readily available in plain text format. Many free tools are available [GSR01] for converting TIGER to a database format suitable for research purposes and also for gathering, analysing and plotting the TIGER data graphically such as ArcExplorer, Autodesk MapGuide Viewer and Geographic Explorer. Therefore, for the purpose of running experiments for the external memory, we used the TIGER/Line datasets as well.

There are a number of ways in which we can utilise the TIGER/Line dataset files. The TIGER/Line datasets organise different kinds of information into many logical layers of sets of maps. Each layer represents a thematic approach to a particular purpose. For urban planning, the layers that contain data for streets, utility lines, transportation features and related information are useful. The U.S. Census Bureau proclaimed that the buildings represented in its TIGER/Line data they provide each contains a centroid calculated to be within the building [TIGER02]. This fits the criteria of spatial data similar to RADS database, except that it has more objects. We could utilise the layer that represents the buildings within a city as spatial data, as is used by the [SoRo01] who addresses the problem of *k*-nearest neighbour for a moving query point. In their experimental data, they chose real-world datasets extracted from TIGER/Line representing 120 hospitals, 1982 churches and 1603 schools in Maryland, USA.

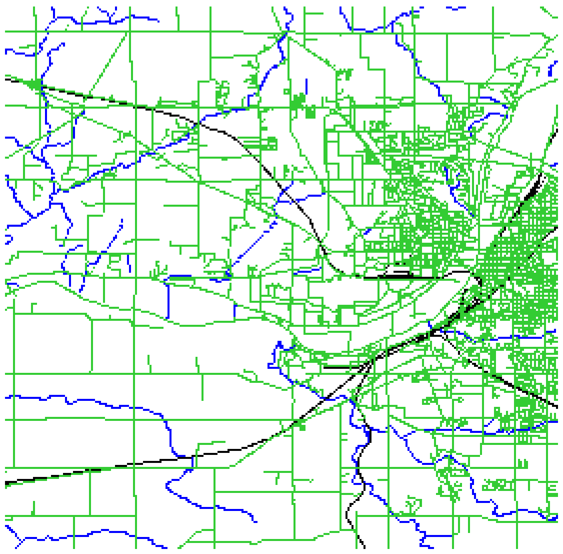


Figure 35. Real-life TIGER/Line datasets defining roads, rails and streams, among others, provided by the US Census Bureau using topology and graph theory design principles

4.4.2 Experiment Settings

We conducted extensive experimental study to evaluate the performance of MPRQ-Disk with large spatial databases that reside on disk. In this study, we used both synthetic datasets as well as real-life datasets. Synthetic datasets are generated from the outline of the Singapore map (using various broad parameters as described below). As they are more suited for internal memory experiments, we did not report the results of synthetic datasets here. We note that the external memory results for the Singapore datasets are comparable to their internal memory counterpart. Real-life datasets originate from the TIGER/Line datasets [TIGER02].

Implementations are done in C++ compiled with gcc version 3.4.4 on a Pentium IV 2.0 GHz Linux machine with 512MB RAM, with TPIE for disk implementations. The disk page size is 4096 bytes on our experiment machine. We consider the following factors when evaluating the MPRQ-Disk performance: the number of points in the spatial database *N*, the search distance *d*, different query routes *P*, different R-tree variants as spatial index and the effect of LRU buffering. For all of these experiments, we measure both the overall query time and the number of I/Os (disk accesses) to evaluate the performance of MPRQ-Disk. We ran each query 100 times and take the average of the running times, resulting in better accuracy.

Table 10. The number of spatial objects for various datasets from TIGER/Line. Road segments make up the bulk of the spatial objects. Our experiments only involve all the road objects

Datasets	New Jersey	Montgomery County, MD	Rhode Island
Short code	NJ	MD	RI
All spatial objects	369,814	30,997	58,804
Roads only	331,544	28,719	53,721
Percentage of roads	89.65%	92.65%	91.36%
File size	39.0MB	3.1MB	6.1MB

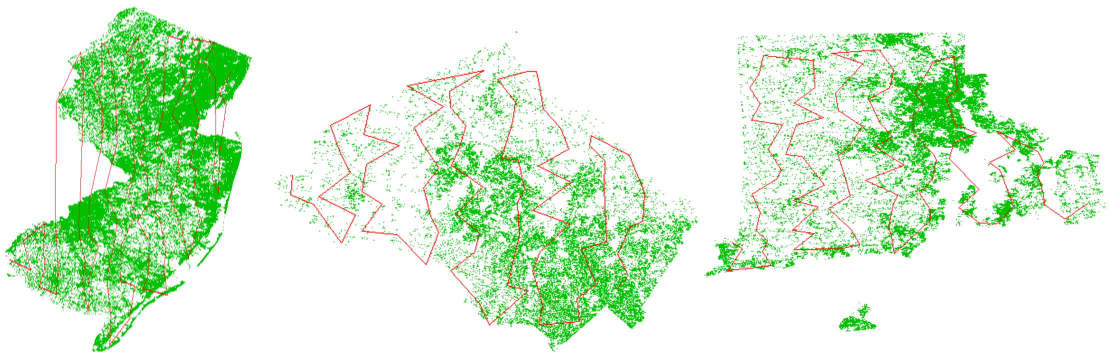


Figure 36. The (a) New Jersey, (b) Montgomery County, MD, and (c) Rhose Island datasets from TIGER/Line; the regionised query paths are shown; all figures not drawn to scale

Real-life dataset. Benchmark data from the TIGER/Line datasets [TIGER02] are used – the selected maps are New Jersey, Montgomery County, MD and Rhode Island. The size of spatial data is shown in Table 10. Note that New Jersey is about twice the size of the Singapore datasets (Figure 20), which is useful for comparison with the Singapore datasets used in internal memory experiments.

Regionised query paths. As real-life routes for maps in the chosen cities are not available, different kinds of synthetic routes are used instead. The maps are divided into rectangular cells of equal size and within each cell, a point is generated and appended into the query route set if it is contained within the polygon that defines the map boundary. We call such query route set *regionised* query path. The final regionised route sizes are 111, 80, 96 for NJ, MD and RI respectively. In addition, we also generated *H-path* and *V-path* for them.

Varying search distances (*r*). The search distances of (55, 60, 65, …, 90, 95) are used for the NJ case in real-life datasets. The number units here represent different real distances depending on maps. Most of the results reported for

MPRQ-Disk experiments use distance *d* = 75 for NJ. As other real-life maps exhibit similar trends to NJ, they are not included.

Table 11. The search distance *d* vs percentage of overlap for various datasets

	Singapore		New Jersey		Montgomery County, MD		Rhode Island	
Regionised path size	80		111		96		80	
Search distance vs Percentage of overlap	200	0.00	50	8.11	50	0.00	50	0.55
	500	0.31	60	12.72	100	1.54	100	0.90
	600	0.55	70	18.82	150	5.91	200	5.48
	700	1.21	75	22.52	175	10.27	250	10.99
	800	2.02	80	26.59	200	15.94	300	18.69
	900	3.03	90	36.48	250	30.31	350	29.19
	1000	4.48	100	48.51	300	47.96	400	42.13
	2000	42.15	150	130.22	400	96.94	500	74.68
	3000	114.09	250	372.38	500	158.69	750	186.17
	4000	211.61	500	1191.27	1000	594.01	1000	334.05
	5000	328.96						
	10000	1048.81						

Data structures. We implemented both algorithms for MPRQ-Disk and RRQ-Disk, as well as PR quadtree and several variants of the R-tree – the R*-tree, KDTopDownPack, HilbertPack and STRPack. Several other R-tree variants were also implemented but not reported here since their performance were worse than those from the representatives above. We also looked into the performance of the PR quadtree with buckets.

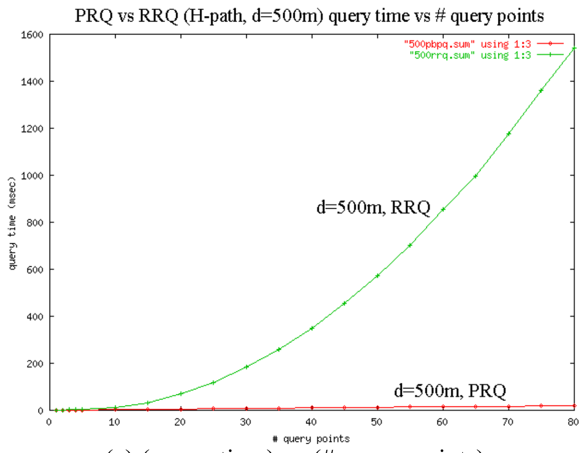
4.5 MPRQ-Disk Performance Evaluation

4.5.1 Baseline Comparison of MPRQ and MPRQ-Disk

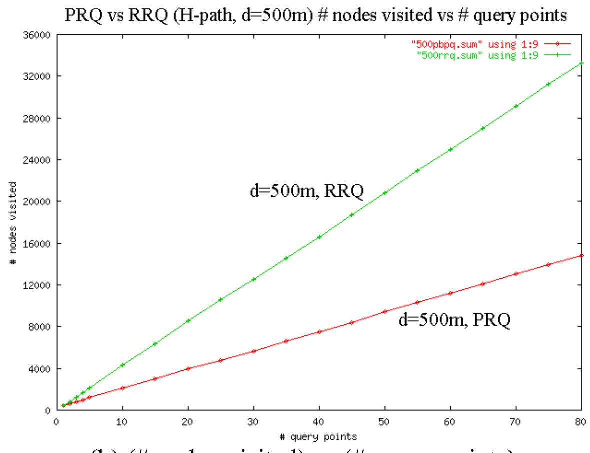
We begin by designing a series of experiments whose aim is to establish whether the results for MPRQ (internal memory) extend for MPRQ-Disk. In the previous chapter, we had established the fact that the MPRQ algorithm outperforms RRQ in many parameters, even in the case where the number of query points is small.

The results shown in Figure 37(a)-(b) were reproduced from the previous chapter for easy of reference. Using varying number of query points (between 1 to 80 in multiples of 5) in the *H-path*, we investigate MPRQ and RRQ as the input query set grows larger. The performance of MPRQ vs RRQ in internal memory indicate that the query time speed-up is 81 times for *m* = 80; and 6.5 times for *m* = 10. For the case on disk, Figure 37(c), MPRQ over RRQ speed-up is 7.93 times for *m* = 80; and 2.46 times for *m* = 10. As for the number of I/Os in Figure 37(d), RRQ incurs 2.5 times more I/Os for *m* = 80.

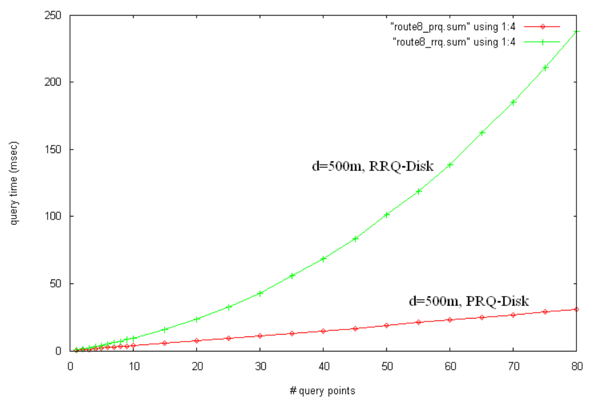
In main memory, the speed-up is significant as the MPRQ pruning rules cut down the query points to the necessary subset (black points) relevant to the MBR at any level. This significantly reduces the amount of expensive distance computations (at the very least, finding MinDist and MaxDist) needed as the spatial index is traversed. However, on disk, the savings in computation is negligible as the cost of an I/O (a few orders of magnitude larger) eclipses it. In spite of this, the MPRQ still performs well because it is able to minimise the I/Os by not visiting a node unnecessarily.



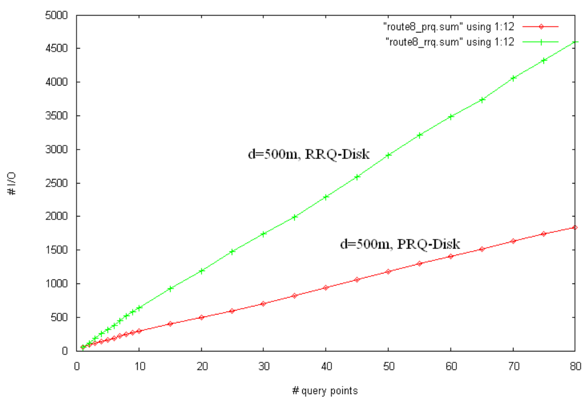
(a) (query-time) vs (# query-points) in internal memory



(b) (# nodes-visited) vs (# query-points) in internal memory

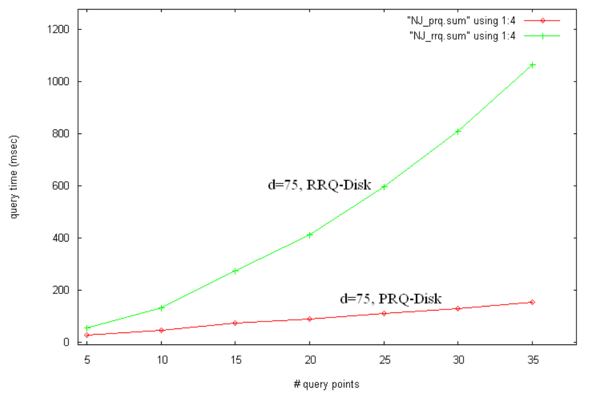


(c) (query-time) vs (# query-points) in external memory

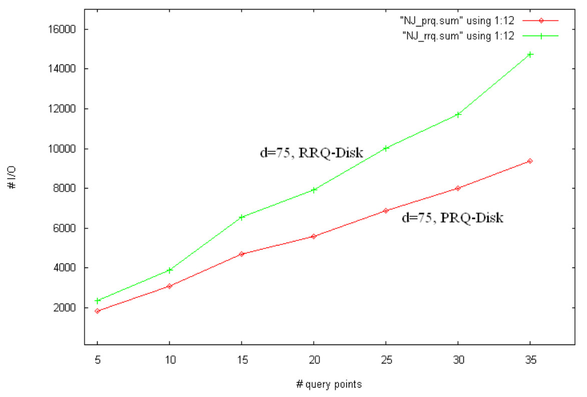


(d) (# I/Os) vs (# query-points) in external memory

Figure 37. Baseline comparison of MPRQ and RRQ in internal and external memory using query path *H-path* and *d*=500m



(a) (query-time) vs (# query-points)



(b) (# I/Os) vs (# query-points)

Figure 38. Comparison of MPRQ-Disk and RRQ-Disk for NJ dataset, query path *V-path* and *d*=75

Figure 38 shows the results comparing both the query times and the number of I/Os for MPRQ-Disk and RRQ-Disk in real-life New Jersey dataset, where the data is non-uniform. We chose *d* = 75 such that it returns about 20% of the total points when *m* = 35 using *V-path*. The query time speed-up is 7 times for *m* = 35. In general, we observed that the query time speed-up increases with the number of query points.

The reduction in the number of I/Os for MPRQ-Disk versus RRQ-Disk is also significant. For the case of query route *H-path*, the number of I/Os rises linearly with the number of query points for both MPRQ-Disk and RRQ-Disk. Figure 37(d) and Figure 38(b) show that, on average, the number of I/O requests by MPRQ-Disk is about 41.5% and 69.1% of that for RRQ-Disk for the Singapore dataset and the NJ dataset, respectively.

The results in this subsection established one fact – that the MPRQ-Disk algorithm performs correspondingly to MPRQ. In further sections, we just concentrate on MPRQ-Disk to find out how it fares with other parameters in further experiments.

4.5.2 Data Structures

PR Quadtree

For the PR quadtree, we observed that there are improvements in query time as the tree depth increases. As we vary the logical disk block size (LDBS), the average time does not observe any patterns of consistencies as the LDBS increases (bs = 1, 2, 4, 8, 16, 32). This is because the external data structures are mapped automatically by TPIE onto the physical location in the disk, and it is not possible that the whole data structure are in one consecutive region as our datasets include up to 160000 points. Disk fragmentation causes slight differences in the query time, due to the increase in latency time and seek time as shown in Figure 39.

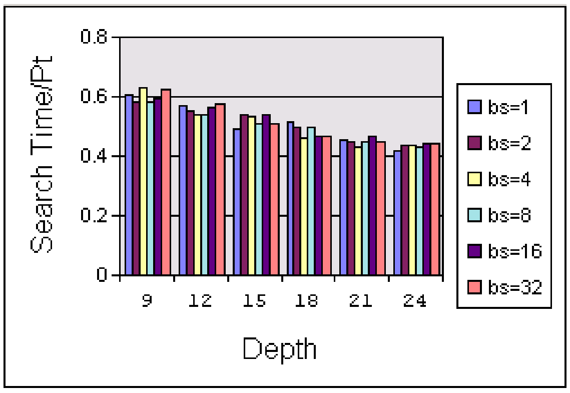


Figure 39. PR quadtree
(query-time/point) vs (tree depth)

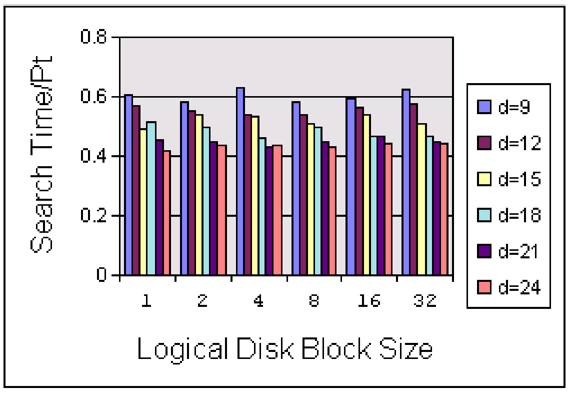


Figure 40. PR quadtree
(query time/point) vs (LDBS)

As we group by LDBS as in Figure 40, we observe that generally the query time decreases when the depth increases. At some point, the increase in depth (d = 9, 12, 15, 18, 21, 24) does not help to improve the query time when LDBS = 4 and LDBS = 16. This is due to the fact that reading a few logical blocks in advance may not help improve the query time because the event points are not necessarily near to each other in the PR quadtree. The ordering of the datasets plays a significant role when building the tree.

Bucket PR Quadtree

The bucket PR quadtree is an extension of the PR quadtree with bucket implementation at the leaves of the tree. In general, the query time improves when the depth of the tree is increased (Figure 41 and Figure 42). This is consistent with the results of their internal memory counterparts. When LDBS = 4, we see that the bucket implementation actually helps when the bucket size is above 8. When we double the LDBS to 8, we see that the query time is decreased when the bucket size is above 16 but increases when the bucket size and tree depth are increased. This is because the bucket implementation is a sequential list where the event points are stored when the tree depth is reached. When we perform a proximity query, the events that match is all in vicinity to each other and there is a high chance that they are stored in the same bucket. The larger the LDBS, the better the performance since we are going to search the whole bucket when we reach a leaf.

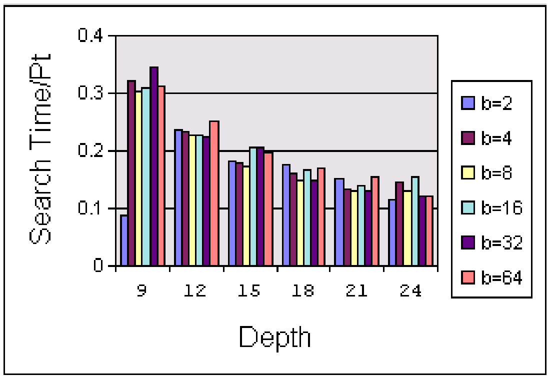


Figure 41. Bucket PR quadtree (query time/point) vs (tree depth) for logical disk block size of 4

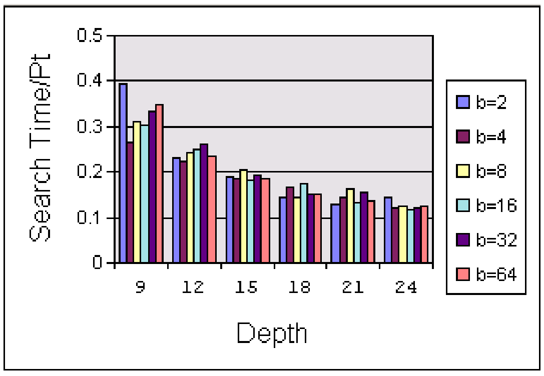


Figure 42. Bucket PR quadtree (query time/point) vs (tree depth) for logical disk block size of 8

When we group the results by tree depth, we see that the LDBS reduces the query time slightly when we double it. The only time when it helps is for the case where depth is 9. This happens because at a depth of 9, most event points get stored in the buckets. Pruning rules are less effective if a large search space is to be covered. Most of the time, we have to search through the buckets sequentially. Therefore a larger LDBS helps reduce query time by reading ahead. However, if the bucket is too large (e.g. bucket size of 64) we actually did not get any savings from increasing the LDBS especially if the LDBS is much smaller than the bucket size. We have to execute almost the same number of I/Os as in the case of the smaller LDBS.

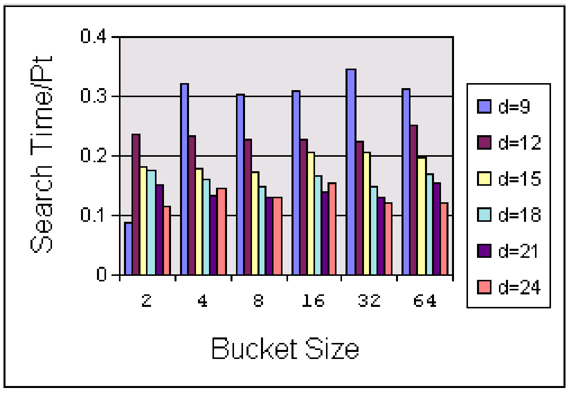


Figure 43. Bucket PR quadtree (query time/point) vs (bucket size) for logical disk block size of 4

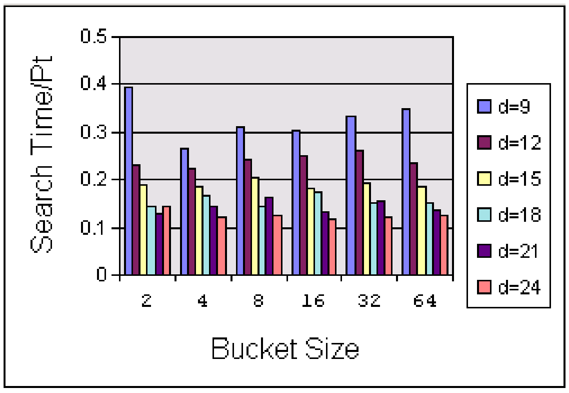


Figure 44. Bucket PR quadtree (query time/point) vs (bucket size) for logical disk block size of 8

R-trees and Variants

We ran experiments on the R-tree family based on different node splitting strategies (QuadSplit, LinearSplit, NewLinearSplit, R*-Split) and different R-tree bulk-loading strategies (HilbertPack, STRPack and KDTopDownPack). In Figure 45, the LinearSplit performance decreases with the increase in bucket size until the size reaches 32 where it remains stable. The performance of the R*-Split is more or less consistent regardless of the bucket size (Figure 46).

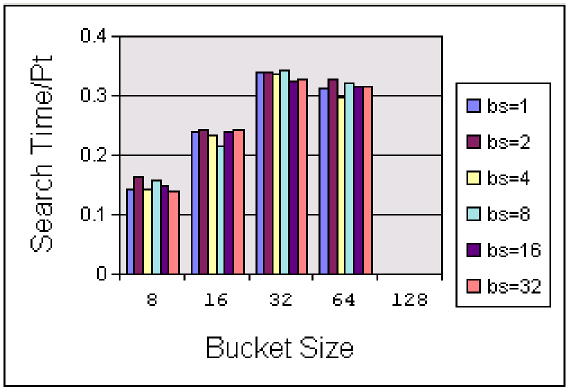


Figure 45. R-tree (Linear Split) of different logical disk block size

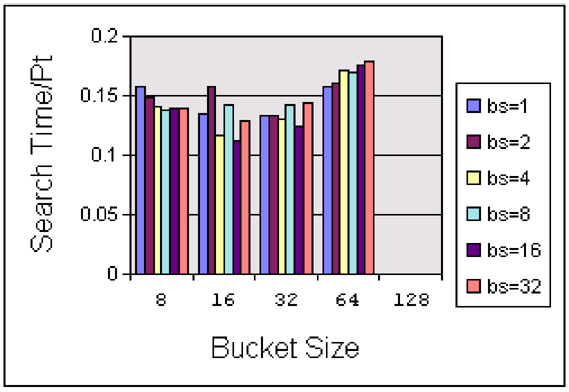


Figure 46. R-tree (R*-Split) of different logical disk block size

On the average, the R*-Split is two times faster than the LinearSplit and averages between 0.12 to 0.17 seconds.

The results of three different ways of bulk-loading an R-tree are in Figure 47, Figure 48 and Figure 49. The query time increases as bucket size increases because we segregated the event points first before building the R-tree bottom up so that there are no overlapping MBRs. This is due to the time taken to construct the R-tree is consistent but the larger buckets take longer to be searched because there will be less branch pruning before we hit a bucket on the leaf node level. This problem can be addressed with parallel algorithms when searching the R-tree. The KDTopDownPack R-tree in Figure 49 still retains the best performance like its internal memory counterpart.

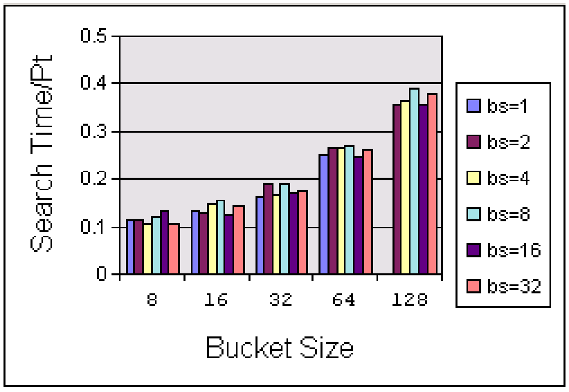


Figure 47. R-tree (HilbertPack)
of different logical disk block size

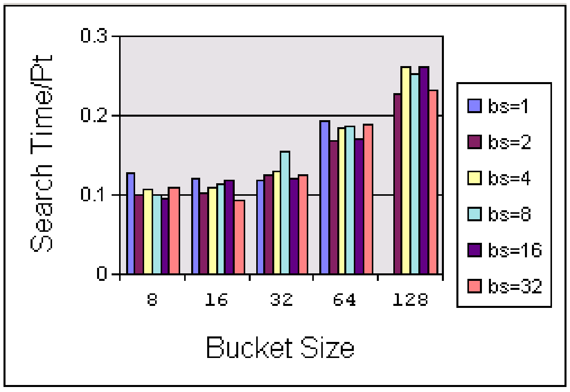


Figure 48. R-tree (STRPack)
of different logical disk block size

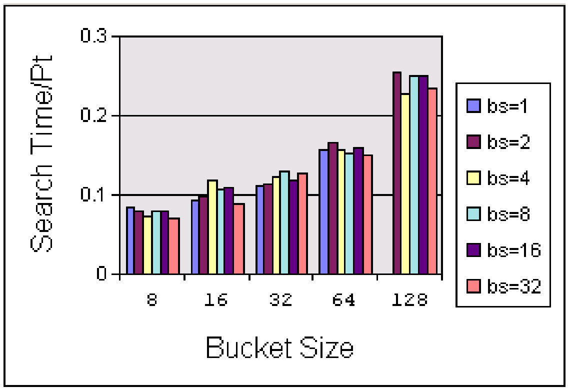


Figure 49. R-tree (KDTopDownPack)
of different logical disk block size

Now we group the bucket size together to see the effects of the LDBS. We observe that in Figure 50 the LDBS does not play a role in the performance, only the bucket size has effect on query time. In Figure 51, the R*-Split shows that a bucket size too large or too small has an adverse effect on the search time. This is true when we have a small LDBS (≤ 4), which does not help to cache the search space because the points are too far away (the tree is built in no particular order) to take advantage of the advance reading of the contiguous blocks. But when the LDBS becomes large (> 4) at the expense of more internal memory used, the larger buckets (for instance, 128) are "over read" by the advance cache because the LDBS exceeded the bucket size making some of the cache unnecessary.

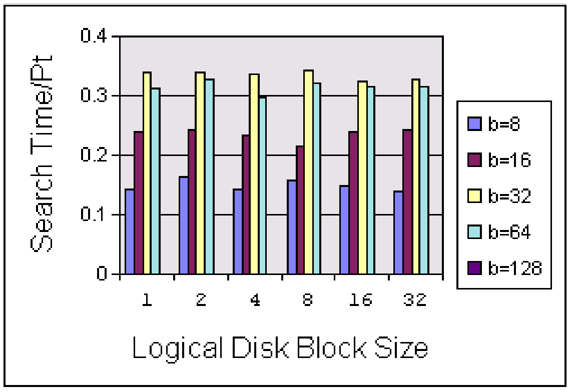


Figure 50. R-tree (Linear Split) of different bucket sizes

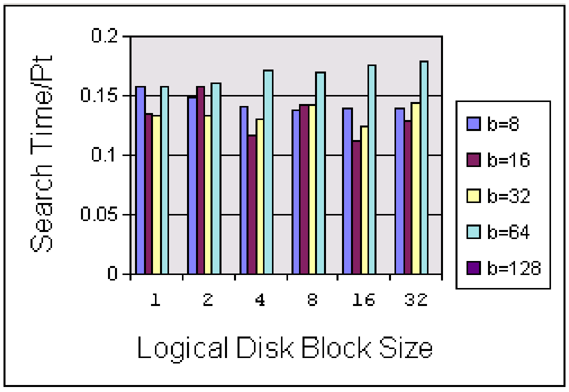


Figure 51. R-tree (R*-Split) of different bucket sizes

For the three different bulk-loaded R-tree structures, their results are presented in Figure 52, Figure 53 and Figure 54. Different LDBS does not have any effect on the trees because of the time taken to build the tree is fairly consistent for each tree. The reason is because we are exploring with just one disk, rather than multiple disks which will definitely influence the time. Much of the work is due to computation for separating the event points into their spatial region before we actually index those points. Building the tree does not take a lot of disk reads all across the index, only disk writes onto the single disk index structure. This difference is only evident in the amount of time taken to build the tree plus searching the tree, while the query time is of course influenced by the bucket size instead of the LDBS.

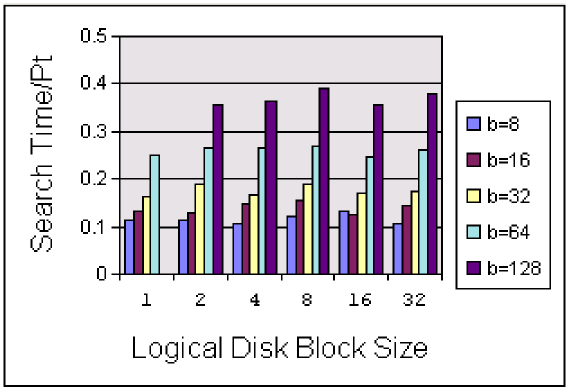


Figure 52. R-tree (HilbertPack)
of different bucket sizes

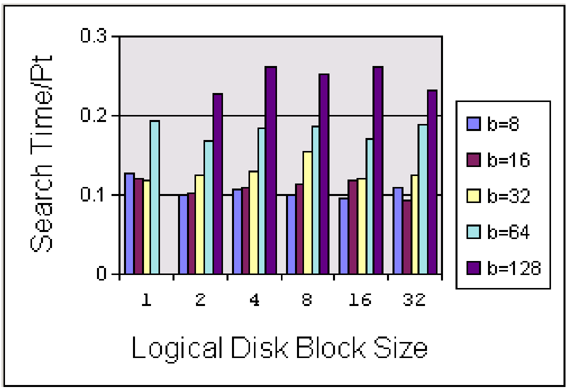


Figure 53. R-tree (STRPack)
of different bucket sizes

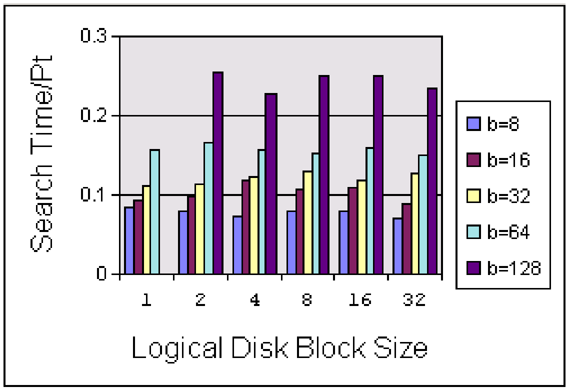
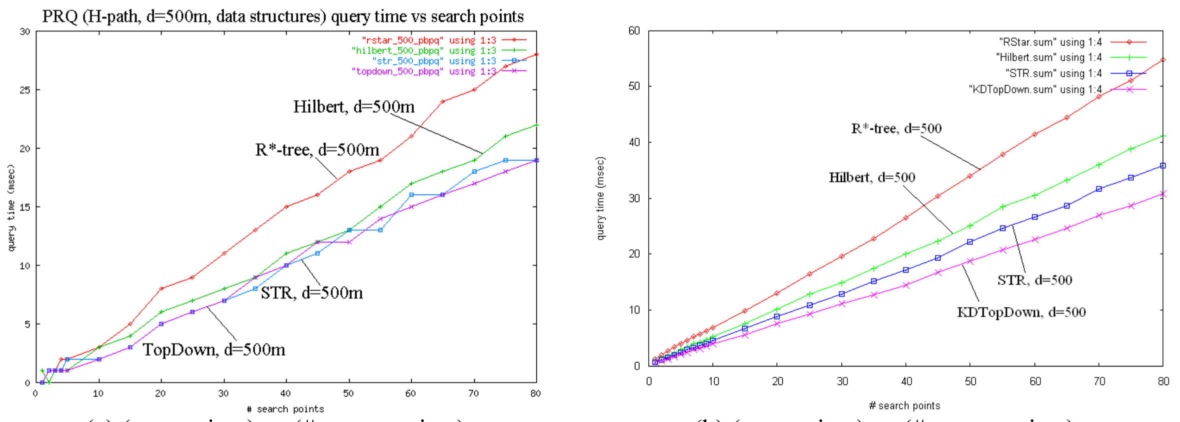


Figure 54. R-tree (KDTopDownPack)
of different bucket sizes

Underlying Data Structures

The underlying spatial index will have effect on the performance of MPRQ-Disk because objects that are spatially close and indexed as such will result in lower I/Os and improved query time, due to locality of reference. We ran similar sets of experiment on different variants of the chosen R-tree data structures, namely, R*-tree, HilbertPack R-tree, STRPack R-tree and our own KDTopDownPack R-tree. The results tend to be similar to previous results for both MPRQ-Disk and RRQ-Disk. To obtain a more detailed comparison of the different R-tree variants, Figure 55 shows the performance of only MPRQ-Disk on the different R-tree variants.



(a) (query-time) vs (# query-points) in internal memory

(b) (query-time) vs (# query-points) in external memory

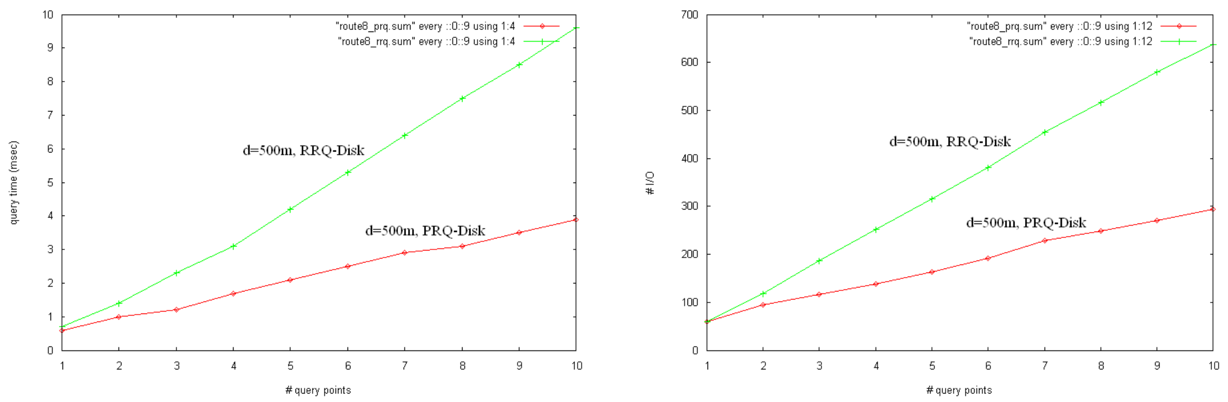
Figure 55. MPRQ-Disk performance on different R-tree data structures: HilbertPack, R*-tree, STRPack and KDTopDownPack for query distance *d*=500m

For the case where database resides in internal memory, the performance (from best to worst) is as follows: KDTopDownPack, STRPack, HilbertPack, R*-tree. In particular, KDTopDownPack and STRPack are very close in terms of performance. However, we can clearly see that R*-tree is outperformed by the others which are bulk-loading algorithms that results in better indexing of spatial points with minimal area of MBRs overlapping.

On the other hand, for the case where the spatial database resides on disk, we can arrange their performance again as in the internal memory case, with clear distinction. The dominance of I/O costs in the overall query time for different data structures clearly shows. KDTopDownPack has a better packing algorithm for objects as compared to the rest.

4.5.3 Small Set of Query Points

In general, we expect MPRQ-Disk to perform better as the number of points in the query route *P* increases. As a stringent test we have also zoomed into the cases where $1 \leq m \leq 10$. Figure 56(a) shows that MPRQ-Disk runs slightly faster for the special case of just one query point when m = 1 (normal single point range query) because the PointOut pruning rule that generally exert more computations for MPRQ-Disk (as opposed to RRQ-Disk) did not fire. This is by design. The rule only fires when $m \geq 2$. Meanwhile, the NodeIn rule is fired when the index traversal reaches a point where the query distance covers an entire MBR which triggers all of its children to be reported without further computations. This makes MPRQ-Disk faster than RRQ-Disk even when there is just one query point.



(a) (query-time) vs (# query-points) (b) (# I/Os) vs (# query-points)

Figure 56. MPRQ-Disk performance with small number of query points ($m \leq 10$) and d=500m

As for the number of I/Os, Figure 56(b) reveals that at m = 1, both MPRQ-Disk and RRQ-Disk incurs the *exact* same amount of I/Os. This is true because even if NodeIn rule fired, it still has to traverse until the leaf level to report all results although it does not need any further calculations.

Additional results for RI and MD datasets, *V-path* and *D-path* also show identical trends with respect to performance comparison between MPRQ-Disk and RRQ-Disk. Therefore, for the remainder of this study, it suffices to report on results for regionised routes.

4.5.4 Effectiveness of Pruning Rules

Table 12 shows the comparison of the effectiveness of different combinations of pruning rules between internal and external memory. We also selected the RI dataset, the largest that can fit entirely into internal memory, for this set of experiments. Finally, the query time for RRQ is also included for comparison.

Table 12. The effectiveness of applying different pruning rule combinations, comparing internal and external memory. For this comparison, only one real-life dataset is shown

Query time (sec)	SG (internal)	SG (external)	RI (internal)	RI (external)
NodeOut	0.1310 (100%)	1.5470 (100%)	0.0181 (100%)	0.5870 (100%)
NodeOut+PointOut	0.0220 (17%)	0.4319 (28%)	0.0104 (57%)	0.1160 (21%)
NodeOut+NodeIn+PointOut	0.0200 (15%)	0.4228 (27%)	0.0094 (52%)	0.1046 (18%)
RRQ	1.6590	8.4027	0.1629	2.1622

Since we established that PointOut is much more effective than NodeOut in the internal memory case, in this experiment we focus on PointOut. We observed that by adding pruning rule PointOut on top of NodeOut in external memory, we obtain a 72% decrease in query processing time. This is not as good as the internal memory case of 83% as the number of I/Os has taken a toll on query time. PointOut computation is a memory intensive computation, but the bulk of query time is still tied to disk accesses no matter how much we prune the path with PointOut. Adding NodeIn will gain us an extra 1%-3% of query time in the external memory case, as we only save some computation time but still need to access the necessary disk nodes to obtain the results. This trend is similar to the internal memory case. The RI dataset also show the same trend in query time reduction, but at a slightly different quantum.

We observe that for internal memory, the RRQ is almost 3 orders of magnitude slower than MPRQ. For external memory, the gap closes as the number of I/Os is the dominant factor, not internal computations. Yet, RRQ is still 20.67 times slower.

Table 13. The effectiveness of applying different pruning rule combinations, comparing different datasets

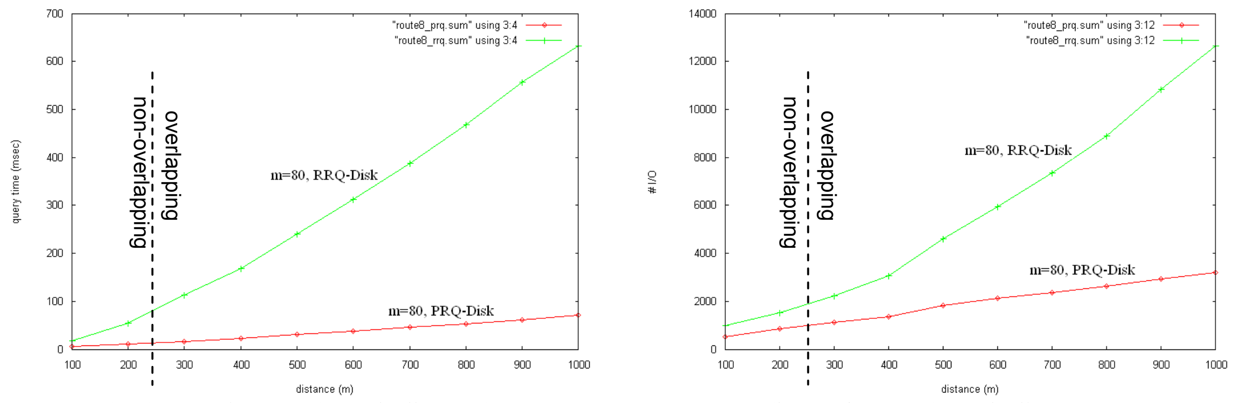
Query time (sec)	SG	NJ	MD	RI
NodeOut	0.1310 (100%)	0.2215 (100%)	0.0104 (100%)	0.0181 (100%)
NodeOut+PointOut	0.0220 (17%)	0.1085 (49%)	0.0049 (47%)	0.0104 (57%)
NodeOut+NodeIn+PointOut	0.0200 (15%)	0.0741 (33%)	0.0045 (43%)	0.0094 (52%)
RRQ	1.6590	2.9501	0.0493	0.1629

Table 13 compares the effectiveness of pruning rules across external memory datasets. The search distance is tweaked for each dataset such that 20% of the database is returned, and regionised routes are used for all datasets including SG. There is a difference in query time reduction between the SG and the rest. This is because the SG dataset is so small it can totally fit into internal memory. This causes less paging operations (loading disk nodes into internal memory) than other datasets which results in better query time.

The trend for NJ, MD and RI is about the same; PointOut results in about half the query time reduction (47%-57%) and applying NodeIn will result in 4%-16% further reduction in query time. The NJ dataset exhibit better reduction for NodeIn (16%) because its map is much denser than that of MD and RI. Thus, once NodeIn fires it is able to return more results for the same number of I/Os for the same MBR area. Similar to past trends, on average RRQ is 2-3 orders of magnitude slower.

4.5.5 Size of the Search Distance

We now investigate the performance of MPRQ-Disk across different query distances *d*. Given any set of query points, when *d* is large, overlapping of query regions will result in many duplicate results obtained by RRQ-Disk (since each query point is a standard range query, independent of the rest of the points in the same query points set, no matter how close they are to the current point in query) which in turn results in a longer post-processing time to remove duplicates.



(a) (query-time) vs (search-distance) (b) (# I/Os) vs (search-distance)

Figure 57. MPRQ-Disk performance for varying distances *d* with *H-path* 80 query points

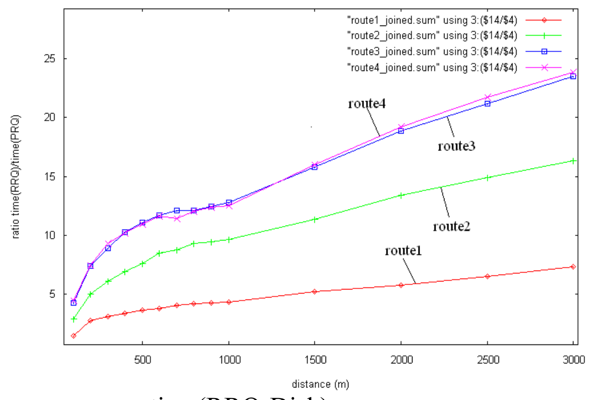
Recall that for $d < 250$m, there is no overlapping of search area because the *H-path* is made up of query points with regular interval of 500m along the x-axis. Figure 57 shows that for non-overlapping areas, where no redundant results are present, RRQ-Disk grows similarly to MPRQ-Disk (in terms of I/Os). However, when overlapping occurs, RRQ-Disk uses more I/O requests (for duplicates actually) which is totally redundant and this contributes to its long query time.

In fact, MPRQ-Disk growth is linear because excessive overlapping in query regions does not add to the algorithm's running time. The larger the query distance, the longer it takes to complete the query.

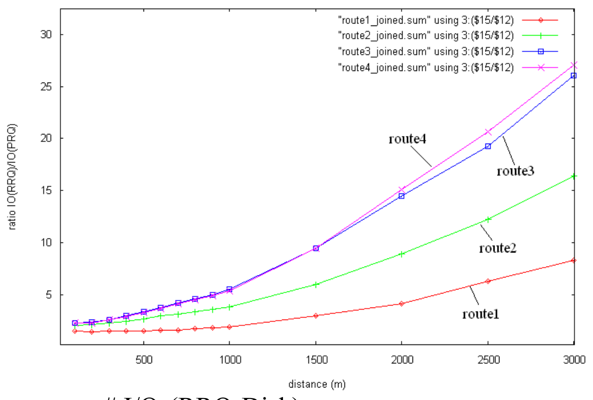
4.5.6 Performance of Real-life Routes

Real-life routes provide an insight into how the MPRQ-Disk algorithm fares when deployed for use. Our target application is RADS which helps a user plan a route and subsequently discovers POIs along the planned route [NgLH04]. The performances of the four real-life routes (route1-4) are shown in Figure 58 showing clear advantages of MPRQ-Disk over RRQ-Disk.

In Figure 58(a), the query time speed-up for real paths are generally similar to those for the synthetic *H-path* (shown in Figure 57). The reduction in the number of I/Os for MPRQ-Disk also widens with the query distance.



(a) $\frac{\text{query-time(RRQ-Disk)}}{\text{query-time(MPRQ-Disk)}}$ vs (search-distance)



(b) $\frac{\text{\# I/Os(RRQ-Disk)}}{\text{\# I/Os(MPRQ-Disk)}}$ vs (search-distance)

Figure 58. MPRQ-Disk performance for real-life paths (route1-4)

4.5.7 Comparison of MPRQ Algorithms

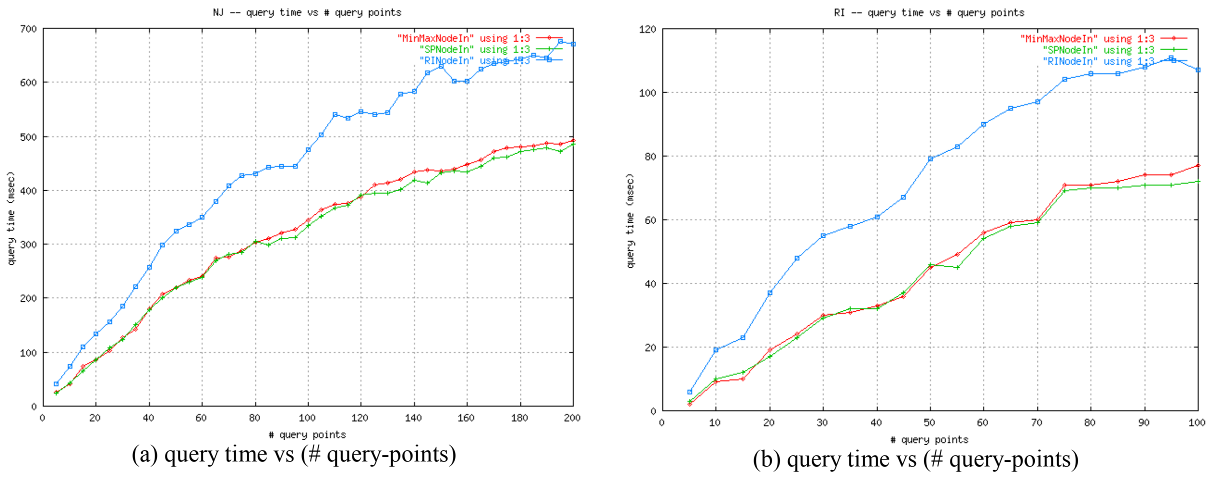
Earlier experiments established the fact that the speed-up of MPRQ-MinMax against RRQ increases with the number of query points, the search distance, the presence of clustered data, different planned routes, different spatial representations of the spatial database, as well as the bucket sizes. Hence, we will just focus on the comparison among MPRQ-SP, MPRQ-RI and MPRQ-MinMax.

To compare the performance of the various approaches for implementing the PointOut and NodeIn, we implemented all the three pruning rules using the algorithm described in Figure 27 and those from Section 4.3.1 and Section 4.3.2. For the NJ dataset, we chose a random path that has 200 query points (i.e. n = 200). For the relatively smaller RI dataset, we chose a random path of 100 query points.

We did not show the total number of I/Os for MPRQ-MinMax, MPRQ-SP and MPRQ-RI because all three approaches does the same pruning (PointOut and NodeIn) under the same circumstances, i.e. the input query path is the same. Therefore, all three traverse the tree in the same manner. The only difference is in the speed of traversal attributed to the different pruning strategies.

In Figure 59, the performance of MPRQ-MinMax and MPRQ-SP are almost similar, with the latter doing slightly better when $n \geq 80$. We attribute this to the initially high startup cost of MPRQ-SP (sorting along axis major) gradually being recouped after which the performance is better for MPRQ-SP. In comparison, MPRQ-RI pruning performs worse than the other two even from the beginning. At n = 200, MPRQ-RI is 1.38 times slower than MPRQ-SP. Our theoretical results (Table 9) already show that while the approach of MPRQ-RI is more elegant, the associated costs are expensive because it involves multiple insertion and query of the interval set data structure with varied path length.

On the other hand, MPRQ-SP can still be improved further because binary search on the sorted path is not quite optimal when the path is exceptionally short (e.g. $n \leq 5$). This is because for short paths, which occur frequently at the bottom of the R-tree during traversal, binary search does more comparison than plain sequential search. Since MPRQ-SP uses a sorted path that relies on biased binary search routines for the PointOut pruning, we believe that the running time can be further improved by employing hybrid sequential search and biased binary search for PointOut pruning. MPRQ-SP requires some pre-processing time, but we shown that it generates a lot of savings in the later search stage. An added advantage is that it is very easy to implement, with relatively good results when compared to RRQ and MPRQ-MinMax.



(a) query time vs (# query-points)

(b) query time vs (# query-points)

Figure 59. Performance of the MPRQ-MinMax (red), MPRQ-SP (green) and MPRQ-RI (blue) for (a) NJ dataset and (b) RI dataset

4.5.8 Effect of LRU Buffering

Using KDTopDownPack to construct the 40000 dataset, we designed a query consisting of a real-life path of 34 points and d = 500m. We vary the least recently used (LRU) buffer size from 10%, 11%, …, 19%, 20%, 30%, …, 90% of the total internal nodes. Previous studies of LRU buffering [ThSe96, LeLo00] suggest that as little as 10% buffer size (i.e. buffer size equals $n*p/100$ of the total number of n nodes given p percent) could halve the number of I/Os required. Our aim is to prove a fair case for RRQ-Disk (as even a straightforward implementation benefits from a LRU of some sort in modern databases) against MPRQ-Disk. We aim to check the hypothesis that RRQ-Disk performs better with the help of LRU found in the O/S.

We observe that an LRU buffer as little as 10% cuts down I/Os by approximately 68.9% for RRQ-Disk (with 91.63% buffer utilization), mostly because the spatial index is traversed repeatedly for each query point p_i and down a slightly different path the next time if p_{i+1} is near. RRQ-Disk benefited if nodes from the previous search is retained in the LRU buffer. MPRQ-Disk does not show any effect as it optimally accesses only the nodes that are relevant, and only once in the spatial index, for all query points in P.

In Figure 60, LRU buffer ≥ 17% for RRQ-Disk improves its performance only marginally. Our experiments run all the way to 90% (although in practice, this is not feasible unless the spatial database is small) which shows that MPRQ-Disk still requests 12.96% fewer I/Os than RRQ-Disk in spite of the presence of LRU that should benefit the latter.

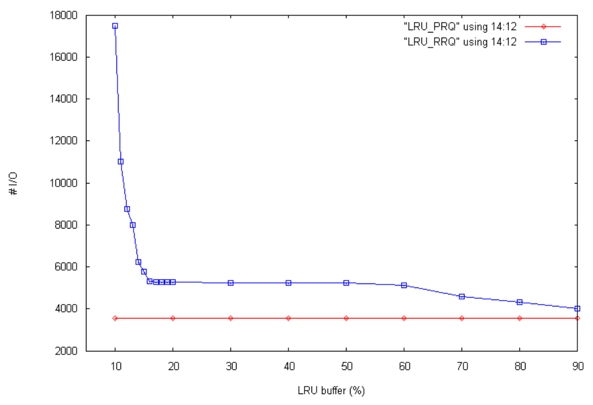


Figure 60. MPRQ-Disk and RRQ-Disk under different buffer sizes

4.6 MPRQ-Disk vs Spatial Join Algorithms

In this section, MPRQ-Disk is evaluated against spatial join approaches that can also be used to solve MPRQ. We have carefully chosen the high-performance spatial join techniques of [BrKS93] which aims to join two datasets indexed by two R-trees, and the slot index spatial join [MaPa03] which aims to join a non-indexed dataset with one indexed by an R-tree. Due to the similarity of spatial joins to MPRQ, performance evaluation is imperative.

The MPRQ-Disk algorithm used to compare with other spatial join algorithms defaults to MPRQ-MinMax with KDTopDownPack, using LDBS = 4, bucket size of 8, and with all three pruning rules (NodeOut, NodeIn, PointOut) in effect. All experiments for SJ4 and SISJ are performed 100 times and the average query time is taken.

4.6.1 High-Performance Spatial Join

An efficient full distance spatial join algorithm, SJ4, was introduced in [BrKS93]. SJ4 is already proven to outperform another class of spatial join algorithm, the distance semi-join [ShML02]. The result is reproduced in Figure 61(a), with SJ4 labelled SJ-SORT. For join result size greater than 10K (one join pair is one result), which MPRQ is designed for, SJ4 is clearly faster than distance semi-join algorithms HS-KDJ, B-KDJ and AM-KDJ. The reported response time for SJ4 is ≈37.5 seconds (10,000 result pairs) and ≈75 seconds (100,000 result pairs). Hence, we are motivated to compare MPRQ-Disk to SJ4.

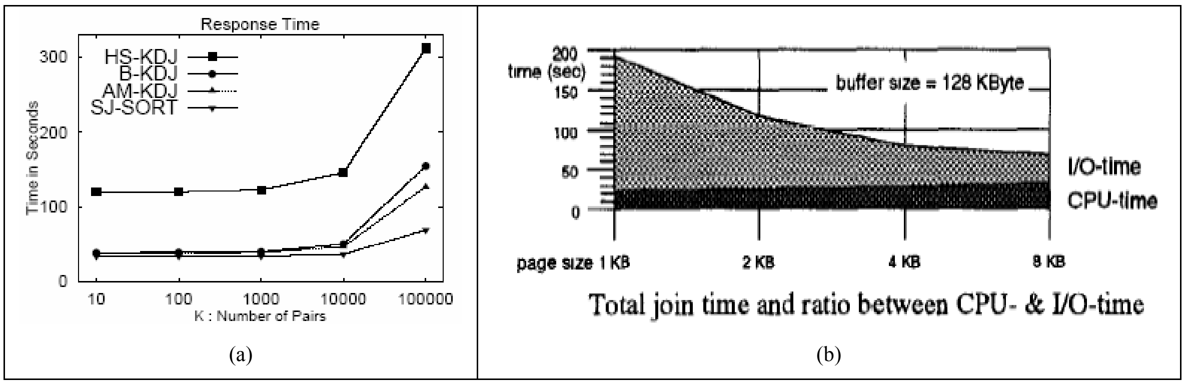


Figure 61. (a) The performance of distance semi-join algorithms (B-KDJ and AM-KDJ from [ShML02]; HS-KDJ from [HjSa98]) compared to SJ4 (SJ-SORT), (b) the performance of SJ4 full spatial join algorithm reproduced from [HjSa98]

Taking into account the difference in hardware speed and the amount of RAM between [BrKS93] and our work (Moore's law), we benchmark SJ4 and MPRQ-Disk with our 2.4 GHz CPU, 4 GB RAM Linux machine. As the dataset used in experiments for SJ4 is no longer available (TIGER/Line 1990), we used our NJ dataset [TIGER02] to benchmark. A total of 331544 roads and 9759 railways were selected for the benchmarking. Spatial indexes for SJ4 (which requires two independent R-tree indexes) and MPRQ-Disk were constructed on disk using TPIE [Veng94] with our bulk-loading KDTopDownPack algorithm. Bulk-loading the data points significantly reduces the amount of overlapping rectangles, thus reducing the number of generated candidate pairs in the SJ4 algorithm, benefiting it directly as SJ4 uses intersection tests and plane sweeping at every level of the index.

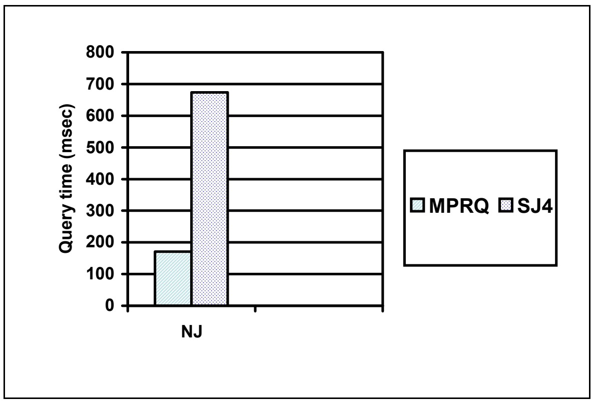


Figure 62. Benchmarking SJ4 to MPRQ-Disk using the NJ dataset of 331,544 (roads) × 9,759 (railways)

Figure 62 shows that MPRQ-Disk outperforms SJ4 in query time. Both MPRQ-Disk and SJ4 assume the spatial dataset to be indexed with the R-tree. We measured only the query time instead of the total time, which includes time to construct the spatial index. We are sure that SJ4 will cost even longer as it requires both the data points and the query points to be constructed as two independent R-tree, whereas MPRQ-Disk only constructs one R-tree for the data points.

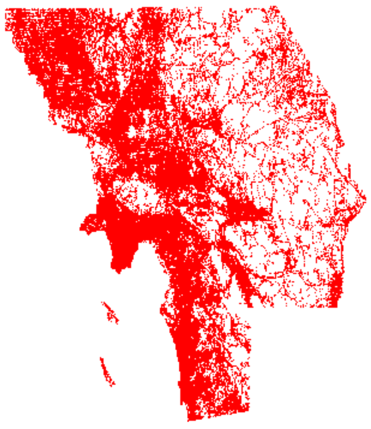


Figure 63. Roads from all the 5 counties of the California dataset, obtained from TIGER/Line 2006

For a larger dataset, we had chosen all the roads from 5 counties (Kern, Los Angeles, Riverside, San Bernardino and San Diego) within California, USA from TIGER/Line 2006 (2nd ed). There are a total of 643776 roads, as illustrated in Figure 63. Some features were selected to be query points (routes) and they are selected in such a way that we get a small, medium and large ratio between the data points and routes, for scalability concerns.

Table 14 depicts the query time of MPRQ-Disk vs SJ4. For any ratio of database to query size, MPRQ-Disk outperforms SJ4. In the small dataset, MPRQ-Disk and SJ4 are almost identical. For medium and large datasets, the sorted intersection tests performed by SJ4 in each iteration have increased its response time significantly. As for MPRQ-Disk, when the R-tree is traversed, the set of query points are quickly reduced and vary slightly for each rectangle during query. Since each set of candidate points for a rectangle is a subset of the set at the upper level, no additional disk accesses are needed.

Table 14. Performance of MPRQ-Disk vs SJ4 in large dataset with small, medium and large routes

Dataset	Roads	Route		Ratio	MPRQ (ms)	SJ4 (ms)
Small	643,776	Physical features	763	1:844	250	262
Medium		Railroads	9,641	1:105	266	459
Large		Hydrography + Non-visible features	247,890	1:2.6	1090	2316

Next, we rerun some of the datasets from Section 4.4 to compare MPRQ-Disk and SJ4 using very small routes, in which MPRQ-Disk was originally designed for. The results are presented in Table 15. In the NJ dataset, SJ4 takes much longer to run, compared to a larger dataset in Table 14 (the small dataset). The routes used in NJ are regionised routes designed to be spatially far. Due to this, SJ4 cannot take advantage of locality of reference as in the previous experiment where a bunch of spatially close route points would likely be read together into main memory.

Table 15. Performance of MPRQ-Disk vs SJ4 in very small routes

Dataset	Roads × Route	MPRQ (ms)	SJ4 (ms)	Improvement
NJ	331,544 × 111	190.9	447.5	134.4%
MD	28,718 × 80	13.3	19.5	46.6%
RI	53,721 × 96	25.0	50.1	100.4%

4.6.2 Slot Index Spatial Join (SISJ)

SISJ is an algorithm that joins a non-indexed dataset with one indexed by an R-tree [MaPa03]. In certain spatial queries, the non-indexed dataset could be the intermediate result of another database operator. For instance, in a multi-way spatial join operation involving three datasets $A \bowtie B \bowtie C$, the spatial join algorithm could perform $(A \bowtie B) \bowtie C$ or $A \bowtie (B \bowtie C)$ with the intermediate result R joined to the remaining dataset. SISJ has the advantage of being useful when R is fairly large and it is costly to materialise R before processing it.

SISJ distributes the R-tree entries at a specific level into S partitions, called slots, and builds an in-memory index from them. In each slot, a slot index keeps track of a list of pointers to all corresponding entries in the slot, along with a MBR of all the entries. Slots are basically a kind of hash table which is small enough to fit in main memory. The non-indexed dataset is also partitioned into the S buckets with the same spatial extents as the MBR of the slots. The algorithm finally joins each bucket with the R-tree data under the nodes pointed to by the corresponding slot. Figure 64 shows an example of SISJ for an R-tree at level 1 (second level from the root) constructed from all its MBRs at that level. Note that SISJ is applied only on one specific chosen level of the R-tree and the slots' MBR can also be overlapping.

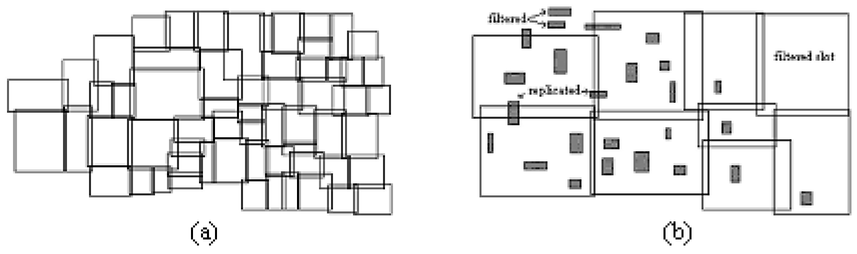


Figure 64. An R-tree and a slot index built over it. (a) the entries for an R-tree at level 1, (b) a slot index built from the R-tree entries and hashed data from the non-indexed dataset. Data that spread across two or more slots are replicated for queries. Data that are outside all slots are filtered. SISJ is performed between a slot and its corresponding hashed data only

There are four slot index construction heuristics that determine the extents for space partitions used for hashing the non-indexed dataset, namely SplitXL, SplitHC, SplitSTR and IRS. SplitXL sorts MBRs w.r.t. their lower x-bound and divide them into *S* equal-sized groups. SplitHC sorts MBRs w.r.t. the Hilbert value of their centre and divide them into *S* equal-sized groups. SplitSTR sorts MBRs using STRPack algorithm and divide them into *S* equal-sized groups. Finally, IRS inserts the entries into *S* slots using the R*-tree insertion algorithm. Among all four, IRS consistently gives the best query cost savings in the original paper.

For comparison, we used the same dataset listed in Table 14. The overall query cost (in seconds) are measured and presented. Our chosen dataset represents very well the different scenarios that a spatial join result set would be. Typically, the ratio for small dataset is similar to a distance semi-join query where a small distance limits the result to the top few results from an input query. The ratio for large dataset represents a spatial query that touches the whole map, returning many results.

For SISJ, the buffer size allocated was 512K with 4K disk page size. [MaPa03] has empirically shown that buffer size of one order of magnitude smaller than the dataset size is realistic, for datasets on any scale. The dataset size is ~5.15MB, so 512K is about one order of magnitude smaller. The larger the buffer is, the more hashed data from the non-indexed dataset can be stored in main memory for spatial join processing with the slot indices, which in turn helps to cut the number of disk access needed to process the hashed dataset buckets. For MPRQ-Disk, its parameters were carefully chosen so that the full route cannot fit in main memory during its execution. Note that MPRQ-Disk does not maintain a buffer in main memory to store MBRs.

Table 16. Performance of MPRQ-Disk vs SISJ in large dataset with small, medium and large routes. All four slot index construction policies are compared

Dataset	Roads	Route	Ratio	MPRQ (s)	SISJ IRS (s)	SISJ SplitSTR (s)	SISJ SplitHC (s)	SISJ SplitXL (s)	Speed-up MPRQ vs SISJ IRS
Small	643,776	763	1:844	0.250	1.01	1.06	1.18	1.23	4.04
Medium		9,641	1:105	0.266	9.09	9.89	11.71	12.30	34.17
Large		247,890	1:2.6	1.090	176.06	194.34	232.62	256.43	161.52

The SISJ slot indices performance is consistent with the results in [MaPa03], which shows that IRS is the fastest, followed by SplitSTR, SplitHC and SplitXL. MPRQ-Disk fares better compared to IRS. Table 16 shows that the speed-up for small dataset is 4.04 times and for large dataset is 161.52.

For small datasets, in SISJ both the slot indices data (R-tree of indexed roads) and all buckets of hashed data (non-indexed route) could fit in main memory. So, a plane sweep algorithm is performed in main memory across all indices to find the spatial join result pairs. For medium datasets, only the data under a slot index fit in memory. In this case, SISJ uses the indexed nested loop join, considering the slot as the root of the R-tree; for each rectangle in the hash bucket, a window query is applied. To process the hash bucket fully, since there is not enough space in the buffer, a number of blocks have to be loaded from disk and an equal number of blocks have to be written from some other hash buckets. For large datasets, neither the data under a slot nor the bucket fit in memory. SISJ will perform joins similar to a recursive hash-join algorithm. The slot acts as the virtual root of an R-tree and a hash bucket as the non-indexed dataset. I/O cost is incurred for each and every slot as slot data are read into the buffer.

The similarity of SISJ and MPRQ-Disk lies in an indexed R-tree, and that is all there is to it. SISJ needs to build an extra slot index on an R-tree as a pre-processing step. The performance of SISJ in reality depends on hashing the non-indexed input and the resulting algorithm used (different algorithm depending on whether the data in a slot and hash bucket could fit in main memory), with plane sweep being the most common. Compared to SISJ, MPRQ-Disk is an easier to implement method. We had looked into the plane sweep algorithm for MPRQ (as a rectangle intersection problem), but our research shows that it is slower than the simple MinMax method.

4.7 Summary

In this chapter, we revisited the MPRQ problem and the efficient MPRQ algorithm which we proposed for solving MPRQ. MPRQ and its performance in internal memory were studied to depth in the previous chapter, with comparisons to the RRQ. More often than not, spatial databases contain more data than can fit into the internal memory. Hence we address the case where the spatial database is large where external memory must be used for processing MPRQ. Our equally extensive experimental results show that MPRQ-Disk promises good performance in answering MPRQ in terms of query time as well as the number of I/Os, even for the case where RRQ-Disk benefits from an implicit disk buffer that is the norm in database systems and against distance semi-join algorithms and spatial join algorithms as well.

As expected, the speed-up increases proportionally with the number of query points as well as with the query distance for MPRQ-Disk. In addition, this speed-up holds for a large variety of problem parameters: over different number of query points in the query path *P* (even for very small queries), different search distances *d*, as well as different spatial representations of the spatial database.

In the database literature, there are a plethora of spatial join algorithms for this is an active area of research. Interestingly, some spatial join algorithms can, with some modification, compete with MPRQ-Disk for solving the MPRQ problem. One example is the high-performance spatial join algorithm SJ4. Another class of spatial join algorithms seeks to join a dataset indexed in an R-tree to a non-indexed query set. It is fundamentally similar to the definition of the MPRQ problem. An example is the SISJ. Thus, we compared MPRQ-Disk to both SJ4 and SISJ. MPRQ outperforms both algorithms in three dataset to query size ratios, designed on real-life datasets that is representative of a real-life spatial join queries. The small ratio closely resembles the design of MPRQ algorithm, which was motivated by performing queries on a large dataset and a small input query.

In conclusion, our study shows that MPRQ-Disk is superior to RRQ-Disk. With this understanding, we had set out to adopt MPRQ-Disk in two applications, namely (i) RADS, which was described in Chapter 1 and (ii) the PepSOM algorithm for the peptide identification problem in bioformatics [NiNL06], which is described in Appendix A.

PART II

Reverse Nearest Neighbour

Chapter 5 RNN and Related Work

The reverse nearest neighbour (RNN) query is a relatively new area of research which was introduced by [KoMu00]. In the nearest neighbour (NN) problem, the concept of *influence* of a data point *p* in database is the notion that *p* exerts influence on its nearest neighbours; any changes in *p* might affect its neighbours, which is true for many real-world applications. Therefore much attention is focused on finding *p*'s nearest neighbours. The NN problem is a well-researched problem with many efficient *k*NN algorithms [Same06] proposed that can find the top *k*-nearest neighbours of any given point.

In contrast, the notion of influence of *p* in the RNN problem is the conjecture that as other data points exert *their* influence on *p*; when *p* changes these data points must be directly affected. This is a stronger notion of influence compared to the case of NN. For example, in a virtual reality shooting game, a smart computer gear that a player Pete wears can find and rank Pete's top *k* nearest enemies (a *k*NN query) so that Pete can shoot them (higher chance of hitting close targets). At the same time, Pete's gear will also identify all enemies whose top *k*-nearest neighbours include Pete (a R*k*NN query) so that Pete can get far away from them! The RNN query is also useful in other real-life business applications such as decision support systems, continuous referral systems, profile-based marketing and maintaining document repositories [KoMu00].

5.1 The R*k*NN Problem

The R*k*NN problem is non-trivial and more challenging than its counterpart, the *k*NN problem because it cannot be answered by simply complementing the result set or the function of *k*NN. The relationship between *k*NN and R*k*NN is asymmetrical. In fact, in the latter, spatial locality w.r.t. a query point does not apply. We illustrate this behaviour using the example in Figure 65, with $P = \{p_1, p_2, p_3, p_4, p_5\}$ and q as the query point. We observe that q becomes the NN of p_1 and p_2 w.r.t. $P \cup \{q\}$, hence the $\text{RNN}_{P\cup\{q\}}$ of q are p_1 and p_2. Note that p_1 is a RNN of q although it is far away from q but p_3 is not a RNN of q although it is closer to q than p_1 and p_2.

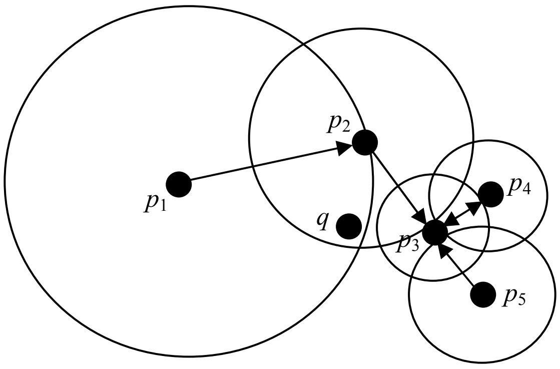


Figure 65. A reverse nearest neighbour example with $k = 1$

5.2 Formal Problem Definition

The reverse nearest neighbour (RNN) query asks the following: given a query point q, find a set of points whose nearest neighbour (NN) is q. The RNN problem is also known as finding the *influence set of q* problem.

Let *SDB* be a database of n 2-d points ($|SDB| = n$). Let d be any Minkowski metric distance function L_p on $\Re^2$ and any $x, y, z \in \Re^2$ satisfy the conditions $d(x, x) = 0$, $d(x, y) = d(y, x)$ and $d(x, z) \leq d(x, y) + d(y, z)$ in general. Before we define RNN, let us define NN. The set NN of a query point $q \in \Re^2$ is the set $\text{NN}(q) \subseteq SDB$ such that $\forall p \in \text{NN}(q)$, $\forall p' \in SDB \setminus \text{NN}(q)$, $(d(q, p) < d(q, p'))$. We generalize this to $k\text{NN}(q)$, the smallest unique set S that contains at least k points from SDB such that $\forall p \in S$, $\forall p' \in SDB - S$, $(d(q, p) < d(q, p'))$. The set of reverse nearest neighbours of a point q, $\text{R}k\text{NN}(q) = \{p \in SDB \mid q \in k\text{NN}(p)\}$. $\text{R1NN}(q)$ is correct for any arbitrary set $k\text{NN}(q)$ where the top k points with the smallest distance from q, called the k-nearest neighbour, is chosen arbitrarily. It could be possible that $|k\text{NN}(q)| > k$ if while processing the kNN query, k-1 points has been discovered and $\exists p_1, p_2 \in SDB \setminus (k\text{-}1)\text{NN}(q)$ where $d(p_1, q) = d(p_2, q)$ and $|(k\text{-}1)\text{NN}(q) \cup \{p_1\}| = k$. In this case, we terminate when $|k\text{NN}(q)| = k$ is satisfied. Figure 66 illustrates the case.

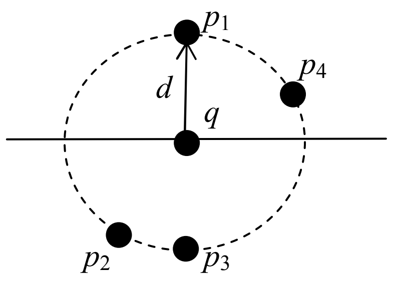


Figure 66. The case where $|k\text{NN}(q)| > k$ when $k < 4$. This is because all points p_1, p_2, p_3, p_4 lie in equal distance from q. In cases like these, an arbitrary set $k\text{NN}(q)$ of size k will be returned

In the remainder of this thesis, we write $\text{1NN}(q)$ simply as $\text{NN}(q)$, $\text{R1NN}(q)$ simply as $\text{RNN}(q)$, and d is L_2 Euclidean distance metric for illustrative purposes. The terms RNN and NN can also be taken to mean the general, respective problem. The distance function d is equivalent to the *dist* used in previous chapters.

5.3 Related Work

The naïve method to answer a RNN query is extremely slow and expensive. R*k*NN(q) can be answered by computing the *k*NN(p) for each and every data point p in the database of size n and subsequently returning all the points p whose *k*NN(p) contains q. The running time of this method is O(n^2) for linear data points and O(n log n) if the data points are spatially indexed by a height-balanced hierarchical structure such as the R-tree [Gutt84]. The space complexity is O(n) for $k = 1$ and O(n^2) for $k > 1$.

In general, the approaches to answering RNN queries can be classified into two categories: *voronoi* approach and *hypersphere* approach. Voronoi approaches use the concept of Voronoi cells to perform space pruning. Based on certain geometrical properties between data points and the properties of RNN, algorithms using this approach are able to filter off a large number of data points and keep a much smaller set of candidate points for verification. One disadvantage of space pruning approaches is that they do not scale for high-dimensional data. Hypersphere approaches use the observation that if $d(p, q) < d(p, k\text{NN}(p))$, then p is a correct answer. Algorithms using this approach usually perform pre-computation on all the points (each point up to its *k*NN for a given k) in the database and construct a spatial index with this observation embedded. The drawback of hypersphere approaches is that they cannot handle queries with an arbitrary k in which the spatial index is not constructed for.

The RNN-tree [KoMu00] was the first approach to answer RNN queries. The idea is to pre-compute the distance r of a point p to its NN and represent it by a vicinity circle (VC) with radius r centred at p. All vicinity circles VC(p, d(p, NN(p))) for all points are stored at the leaves of an R-tree. Hence, an RNN query is transformed into a point enclosure query where RNN(q) = {$p \in SDB$ | q falls inside VC(p, d(p, NN(p)))} (proof in [KoMu00]). One drawback for this method is that it requires another spatial index to handle the dynamic case where insertions and deletions to the dataset are required. This problem was circumvented by [YaLi01] who proposed the Rdnn-tree so that NN and RNN queries can be answered. As a result, only one index needs to be maintained. The Rdnn-tree is also designed to answer NN and RNN queries together in a single tree traversal. Subsequently, a bulk-loading method for the Rdnn-tree was proposed [LiNY03]. The Rdnn-tree is not easy to update as it still involves massive changes to many nodes in the dynamic case.

One huge disadvantage of the RNN-tree and Rdnn-tree is that when constructed for k = 1, they can only answer R1NN queries. To answer R2NN queries, another index must be constructed with VC(p, d(p, 2NN(p))). In general, k indexes are required to answer any arbitrary RkNN(q), which is impractical.

The first RNN algorithm taking the Voronoi approach was proposed in [StAE00, SRAE01]. Assuming Euclidean distance metric, the space around a query point q is divided into 6 equal *constrained regions* of 60° each (Figure 67), and it can be proven that in each region S_i, either there exist one and only one point $p_i \in S_i$ such that $p_i \in$ RNN(q), or such a p_i does not exist at all in S_i [StAE00]. In a later work [SRAE01], a coarse filtering and refinement algorithm was proposed taking advantage of the results in [StAE00] to answer RNN queries. Constrained regions present a well-known fact about RNN, i.e.

in a plane, there can only be at most 6 RNNs for Euclidean metric and at most 8 RNNs for Manhattan metric. However, the number of RNNs is unbounded for R*k*NN where $k > 1$. The problem of constrained regions is that they suffer from the *curse of dimensionality*. The number of regions to be searched increases exponentially with dimensionality.

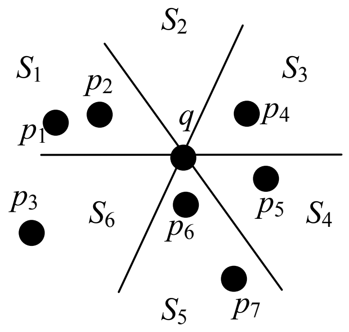


Figure 67. Example of constrained regions around a query point q using Euclidean metric in 2-d space

The idea of using the perpendicular bisector plane for pruning was proposed in [TaPL04] as the TPL algorithm. TPL works only on points indexed by an R-tree. It first retrieves a set of potential candidates into S_{cnd} in ascending order of their distance to the query point q. Candidate points in S_{cnd} are pruned against each other and also against already seen points in a refinement set S_{rfn}. Pruned items are inserted into S_{rfn}. MBRs, however, are "half-pruned" into residual area by the perpendicular bisector idea when a new point is discovered. Subsequently, an MBR gets smaller when more points causes it to be further reduced in size, if not eliminated altogether. TPL is illustrated in Figure 68. The disadvantage of TPL is that it is being too paranoid by saving all pruned items in S_{rfn} and nothing is ever discarded. As a result, although the size of S_{cnd} is kept to a minimum, the refinement step is too cumbersome as S_{rfn} (used for future pruning) grows very quickly. It is not unrealistic to imagine that S_{rfn} may well outgrow main memory allocation, although this issue is not addressed in the paper.

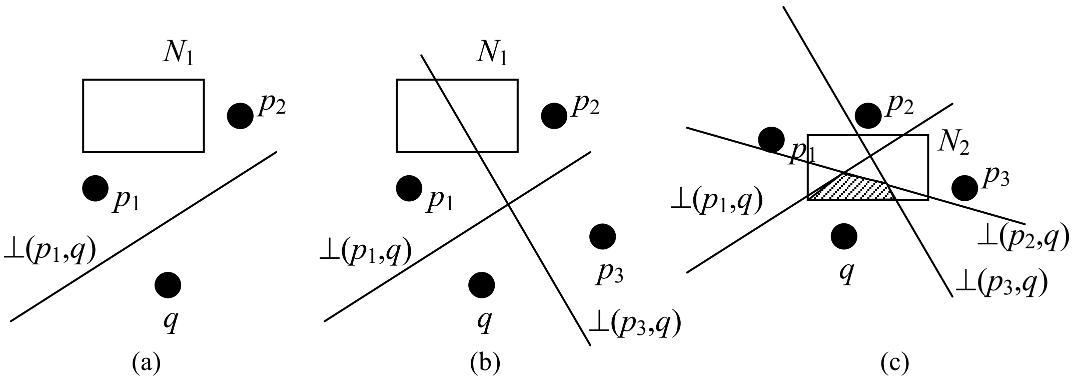


Figure 68. The TPL algorithm. (a) A bisector perpendicular line $\perp(p_1,q)$ prunes off half the space. Point p_2 and MBR N_1 are both nearer to p_1 than q, therefore can be pruned (b) When p_3 is discovered, a new $\perp(p_3,q)$ is introduced leading to more pruned space where RNN cannot exist (c) An MBR N_2 is pruned by three bisector perpendicular lines, only the points that fall in the residual area (shaded) can be the result

The existing RNN solutions mentioned so far either rely on pre-computation which is expensive to maintain in a dynamic setting where frequent updates are required [KoMu00, YaLi01, LiNY03], or are applicable only in Euclidean space in which similarity is based on the L_2 norm [StAE00, SRAE01, TaPL04]. In [TaYM06], techniques for answering RNN that solves these issues were presented. The work is in general metric spaces that assumed no detailed representation of the data objects, instead the only sufficient conditions are that (i) there exist a computable distance between any two data objects that satisfies the triangle inequality property, and (ii) the distance can be indexed. The data structure used is the M-tree [CiPZ97] as it is a dynamic structure specifically designed for external-memory access and it aims to minimize the overlap among the cluster of indexed spheres. Since the authors did not name their algorithm, we simply christen it TYM in this thesis.

So far, all the abovementioned RNN algorithms provide an exact answer for the RNN of a query point q. In contrast, there is another class of RNN algorithms that aims to be fast but will only provide approximate RNN results [SiFT03, XiHL05, AFST07]. Among these approaches, [XiHL05] introduces ERkNN, an efficient algorithm that can be implemented on the widely-used R-tree, hence its immense potential. The algorithms proposed in [SiFT03, AFST07] are extremely slow and their performance depends heavily on the non-trivial efficient implementation of boolean range query. For this reason, we chose ERkNN for further discussion.

ERkNN is shown to be an order of magnitude faster than [SiFT03], with better recall too. It is also faster than the TPL algorithm in terms of processing time, mainly because ERkNN is an approximate method. ERkNN uses a local *k*NN-distance estimator utilising PDE (parzen density estimator with uniform kernel [Fuku90]) or kDE (*k*NN density estimator [Fuku90, KaSa01]) to retrieve R*k*NN candidates. The local *k*NN-distance is the distance from a data point to its *k*-th nearest neighbour, estimated by a density function of a small number of neighbouring samples around the query point q. The advantage is estimation-based filter has a lower computation cost than space pruning strategies. In the coarse filtering step, ERkNN retrieve a set of candidates p_i whose distance to q is equal to or greater than p_i's estimated *k*NN-distance as R*k*NN candidates. In the refinement step, range queries are used to verify the candidates.

5.4 Variants of the RNN Problem

There are many other variants of the RNN problem which are beyond the scope of this research. For instance, the bichromatic-RNN problem [SRAE01, KMSX07], RNN in graphs [YPMT06] and continuous RNN monitoring for a moving query point [XiZh06, BJKS07, KMSX07, WYCT08].

In the bichromatic-RNN problem, given a set *TDB* of sites, a set *SDB* of points, and a query site q, B-RNN(q) finds all points that have q as their nearest neighbour site, i.e. B-RNN(q) = $\{p_i \in SDB \mid \forall s \in TDB, d(q, p_i) \leq d(p_i, s)\}$. The set *TDB* of sites can be viewed as blue-coloured points whereas the set *SDB* can be viewed as red-coloured points (hence, bichromatic) and the goal is to retrieve all red-coloured points closer to q than to any blue-coloured points.

For the continuous-RNN problem, given a set *SDB* of points, some time interval T_j and moving query point q, the goal is to keep track of $\text{RNN}_j(q)$ where $\text{RNN}_j(q) = \{p_i \in SDB \mid \forall o \in SDB, d(q, p_i) \leq d(p_i, o)\}$ at time interval T_j.

5.5 Summary of RNN Algorithms

The RNN algorithms found in the literature can be broadly classified by three properties they possess. Table 17 shows the summary of the RNN algorithms, which was first compiled by [TaPL04] and expanded here to cover some of the newer published work together with our proposed novel RNN algorithms, RNN-Grid and RNN-C tree.

Table 17. Non-exhaustive list of RNN algorithm summary properties adapted from [TaPL04], and expanded. This list only includes monochromatic RNN algorithms for static query points

RNN Algorithm	Dynamic data	Arbitrary dimensionality	Exact result
KoMu00 (RNN-tree)	✗	✓	✓
StAE00	✓	✗	✓
YaLi01 (Rdnn-tree)	✗	✓	✓
MaVZ02[†]	✗	✗	✓
SiFT03[†]	✓	✓	✗
TaPL04 (TPL)	✓	✓	✓
XiHL05 (ERkNN)	✓	✓	✗
TaYM06 (TYM)	✓	✓	✓
RNN-Grid	✓	✓	✗
RNN-C tree	✗	✓	✓

To the best of our knowledge, apart from the RNN-C tree, only TYM is designed for solving RNN in metric space. However, there are several major differences between RNN-C tree and TYM though. The construction of RNN-C tree is based upon 1NN distance, and the final data structure is independent of the order of data points. A RNN-C tree is constructed bottom-up while M-tree is built from top-down. Due to this, the RNN-C requires no split policy. The construction algorithm in M-tree tries to avoid enlarging the covering radius when adding a node, and if that is not possible, try to minimise the covering radius enlargement.

The centre of a cluster (centroid) is not a member of the cluster in RNN-C tree but for TYM, the centre of a node (called routing object) is one of the points in the intermediate entry. The fanout of a node $f_{\min}$ has no direct relationship with k in RNN-C tree but in TYM, the algorithm is designed with the assumption that $k < f_{\min}$. The key difference in pruning strategy is that RNN-C tree makes use of *the sum of clusters* to prune clusters at all levels,

[†] These R*k*NN work are not covered in the related work section. For further information, refer to the paper.

while TYM uses a node's parent distance to save on the cost of distance computation. TYM was not able to make use of node size for pruning.

5.6 Statistical Analysis

We propose a method to answer RNN queries based on parameter extraction approach. A parameterised function fitting the correlation between k_1NN and Rk_2NN is designed to be used in our novel algorithm, RNN-Grid (note that we distinguish both k's in this section). The goal is to find such a function so that for some given confidence value, we could retrieve the number of k_1 NN candidates such that Rk_2NN can be answered with certainty. As a result, the RNN-Grid is a fast, approximate RNN algorithm as it uses the resulting table from the parameterised function. Section 5.6.1 details the correlation analysis.

Section 5.6.2 describes an analysis of the randomness of clusters formed by representative points picked from a cluster. Given a uniformly distributed spatial dataset S_1 and the 1NN graph is built on S_1, this will result in a graph of many disjoint components (called clusters) C_i. Suppose the centroid c_i is computed from each of the C_i to form a dataset S_2, a 1NN graph is built on S_2, and the process is repeated until $|S_j| \leq 3$, are all the points in S_j ($j > 1$) random? How does the size of clusters reduce from one level to the next? This is an important factor that determines the height of our proposed RNN-C tree, with respect to the dataset size.

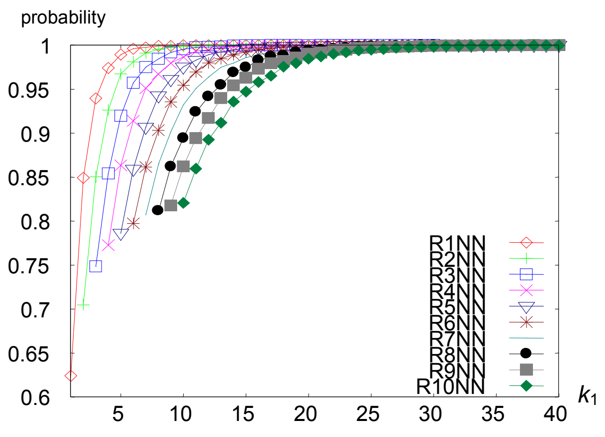
5.6.1 Correlations between NN and RNN

We conducted a study of the correlations between k_1 and k_2 by defining a function $f(k_1, k_2)$ which returns the probability value $\mathrm{P}(\mathrm{R}k_2\mathrm{NN}(q) \subseteq k_1\mathrm{NN}(q))$. The probability value is calculated using the formula

$$\mathrm{P}(\mathrm{R}k_2\mathrm{NN}(q) \subseteq k_1\mathrm{NN}(q)) = \frac{k_1\mathrm{NN}(q) \cap \mathrm{R}k_2\mathrm{NN}(q)}{\mathrm{R}k_2\mathrm{NN}(q)} \text{ or 0 if } \mathrm{R}k_2\mathrm{NN}(q) = \varnothing$$

During the study, we set $k_1 = k_2 \ldots 100$ and $k_2 = 1 \ldots 100$, and measured the average probability value. For the analysis, we studied three types of data distributions: uniform, normal (Gaussian) and real-life data from the TIGER/Line database [TIGER02]. For uniform and normal distributions, 10K data points and a single query point q were generated per measurement. The process is repeated 100 times to obtain the average, for $k_1 \geq k_2$. For real-life data, we used datasets from 4 counties to obtain the average, and only the query point is regenerated 100 times per dataset since the datasets are static. In each run, all the data points are subjected to the naïve method to compute their NNs for the RNN results. This method has a running time of $O(n^2)$, using $O(n.k_1)$ space.

In Figure 69, the probability curves were plotted for $k_2 \leq 10$, against $k_1 \leq 40$ for the uniform and normal distributions. Each line represents the average $\mathrm{P}(\mathrm{R}k_2\mathrm{NN}(q) \subseteq k_1\mathrm{NN}(q))$, for increasing k_1. Recall that $\mathrm{R}k\mathrm{NN}(q) = \{p \in SDB \mid q \in k\mathrm{NN}(p)\}$. Therefore, there are no probability values for $k_1 < k_2$ because the definition of $\mathrm{R}k\mathrm{NN}(q)$ is undefined (we need exactly k NN to make the set definition).



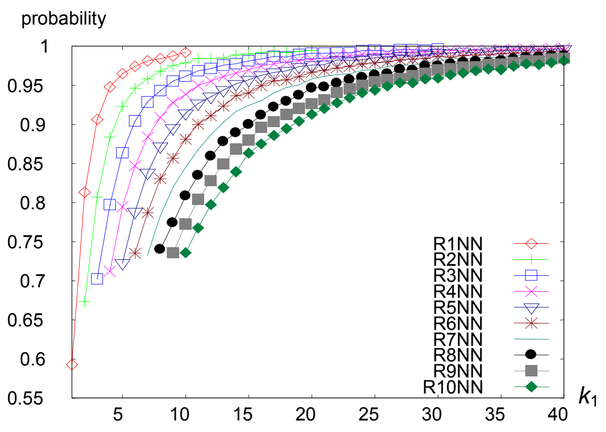


Figure 69. Correlation analysis between NN and RNN for uniform (left) and normal (right) distributions. The chart plots the probability values against the number of NN (k_1). Each line represents a k_2 value

The R1NN lines begin with 0.6242 and 0.5928 for uniform and normal distribution respectively and reaches 1.0 when $k_1 = 12$. The probability value stays for $k_1 > 12$. It is generally observed that the probability value stays at 1.0 for subsequent k_1 values once it is reached, for all k_2 and all three data distributions. In fact, the starting probability value (when $k_1 = k_2$) increases gradually from 0.6242 to 0.8207 and stabilises at ≈0.8319 (for very large k). This is taken to mean that we will obtain 83% of the RNN results (for any $k \geq 10$) if we were to find the top k NN. Generally, to obtain all RNN results with any λ certainty, one would have to find the k_1 value that corresponds to $P \geq \lambda$.

In the normal distribution, the starting values for $5 \leq k_2 \leq 10$ are 0.7225, 0.7353, 0.7327, 0.7399, 0.7357 and 0.7359 respectively. This is approximately 10% lower than those for the uniform distribution. The reason for this is that our query point q is uniformly random in the plane, and at the edge of the plane, q is slightly disadvantaged by the sparse points inside the plane and no points outside the plane. The analysis also confirms our conjecture that when $k_1 >> k_2$, the quality of the results improve greatly. For instance, when k_2 = 5 and k_1 = 5, the P values are 0.7862 and 0.7223 for uniform and normal distribution respectively. But when k_2 = 5 and k_1 = 15, the P values rise to 0.9969 and 0.9570 respectively.

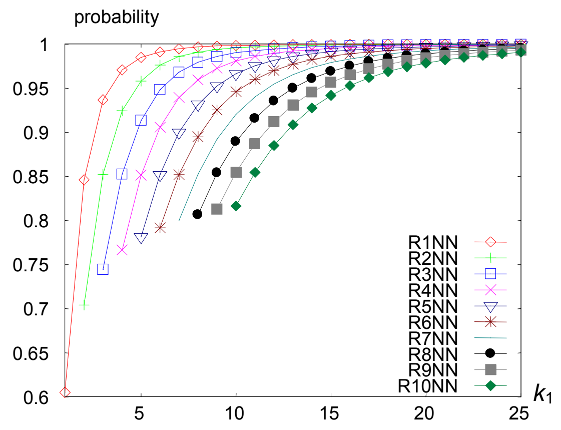


Figure 70. Correlation analysis between NN and RNN for 4 real-life datasets. The chart plots the probability values against the number of NN (k_1). Each line represents a k_2 value

The trend for real-life datasets is similar to those of uniform distribution, except with a lower R1NN starting value of 0.6052. It exceeds 0.995 at k_1 = 8, exceeds 0.9995 at k_1 = 8, and reaches 1.0 only at k_1 = 53. The real-life datasets also conform to our conjecture that when $k_1 >> k_2$, the quality improves greatly. With this conjecture proven empirically, we could conclude that finding a much higher number of *q*'s NNs will increase the chances of getting the required R*k*NNs during the query.

The results in these analyses were inserted into a probability chart in our RNN-Grid codes, accessible via a lookup function which is the first line of the pseudocode in Figure 72. The confidence level *c* is a user-supplied parameter to obtain the desired level of R*k*NN results. The higher the value of *c*, the more candidates will be returned, leading to a higher chance of obtaining the correct RNN results but at the expense of higher processing costs.

5.6.2 Randomness of Clusters

[EpPY97] had studied data points representation similar to *k*NN graph and presented the theoretical result $|C_{i+1}| \approx 0.31*|C_i|$ which guarantees finite RNN-C tree height. The equation says that the number of clusters is approximately 1/3 of the number of data points at any level, which means there are 1/3 of the data points on a level above, to construct *k*NN graphs with. In other words, by designing effective pruning rules for pruning clusters in a RNN-C tree, the query algorithm can potentially prune off data points 3 times the size of a cluster. The higher the tree level in which pruning takes place, the more data points are pruned off as each cluster contains points that in turn represent even smaller clusters.

One concern that motivated this analysis is the "randomness" of clusters formed by centroids of a cluster. Although at the leaf level the points may be random, the randomness of centroids of clusters is unknown. This is even more interesting when the clusters are not formed by random points at the leaf level, but instead by points with geographical significance. Hence, we conducted an analysis of both synthetic and real-life spatial datasets to measure the randomness of clusters.

For the synthetic datasets, we randomly generated 2^i*1000 $(0 \le i \le 6)$ 2-d points on a plane of 10000^2 unit sq and constructed the *k*NN graphs to form clusters. For each cluster, a centroid is calculated and propagated one level up to represent dataset points for another round of constructing *k*NN graphs. This is repeated until less than three points are left. At each level, the points are generated and built 100 times and the average and standard deviation are recorded. The aim is to compute the ratio of reduction to see how close it is to the theoretical result. The same process is repeated for two real-life spatial datasets, MD and RI (refer to Table 10), except that they are run only once per level as the data points are static. Although results for random points are backed by theory, it is interesting to see whether real-life datasets display the same traits; if not, how different the ratio would be.

Table 18. Synthetic datasets of randomly generated points of size 2^i*1000 ($0 \le i \le 6$) and their standard deviation at different levels of the *k*NN graphs (level 0 is the leaf level). The ratio of the size to its lower level is also calculated

Level	Size	Ratio	Std dev	Size	Ratio	Std dev	Size	Ratio	Std dev	Size	Ratio	Std dev	Size	Ratio	Std dev	Size	Ratio	Std dev	Size	Ratio	Std dev
0	1000			2000			4000			8000			16000			32000			64000		
1	311.70	0.31	7.43	619.90	0.31	9.00	1242.11	0.31	13.87	2477.2	0.28	13.98	4965.4	0.31	27.85	9960.4	0.31	24.79	19901.8	0.31	82.80
2	89.17	0.29	4.50	177.93	0.29	7.38	354.42	0.29	9.01	702.4	0.28	11.04	1409.7	0.28	11.01	2854	0.29	9.32	5690.4	0.29	48.77
3	25.61	0.29	2.76	50.41	0.28	3.91	99.61	0.28	4.60	197.2	0.27	6.49	392.5	0.28	9.40	800.4	0.28	11.66	1585.2	0.28	15.17
4	7.32	0.29	1.45	14.51	0.29	2.02	28.13	0.28	2.61	55.2	0.28	2.04	106.1	0.27	4.90	225.1	0.28	8.07	440.7	0.28	5.90
5	2.19	0.30	0.83	4.21	0.29	1.10	7.99	0.28	1.49	16.4	0.30	1.36	30.1	0.28	3.29	63.4	0.28	3.61	120.4	0.27	3.67
6				1.29	0.31	0.50	2.45	0.31	0.77	4.4	0.27	1.02	8.2	0.27	1.72	19.8	0.30	2.76	35.4	0.29	2.15
7										1.4	0.32	0.80	2.6	0.32	1.36	5.4	0.28	0.80	10.6	0.30	1.36
8																1.6	0.31	0.52	2.8	0.26	1.17

The theoretical result could be observed from the calculations in Table 18. The ratio $|C_{i+1}|/|C_i|$ lies in the range of 0.26 to 0.32, with a mean of 0.29 over all datasets and all levels. There is no significant difference between a small dataset (1000 points) and a large dataset (64000 points) except the resultant tree height. The standard deviation averaged 0.96% and 2.18% for levels 1 and 2 respectively, which means the average ratio presented is truly a good representation.

Table 19. Two real-life dataset MD and RI used to construct *k*NN graphs.

Level	MD	Ratio	RI	Ratio
0	28719		53721	
1	8690	0.30	16446	0.31
2	2367	0.27	4573	0.28
3	625	0.26	1252	0.26
4	155	0.25	330	0.25
5	41	0.26	82	0.30
6	13	0.32	25	0.30
7	3	0.23	5	0.20

Table 19 shows the analysis for the real-life datasets. The ratio of clusters reduction ranges between 0.20 and 0.31, with a mean of 0.27 for both. This is quite close to the theoretical result but we cannot draw any conclusions because real-life datasets vary to a great extent. However, from this analysis, it is at least observable that cluster sizes reduce by at least 60% or more at all levels. This guarantees that the RNN-C tree that we propose has height that is not only finite, but logarithmic as well.

In the next two chapters, we will make use of the results of the statistical analyses in this section to design both an estimated and an exact approach to answering the RNN query, namely the RNN-Grid and RNN-C tree respectively.

Chapter 6 RNN-Grid: An Estimated Approach for RNN Query

In many applications, response time is critical but the accuracy of the RNN results is not. A virtual reality shooting game designer may want to find out the top *k* RNN of the player quickly with some high probability (say, 0.95), rather than tying up resources to find all the RNN with absolute certainty because there are many other aspects of the game that require the same resources. Besides, a "missed" RNN that suddenly appears may well become an element of surprise for the player (enhanced playability) and there is also a good chance of the player shooting down this enemy first (even without the knowledge that the enemy is a RNN). For applications that require fast, approximate results, we proposed an approach based on the grid file data structure [NiHS84].

6.1 The Grid File

The grid file is an elegant data structure that is easy to adopt and implement. It is an intuitive method for solving the NN problem as it provides fast O(1) access to cells (buckets containing data points). So, given a query point *q*, the cell where *q* is located can be retrieved immediately and the data points in surrounding cells be investigated. We begin by describing the grid file data structure.

The grid file is a relaxation of the fixed grid method to allow free distance in all *k* axes. In the original fixed grid data structure, all axes are partitioned into fixed intervals. Although the apparent advantage is the ease of cell referencing (where a simple formulae effectively pinpoints the correct cell for any point of any dimensionality), its drawback far outweighs its usefulness. A set of heavily skewed dataset would cause most data points to fall into just a few cells, effectively turning the fixed grid into a sequential search. The grid file solves this problem by allowing the freedom for the axes to be flexible, in tandem with the data points being inserted. To keep track of the axes, *k* additional arrays (called linear scales) are maintained to find the grid partitions.

Age	Salary
24	1025
40	510
46	975
59	650
65	425

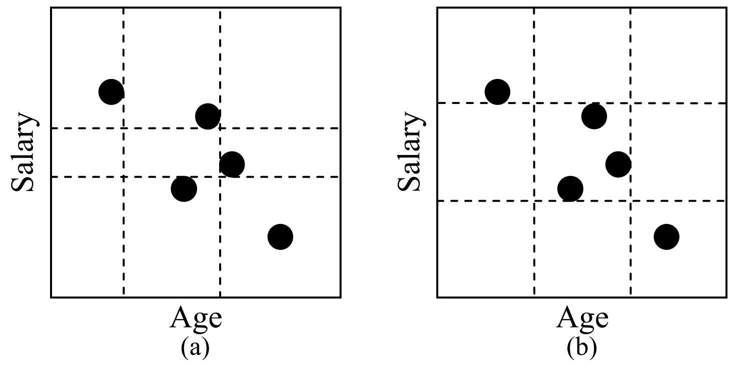


Figure 71. An example of (a) grid file and (b) fixed grid. By allowing flexible axes, the data points can be split into the partitions evenly. In the fixed grid, it is difficult to find a fixed interval so that all data points are evenly distributed

The main objective of the grid file is to evenly spread the data points into all its cells, thereby guaranteeing optimal I/O costs. The grid file slices the space of points in each of the *k* dimensions, producing partitions of rectangles (for *k* = 2), cubes (*k* = 3) or hypercubes (*k* > 3). Each partition is a bucket, and points that fall in that partition have their record placed in a block belonging to that bucket. To perform a search, we first need to determine the positions of the record in each of the *k* dimensions according to the linear scales. After locating the proper bucket in the grid array, the data block is finally accessed on disk.

For example, a 2-d grid file dynamically partitions the search space by maintaining a structure of two 1-d arrays and one 2-d array. The former is called linear scales, and is used to maintain a partitioning of unequal cells such that data points in them are spread evenly and each cell of the grid array points to a single data bucket (more than one cell can point to the same bucket). It shall be noted that the partitioning is highly dependent on the order in which data points are inserted and the bucket size. In general, grid files have a 70% utilisation.

Methods for insertion and deletion of data for the grid file have been proposed. In the best case, insertion operations on the grid file cost one I/O for accessing the linear scales (if they do not fit into memory) and another I/O for accessing the bucket, assuming that the new point does not make the bucket full. In the worst case, full buckets need to be split, causing linear scales to be adjusted. Deletion operations on the grid file in the worst case cause buckets to fall below a threshold utilisation value; they are merged with adjustments to the linear scales.

6.2 RNN-Grid Algorithms

As shown in the problem formulation, the RNN problem is interrelated to the NN problem. Taking advantage of this correlation, we designed an algorithm, which we call RNN-Grid, that makes use of the grid file for solving the NN problem and adapted it further to solve the RNN problem. To the best of our knowledge, no RNN algorithms have been designed around the grid file despite its obvious potential as an approximate approach to the RNN problem.

The key idea for the RNN-Grid is to quickly return the set k_1NN(q) as candidates hopefully large enough to cover all the Rk_2NN(q). Given the spatial database of multi-dimensional points *SDB* in a grid file data structure, the query point q and k_2, our RNN-Grid algorithm will make use of the statistical analysis results to derive k_1, the suitable number of NN to retrieve as candidates. In the refinement step, for each candidate $p \in k_1$NN(q) the same RNN-Grid algorithm is again invoked to check whether $q \in k_2$NN(p). If this condition is met, p is a true positive. It is easy to see that the accuracy of this estimated RNN-Grid approach stems from the value k_1, which in turn is based on statistical analysis. Figure 72 illustrates the basis for RNN-Grid algorithms.

The RNN-Grid algorithms were designed with the underlying assumption that the dataset is 2-d data points and there exists a distance function satisfying the triangle inequality principle. Firstly, we explored two methods for RNN-Grid, best-first wavefront (BFW) and best-first cell expansion (BFCE). Both methods made use of our probability statistical analysis results in Section 5.6.1 for generating candidates. Recall that the analysis provides us with an estimator for the set k_1NN(q) given a confidence value. Experiments have shown that BFCE outperforms BFW, so the former was chosen as the *de facto* algorithm for the RNN-Grid approach. Next, the BFCE method was combined with theorems on pruning with the geometrical RNN properties that were described in [StAE00, TaPL04] to further improve its performance.

```
RNN-Grid(q, $k_2$, c, R)
// Input:  Query point q, the $k_2$-th RNN, confidence c (0-1)
// Output: R – R$k_2$NN of q
begin
  $k_1$ ← lookup($k_2$, c)  // $k_1$ is the number of NNs required
  RNN-Grid-algorithm(q, $k_1$, temp)

  forall p in temp do
     temp2 ← ∅
     RNN-Grid-algorithm(p, $k_2$+1, temp2)
     // +1 because p is in the dataset and must be discounted
     last ← pop(temp2)  // furthest point in temp2 from q
     if dist(q, p) < dist(p, last) then
        R ← R ∪ {p}
     endif
  endfor
end; {procedure RNN-Grid}
```
Figure 72. Basis pseudocode for all the RNN-Grid algorithms (BFW, BFCE, BFCE-PB) except BFCE-CR

RNN-Grid relies on the grid file's inherent insertion and deletion methods to deal with dynamic data, just as other RNN algorithms in [TaPL04, XiHL05] rely on its underlying R-tree data structure's insertion and deletion methods to handle dynamic data.

6.2.1 Best-First Wavefront (BFW) Algorithm

BFW represents a first attempt at solving RNN with the grid file. The key idea in BFW is to locate the cell in which q is located and expand in a rectangular fashion outwards (in waves) to find the k_1 required NN of q. In each wave, *all* cells must be processed to be considered complete. Figure 73 shows an example of BFW. At wave $w > 0$, there are exactly $8w$ cells to process. Let c_i be any cell at wave i, in which all are sorted in ascending order of distances from q, i.e. MinDist(q, c_i). The function MinDist is similar to the one defined in Figure 14. We also maintain a global `CurrMinDist` value, defined as the smallest distance between q and the k-th ($k \leq k_1$) valid results so far. `CurrMinDist` is used to prune both unseen cells and data points. A queue Q is used to process the cells; when a cell c_i is dequeued to be processed, it will be processed if MinDist(q, c_i) < `CurrMinDist`. Let data point $p \in c_i$. p is added to the result set if $d(p, q)$ < `CurrMinDist`. The algorithm will terminate when $\forall c_i$, MinDist(q, c_i) > `CurrMinDist`.

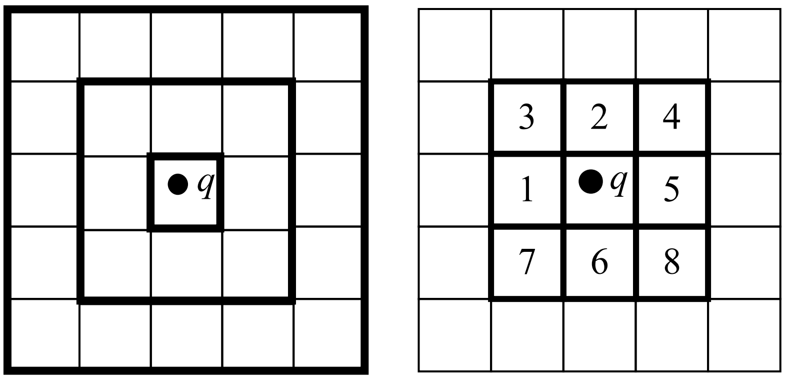


Figure 73. Best-First Wavefront (BFW) algorithm for RNN-Grid. (a) Each wave consists of cells one unit adjacent to the cell of q in the beginning and to the previous wave subsequently. (b) Cells within a wave is maintained and visited/processed in the ascending order of their distances from q. Note that in a real grid file, the cells are not likely to be squares; the example is for illustration only

The algorithm for BFW is given in Figure 74. For simplicity, the part where a counter can be kept to check the algorithm's termination condition is omitted. One disadvantage of BFW is that once a wave is started, all the cells must be processed and the algorithm terminates when the whole wave's cells is further than `CurrMinDist`. This may incur unnecessary computation costs.

```
RNN-Grid-BFW(q, k, R)
// Input:  Query point q, the k-th RNN
// Output: R - Rk₂NN of q
begin
  currMinDist ← 0
  w ← 1  // wave

  Q ← getCell(q.x, q.y)  // returns cell where q is located
  while not Q.isEmpty do
     cell ← dequeue(Q)
     cand ← getBucket(cell)  // get all points from cell
     if |R| ≥ k and MinDist(q, cell) > currMinDist then
        continue;  // proceed to next cell
     endif
```

```
      ProcessCandidates(q, k, cand, R, currMinDist)

      for i in -1*w to 1*w do
         for j in -1*w to 1*w do
            if i=0 and j=0 then continue;  // skip middle cell
            cell ← getCell(q.x+j, q.y+i)
            if not cell exist then
               continue;  // cell might be at grid boundary
            endif
            if |R| ≥ k and MinDist(q, cell) > currMinDist then
               continue;  // proceed to next cell
            endif
            insert cell into Q sorted by $c_i$∈Q|dist(q, $c_i$)
            w ← w + 1
         endfor
      endfor
   endwhile
end; {procedure RNN-Grid-BFW}

ProcessCandidates(q, k, cand, R, currMinDist)
begin
   forall p in cand do
      if |R| < k or dist(q, p) < max{dist(q, p'∈R)} then
         insert p into R sorted by $c_i$∈R|dist(q, $c_i$)
         currMinDist ← max{dist(q, p'∈R)}
      endif
   endfor
   if |R| > k then
      truncate R at position k+1;  // keep the first k results
   endif
end; {procedure ProcessCandidates}
```

Figure 74. The Best-First Wavefront (BFW) algorithm for RNN-Grid

6.2.2 Best-First Cell Expansion (BFCE) Algorithm

The BFCE algorithm is an improvement over BFW. The key motivation is to find a way to process the cells efficiently and terminate as soon as we have enough results guaranteed to be correct. So, the improvement in BFCE stems from the idea that when processing a cell c, insert the neighbouring cells of c into the queue (of course, insertion is still subject to `CurrMinDist` which acts as baseline pruning). The visit order of cells to be processed in the queue Q is still as per MinDist(q, c) $\forall c \in Q$. However, the BFCE algorithm no longer reaches out in rectangular waves; instead it expands by aggressively inserting neighbouring cells of the current cell c into Q (if they are not already in Q). Figure 75 better illustrates an example of BFCE.

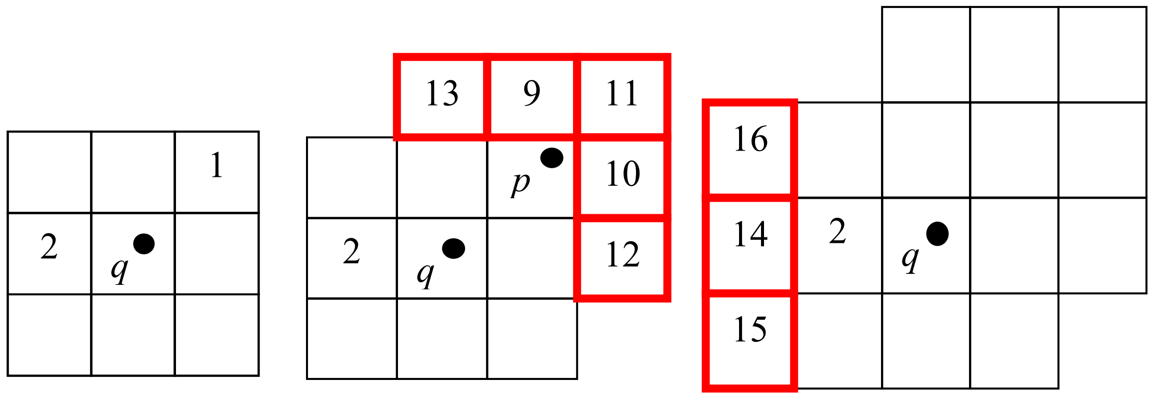


Figure 75. Best-First Cell Expansion (BFCE) algorithm for RNN-Grid. (a) In the beginning, the entire cells one unit adjacent to q is inserted into queue Q in ascending order of their distances to q. Note that not all cell index numbers are shown. (b) Next, we process the nearest cell (1) and found a point p. All cells not in Q are inserted, again in ascending order of their distances to p. (c) We then process the next nearest cell (2) and expand accordingly. Note that the number in the red cells indicates the order in which they are inserted

During cell expansion, only selected adjacent cells qualify to be inserted into the processing queue Q. They are (i) unseen or newly identified cells, and (ii) cells c_i that satisfy the condition MinDist(p', c_i) < `CurrMinDist` where p' belongs to the current cell being processed. In the actual implementation, a set T is used to keep track of cell index numbers of cells that were discarded, so that a cell is not re-inserted into Q again as any given cell in the grid is neighbour to 8 (or less, if at grid boundary) other cells. Meanwhile, the queue Q holds all cells that were found, but not yet processed. Note that not all cells will contribute to the expansion. Some cells, when dequeued for processing, might have neighbour cells that were either fully enqueued (thus waiting to be processed) or fully discarded (pruned off) or a mixture of both. The manner of cell expansion is also not contiguous. Cells are always being expanded in ascending order of the distance to the current pivot point, as shown in Figure 75(b). The algorithm for BFCE is shown in Figure 76.

```
RNN-Grid-BFCE(q, k, R)
// Input:  Query point q, the k-th RNN
// Output: R – Rk$_2$NN of q
begin
  currMinDist ← 0
  T ← ∅  // keeps track of processed cells

  Q ← getCell(q.x, q.y)  // returns cell where q is located
  while not Q.isEmpty do
     cell ← dequeue(Q)
     cand ← getBucket(cell)  // get all points from cell
     if |R| ≥ k and MinDist(q, cell) > currMinDist then
        continue;  // proceed to next cell
     endif
     ProcessCandidates(q, k, cand, R, currMinDist)
     T ← T ∪ {cell}

     for i in -1 to 1 do
        for j in -1 to 1 do
           if i=0 and j=0 then continue;  // skip middle cell
           cell ← getCell(cell.x+j, cell.y+i)
           if not cell exist then
              continue;  // cell might be at grid boundary
           endif
           if cell ∈ Q or cell ∈ T then
              continue;  // cell in processing queue/processed
           endif
           if |R| ≥ k and MinDist(q, cell) > currMinDist then
              continue;  // proceed to next cell
           endif
           insert cell into $Q_{temp}$ sorted by $c_i∈Q_{temp}$|dist(q, $c_i$)
           append $Q_{temp}$ to Q
        endfor
     endfor
  endwhile
end; {procedure RNN-Grid-BFCE}

ProcessCandidates(q, k, cand, R, currMinDist)
begin
  forall p in cand do
     if |R| < k or dist(q, p) < max{dist(q, p'∈R)} then
        insert p into R sorted by $c_i$∈R|dist(q, $c_i$)
        currMinDist ← max{dist(q, p'∈R)}
     endif
  endfor
  if |R| > k then
     truncate R at position k+1;  // keep the first k results
  endif
end; {procedure ProcessCandidates}
```

Figure 76. The Best-First Cell Expansion (BFCE) algorithm for RNN-Grid

6.2.3 BFCE with Perpendicular Bisector (BFCE-PB) Algorithm

Since BFCE performs better than BFW as our experiments have shown, BFCE was selected for further improvement. As described before, BFCE aggressively expands the cells with respect to the data points of a cell inside the processing queue. Although `CurrMinDist` acts as a baseline to prevent cells from being inserted into the queue, it does not help prune off cells fast enough. Hence, we adapted the idea of a perpendicular bisector pruning from the TPL algorithm [TaPL04] for a faster pruning. One can see that once a half-plane, defined as the line that halves the space between two data points, is marked, one-half of the search space will be pruned forever. Therefore, we chose to adapt it into BFCE for maximal pruning.

As the central idea in the TPL algorithm is to demarcate an MBR using multiple perpendicular bisector lines into a residual polygonal area (which may still contain valid RNN results), it only works with R-tree data structures. A main difference of our implementation of perpendicular bisector pruning is that we chose not to create any residual areas from cells, chiefly because it is too costly to maintain them and the cells in a grid file is non-hierarchical in nature (once a cell is pinpointed, all data points within the bucket would have been retrieved). Another difference of our adaptation is that we do not maintain a large S_{rfn} set where all pruned MBRs and data points are sent to. Using simple heuristics, data points are discarded from the S_{rfn} when they can no longer be a true RNN of q.

```
RNN-Grid-BFCEPB(q, k, R)
// Input:  Query point q, the k-th RNN
// Output: R - Rk$_2$NN of q
begin
  currMinDist ← 0
  PS ← ∅  // set of pruned data points
  T ← ∅   // keeps track of processed cells

  Q ← getCell(q.x, q.y)  // returns cell where q is located
  while not Q.isEmpty do
     cell ← dequeue(Q)
     cand ← getBucket(cell)  // get all points from cell
     if |R| ≥ k and MinDist(q, cell) > currMinDist then
        continue;  // proceed to next cell
     endif
     ProcessCandidates(q, k, cand, R, currMinDist, PS)
     T ← T ∪ {cell}

     for i in -1 to 1 do
        for j in -1 to 1 do
           if i=0 and j=0 then continue;  // skip middle cell
           cell ← getCell(cell.x+j, cell.y+i)
           if not cell exist then
              continue;  // cell might be at grid boundary
           endif
           if cell ∈ Q or cell ∈ T then
              continue;  // cell in processing queue/processed
           endif
           forall p in PS do
              terminate ← false
              if MinMaxDist(p, cell) < dist(p, q) then
                 terminate ← true
                 break;  // proceed to next cell
              endif
              if terminate then continue; // proceed next cell
           endfor
           if |R| ≥ k and MinDist(q, cell) > currMinDist then
              continue;  // proceed to next cell
           endif
           insert cell into Q$_{temp}$ sorted by c$_i$∈Q$_{temp}$|dist(q, c$_i$)
           append Q$_{temp}$ to Q
        endfor
     endfor
  endwhile
end; {procedure RNN-Grid-BFCEPB}

ProcessCandidates(q, k, cand, R, currMinDist, PS)
begin
  forall p in cand do
     if |R| < k or dist(q, p) < max{dist(q, p'∈R)} then
        insert p into R sorted by c$_i$∈R|dist(q, c$_i$)
        currMinDist ← max{dist(q, p'∈R)}
     else
        UpdatePrunedSet(q, p, PS, k)  // make use of p
     endif
  endfor
  if |R| > k then
     truncate R at position k+1;  // keep the first k results
  endif
```

```
end; {procedure ProcessCandidates}

UpdatePrunedSet(q, z, PS, k)
// update the PS set with newcomer data point z
begin
  touchedCounter ← 0
  //counts how many points in PS is closer to z than z is to q
  forall p in PS do
     if dist(z, p) < dist(z, q) then
        touchedCounter ← touchedCounter + 1
        if touchedCounter ≥ k then  // z cannot be RNN of q
           break;
        endif
     endif
  endfor
  forall p in PS do
     if dist(p, z) ≤ dist(p, q) then
        if p.count ≥ k then
           PS ← PS - {p}
        else
           p.count ← p.count + 1
        endif
     endif
  endfor
  if touchedCounter < k then  // retain z in PS
     z.count ← touchedCounter
     PS ← PS ∪ {z}
  endif
end; {procedure UpdatePrunedSet}
```

Figure 77. The Best-First Cell Expansion with Perpendicular Bisector (BFCE-PB) algorithm for RNN-Grid

The BFCE-PB algorithm utilises an additional set *PS* to retain pruned data points. The discarded candidates can be put to better use in two ways: (i) to be retained in the set *PS* for pruning all the newly identified cells that are located inside the *p*'s side of $\perp(p,q)$ $\forall p \in PS$, and (ii) to trim the set *PS* of unwanted members to keep *PS* size small. Each data point in *PS* has the property `count`, initially set to 0, in addition to its coordinates. Note that in the algorithm BFCE-PB, during `ProcessCandidates` instead of discarding a candidate *p* that was pruned, it is sent to the `UpdatePrunedSet` to trim *PS* or to be added into *PS*. The set *PS*, in turn, is used during the cell expansion to prevent cells not in *q*'s half-plane to be added into the processing queue (bisector perpendicular pruning). BFCE-PB also uses MinMaxDist from [RoKV95].

Since the RNN-Grid algorithms aim to find the required NNs as candidates (Figure 72) for evaluating RNN, it is not entirely obvious why *PS* is maintained according to RNN conditions. The main reason is that pruned points are considered to be not a candidate for NN(q), but it could in fact still be the RNN(q), which is the final objective of all the RNN-Grid algorithms. However, if a incoming point z is not accepted into *PS*, it means that z is definitely a true negative. For points that are already in *PS*, they too can be discarded if enough incoming points (either accepted into *PS* or not) are seen so that they no longer can be a true positive result. The condition for this to occur is $p \in PS, p.count \geq k$.

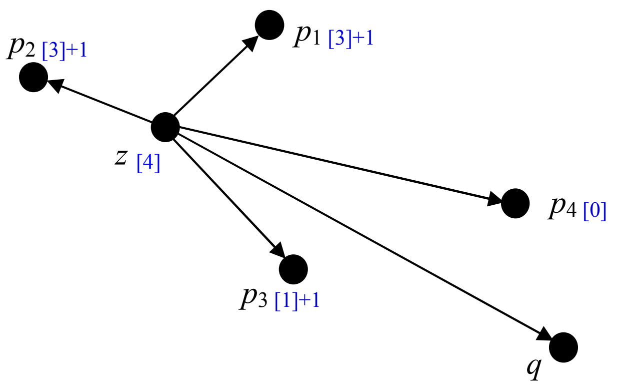


Figure 78. Updating the pruned set *PS* with an incoming point z. The number in square brackets is the counter. The +1 indicates that the counter will be incremented by 1

Figure 78 shows an example of a pruned set *PS* with 4 items, $\{p_1, p_2, p_3, p_4\}$ and their counter value is 3, 3, 1 and 0 respectively. When point z is incoming, the first step is to find out z's counter. The variable `touchedCounter` serves this purpose. The moment `touchedCounter` $\geq k$, it signals that z cannot be a RNN of q and therefore will not be added into *PS*. In the example, if $k < 5$, z would be disqualified (q is the 5NN of z) but if $k \geq 5$, z would be added into *PS*. Next, the counters of existing points $p_i \in PS$ are incremented by 1 regardless of whether z is accepted or rejected, because z is a real point in the dataset. However, the increment for p_i's counter only takes place when $d(p_i, z) \leq d(p_i, q)$. If, after incrementing, p_i's counter value $\geq k$, p_i is removed from PS as it can no longer be a true RNN of q. In the example again, say $k = 4$, p_1 and p_2 will be removed from PS after the increment exercise.

6.2.4 BFCE with Constrained Region (BFCE-CR) Algorithm

The concept of constrained region [StAE00] proposed that three straight lines, one of which is parallel to the x-axis, intersecting at q divide the space around q into six regions of 60° each (assuming L_2 metric). In each region, there can only be exactly 1 RNN or none at all in the case of $k = 1$. This concept was proven in [StAE00] and to answer R*k*NN for any k, we extend the work in that paper and further generalise this concept to any k using the following lemma.

Lemma 3. For any k, if we retrieve exactly k candidate points closest to q from each region, the $6k$ points will be sufficient to answer R*k*NN.

Proof. Let 3 contiguous regions around q be r_1, r_2, r_3 and let there be infinitely many data points. Let the points retrieved from a region always start with the closest points from q (recall that we are using cell expansion which discovers points in this order). Suppose we retrieve k-1 points from q in r_2 and both r_1 and r_3 contain no points. It is easy to see that the k-th point in r_2 could be a valid RNN of q and we had missed it. Suppose we retrieve k+n points from r_2 ($n \geq 1$) and both r_1 and r_3 contain no points, since there are at least k points before the (k+1)th point ($k+n \geq k+1$), the extra n points can never be a RNN of q. Now, let all regions contain some points and we retrieve k points from r_2. Let u be the k-th point in r_2 and point $p \in \{r_1, r_3\}$ located such that either $d(u,$

p) ≤ *d*(*u*, *q*) or *d*(*u*, *p*) > *d*(*u*, *q*) is true. If the first condition is true, then *u* is not a valid RNN of *q* (since now there are *k*-1 points plus point *p*, for a total of *k* points, closer to *u* than *q*) but this fact does not affect our final query answer. If the second condition is true, the fact that *u* is returned as a candidate of the RNN query shows that we are correct. Hence, this proof shows that it is sufficient to retrieve exactly *k* points closest to *q* from each region to answer R*k*NN. ■

The key idea behind BFCE-CR is twofold: (i) retrieve up to *k* points per each of the six regions for a total of 6*k* points, and (ii) for the special case where *k* = 1, also prune points that falls within 60° of a candidate that has already been discovered (constrained region pruning). Unlike the BFW, BFCE and BFCE-PB algorithms, this algorithm is the only RNN-Grid variant that does not follow the general RNN-Grid paradigm as shown in Figure 72. The reason is because BFCE-CR requires the use of six vectors to store up to *k* points from six regions. Figure 79 shows the algorithm in detail.

```
RNN-Grid-BFCECR(q, k, R)
// Input:  Query point q, the k-th RNN
// Output: R - RkNN of q
begin
  initialize regions[1..6]  // vector of six vectors
  filter(q, k, regions[])
  refinement(q, k, regions[], R)  // to discard disqualified cand
end; {procedure RNN-Grid-BFCECR}

filter(q, k, regions[])
begin
  currMinDist[1..6] ← ∞
  initialize bit vector horizon[0..359]  // set all bits to 0

  Q ← getCell(q.x, q.y)  // returns cell where q is located
  while not Q.isEmpty do
     cell ← dequeue(Q)
     cand ← getBucket(cell)  // get all points from cell
     if |R| ≥ k and MinDist(q, cell) > currMinDist then
        continue;  // proceed to next cell
     endif
     ProcessCandidates(q, k, cand, horizon, regions[], currMinDist[])
```

```
      for i in -1 to 1 do
         for j in -1 to 1 do
            if i=0 and j=0 then continue;  // skip middle cell
            cell ← getCell(cell.x+j, cell.y+i)
            if not cell exist then
               continue;  // cell might be at grid boundary
            endif
            for r in 1 to 6 do
               if cell ∊ r and dist(q, cell) > currMinDist[r] then
                  if cell ∉ Q then
                     insert cell into Q sorted by $c_i$∈Q|dist(q, $c_i$)
                  endif
               endif
            endfor
         endfor
      endfor
   endwhile
end; {procedure filter}

ProcessCandidates(q, k, cand, horizon, regions[], currMinDist[])
begin
   forall p in cand do
      if k = 1 then  // check bitvector first
         if horizon[∠xqp] bit = 1 then
            continue  // skip point
         else
            set horizon[∠xqp ± 60] bit ← 1  // wrap around 0-360
         endif
      endif
      r ← getRegion(p)  // each region is 60 degrees, anti-clockwise
                        // starting from line parallel to x-axis
      if |region[r]| < k or dist(q, p) < currMinDist[r] then
         insert p into region[r] sorted by $c_i$∈region[r]|dist(q, $c_i$)
         currMinDist[r] ← max{dist(q, p'∊region[r])}
      endif
   endfor
   if |region[r]| > k then
      truncate region[r] at position k+1;
               // keep the first k results per region
   endif
end; {procedure ProcessCandidates}

refinement(q, k, regions[], R)
begin
   forall r in regions do
      currRank ← 0  // the number of pts already known to be nearer
      forall p in r do
         currRank ← currRank + 1
         distCurrItemFromQ ← dist(q, p) // dist(q, p) is increasing
         numNearer ← currRank - 1
         done ← ProcessRegion(p, r, numNearer, distCurrItemFromQ, k)
                        // process points in the current region
         if done then continue endif
         done ← ProcessRegion(p, r-1, numNearer, distCurrItemFromQ, k)
                        // process points in the left adjacent region
         if done then continue endif
         done ← ProcessRegion(p, r+1, numNearer, distCurrItemFromQ, k)
                        // process points in the right adjacent region
         if done then continue endif
         R ← R ∪ {p}
      endfor
   endfor
end; {procedure refinement}

ProcessRegion(p, r, numNearer, distCurrItemFromQ, k)
begin
   forall u in r do  // points are already in ascending order from q
      if dist(u, p) > distCurrItemFromQ then
         numNearer ← numNearer + 1
      else
         return true
      endif
      if numNearer > k then
         return true
      endif
   endfor
   return false
end; {procedure ProcessRegion}
```

Figure 79. The Best-First Cell Expansion with Constrained Regions (BFCE-CR) algorithm for RNN-Grid

BFCE-CR makes use of six vectors called `regions` to store up to *k* points nearest to *q* discovered from each region in the coarse filtering stage. For the special case where *k* = 1, a candidate point *p* is checked against the `horizon` bit vector first. If *p* falls in an area with bit 1, *p* is eliminated immediately as it cannot be a valid result. Otherwise, *p* is inserted into its correct region, and `horizon` is updated to mark the 60° space to the left and right of *p*. Note that the marking of the bit vector cuts across regions and wraps around the beginning and end of the bit vector. It might overlap into areas with bit already set to 1. In this case, the area of 1-bits will be simply enlarged. Regions are marked 1 to 6 and have a corresponding `currMinDist` of size six also, to record the `currMinDist` for each individual region. $\angle xqp$ refers to the angle from the x-axis in an anti-clockwise fashion. Figure 80 illustrates constrained regions and angles.

The refinement stage aims to eliminate candidate points by counting if there are at least *k* points that are nearer to a candidate point. Let *p* be the *k*-th candidate point in a region *r*. Within the same region *r*, BFCE guarantees that there are at most *k*-1 points between *p* and *q* (from *p* towards *q*). However, on the other direction (away from *q*), there might exist points nearer to *p* than *p* is to *q*. Hence, when processing all candidates *p* in *r*, three variables are used in the processing of *r* and *r*'s two adjacent regions (written as *r*-1 and *r*+1): (i) `currRank` is used to record the number of points between *p* and *q* within *r* only (i.e. *p* being the *k*-th point from *q*), (ii) `numNearer` counts the number of candidates between *p* and *q* as discovered in *r*, *r*-1 and *r*+1, (iii) `distCurrItemFromQ` is a convenience variable assigned as *d*(*p*, *q*) so that it is not re-computed for all the regions. Note that `numNearer` starts with the value `currRank`-1 as through BFCE we know there already exists *k*-1 points if *p*'s ranking is *k*. When processing a region, we simply increment `numNearer` the moment we discover a point *u* nearer to *p* than *q*. As soon as `numNearer` exceeds *k* or *d*(*u*, *p*) exceeds `distCurrItemFromQ`, processing is terminated for the current region and the next point in *r* is processed immediately.

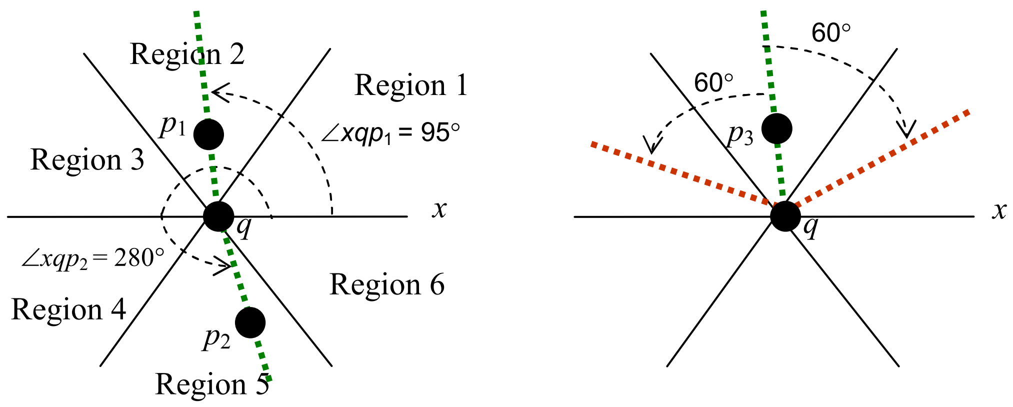


Figure 80. Regions as divided in the constrained region concept. The angle for a candidate point is calculated anti-clockwise from the line parallel to the x-axis. If a candidate point p_3 is discovered and it does not fall within 60° of previously discovered points, all bits within 60° of $\angle xqp_3$ is marked and they cuts across regions

6.3 Experiments and Results

In our experiments, we first set out to compare the various RNN-Grid algorithms. The ERkNN (estimated) approach was chosen to pit against the RNN-Grid algorithms. ERkNN is a fast algorithm utilising statistical estimators to return the candidate set while preserving high recall values, and it outperforms the boolean range query approach (also estimated) of [SiFT03] by a large margin in terms of accuracy and speed.

6.3.1 Experiment Settings

Three measures were used to compare the performance of our RNN algorithms, namely the number of I/Os (disk block accesses), the number of distance computations (CPU cost) and the query time. These measures are consistent with other RNN algorithms in the literature, making performance comparisons possible. The number of I/Os denote the number disk accesses required when answering a R*k*NN query. The number of distance computations (for short, written as #distcomp in the remainder of this thesis) is another accurate measurement, as any algorithm designed to solve RNN is comprised of a distance function as the basic unit to compute the similarity of two data points. A good algorithm tends to perform the optimal #distcomp by filtering only the right candidates from the dataset. Lastly, the query time is measured to compare the overall performance of a R*k*NN algorithm.

Table 20. A pre-computed table of true results for random datasets used to evaluate the quality of estimated RNN query results. The values are computed using the slow naïve method

R*k*NN	Avg |result|	R*k*NN	Avg |result|	R*k*NN	Avg |result|	R*k*NN	Avg |result|	R*k*NN	Avg |result|
20000		50000		100000		200000		400000	
1	1.012	1	0.982	1	1.030	1	1.030	1	0.950
2	1.992	2	2.016	2	1.998	2	2.024	2	1.930
4	3.974	4	4.020	4	3.894	4	4.002	4	4.002
8	8.050	8	7.996	8	7.938	8	7.942	8	7.998
16	15.616	16	15.914	16	15.940	16	15.930	16	15.926
32	31.796	32	31.478	32	31.744	32	31.732	32	31.644

For our experiments, we generated random datasets of size 20K, 50K, 100K, 200K and 400K in uniform distribution. A set of 500 randomly generated query points is used throughout all experiments. Using the very slow naïve O(n^2) method, the R*k*NN is performed 500 times (each time a different query point from the query dataset) with its average taken, for k = 1, 2, 4, 8, 16 and 32. In the RNN-Grid, which is an estimated approach, the results size is averaged over 500 R*k*NN queries (using query dataset) and compared to the pre-computed results. For example, the naïve method takes ~3.5 hours to answer R*k*NN for 400K 2-d points with k = 32.

Table 20 lists all the values of the true results derived from the naïve method. By doing so, we are able to evaluate the quality of our RNN-Grid algorithms and find out how close they are to the true value. The values in the table are upper bounds as typically estimated algorithms miss some true results. The table of true results is also used to ensure that the RNN-C tree returns correct results, which effectively means using the naïve method to double-check its results.

The confidence value for RNN-Grid algorithms was set at 0.995. For high-dimensional datasets, we used a set of 40,700 8-d feature vectors [Gold99], generated from images downloaded from NASA. The feature vectors are normalised to a range of [0, 1]. The ERkNN algorithm was run with its best parameters using the local *k*NN-distance estimator with global adjustments as suggested in [XiHL05]. The TPL and TYM algorithms were run without any modifications.

The experiments were run on a Pentium IV 2.4 GHz Linux machine, with 4 GB RAM. Implementations for RNN-Grid and RNN-C tree were done in C++ and compiled with gcc version 4.1.2. The disk page size is 4096 bytes on the same machine. The I/Os implementation is taken with permission from [TaPL04], thus giving a level platform for fair comparison.

Recall that ERkNN is an estimated algorithm, so we compare RNN-Grid against ERkNN. It uses a local *k*NN-distance (density function) estimator around query point q to estimate the number of candidates required. This approach to guess the number of suitable candidates is similar to RNN-Grid. Therefore, we compare RNN-Grid against ERkNN. In some tables, the results for exact algorithms (TPL or TYM) are also included, for reference. In the last subsection, we show the performance comparison in all three measures.

6.3.2 BFW vs BFCE

The two different approaches for finding NN(q) in the RNN-Grid are compared. This set of experiments is run to decide the better algorithm of the two to serve as the basis for further enhancement. Table 21 shows the results for BFW and BFCE for R*k*NN queries with different values of k. In terms of I/Os, BFCE consistently makes more disk access than BFW. This is because BFW always accesses adjacent cells when processing each wave. So, it is able to take advantage of locality of reference. In contrast, BFCE jumps around the edge of the expanding cells because it processes the cells in a queue of cells sorted in ascending order of MinDist(q, c_i).

BFW returns more candidates during coarse filtering as a result of the wave requirement. When a wave $w > 0$ has started, all $8w$ cells in that wave have to be fully processed. The extra candidates require extra distcomp although they do not contribute to the final results. BFCE uses fewer distcomp as it compares the minimum cells to locate the required k NNs. When k is large, BFCE actually does more distcomp than BFW. This is due to the fact that BFCE sees more cells than BFW. Each expansion potentially adds 5 to 7 cells into the queue and all must be checked (at least one distcomp if the whole cell is pruned, or up to c distcomp – one for each point in the cell, where c is the grid cell size).

Table 21. Performance of BFW and BFCE in dataset of 20K with cell size 64 and disk page 4K

	Avg # I/O		Avg # distcomp		Avg query time (s)	
k	BFW	BFCE	BFW	BFCE	BFW	BFCE
1	3.252	3.252	1020.24	916.612	0.00060	0.00046
2	3.578	3.578	1197.09	1079.65	0.00066	0.00060
4	4.030	4.036	1806.84	1639.68	0.00110	0.00094
8	5.238	5.268	3142.69	2887.74	0.00214	0.00194
16	7.434	7.696	6197.31	5931.95	0.00494	0.00484
32	11.528	12.650	16104.70	16994.70	0.01594	0.01724

BFCE alone is not much better off than BFW. But since BFCE outperforms BFW, albeit not very significantly, we had decided to use BFCE as the base algorithm and enhance it with other known RNN techniques (i.e. perpendicular bisector line and constrained regions) for further performance improvements.

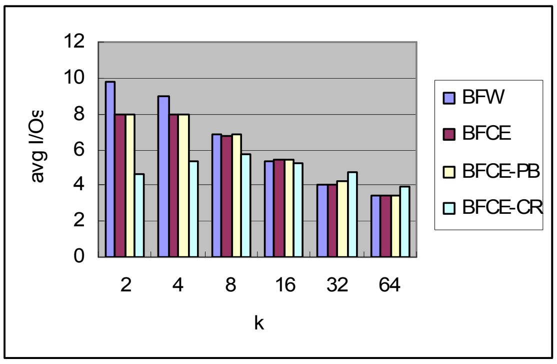
6.3.3 Effect of Grid Cell Size

The grid cell size is a parameter of the grid file. As a grid cell is implemented as a bucket, it is also called bucket size. It is the maximum size of the bucket for any grid file cell. This is where the grid file is different from the fixed grid. In the latter, we typically control the grid by specifying the number of partitions. For a randomly distributed dataset of *n* *d*-dimensional data points, it is logical to divide into $\sqrt[d]{n}$ partitions in the hope that each cell will roughly contain the same number of data points. However, in the grid file, its partitioning algorithm partitions the grid based on data points already in the grid. We typically specify the maximum size that a grid cell can store data points.

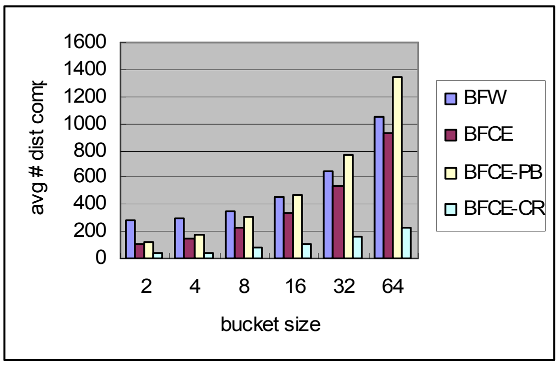
Table 22 shows the performance of RNN-Grid algorithms when different grid cell size is used. In general, the average number of I/Os is expected to decrease when cell size increases. This is because more data points can fit into a cell, so fewer cells need to be accessed. We can see this trend in all algorithms except BFCE-CR. This is due to the fact that BFCE-CR is the only algorithm not following the RNN-Grid paradigm, i.e. not using the probability values table to determine the required number of NNs. As evident in this experiment, apparently BFCE-CR became less efficient than the other three algorithms when $16 \leq k \leq 32$, which means it retrieved more candidates than the others (i.e. the O(6*k*) candidates exceeded the number of candidates required from the probability values table). As the bucket size grows larger ($k \geq 32$), BFW and BFCE's performance almost equals, with the former leading by a slight margin in the average number of I/Os.

Table 22. Effect of grid cell size with 100K dataset, disk page 4K and *k*=1

Bucket size	BFW		BFCE		BFCE-PB		BFCE-CR	
	Avg # I/O	Avg #distcomp	Avg # I/O	Avg #distcomp	Avg # I/O	Avg #distcomp	Avg # I/O	Avg #distcomp
2	9.820	275.884	7.974	111.982	7.974	123.904	4.670	33.810
4	8.996	301.482	7.956	144.636	7.964	180.204	5.318	44.704
8	6.808	356.064	6.806	224.846	6.818	304.114	5.760	74.114
16	5.296	450.654	5.446	332.694	5.478	466.980	5.220	105.998
32	4.010	647.802	4.030	535.808	4.200	761.772	4.728	157.146
64	3.404	1044.000	3.408	933.494	3.464	1348.040	3.948	226.640



(a) avg # I/O vs bucket size



(b) avg #distcomp vs bucket size

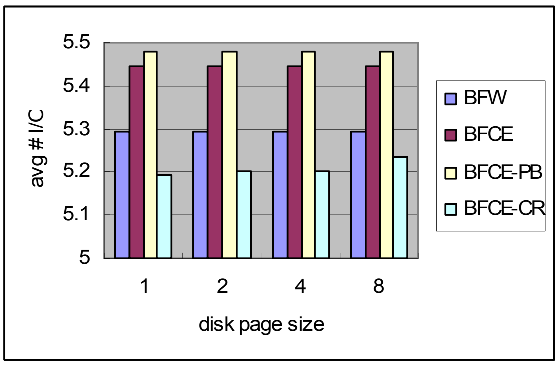
Figure 81. Effect of grid cell size with 100K dataset, disk page 4K and *k*=1

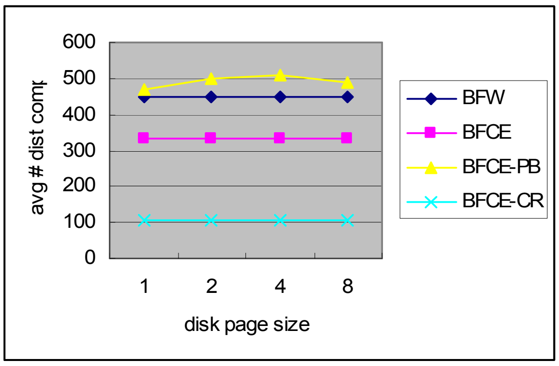
As for the #distcomp, shown in Figure 81(b), when bucket size increases, generally the #distcomp also increases. Although in all our algorithms we have `CurrMinDist` to help us prune off cells before and after they enter the processing queue, for those cells that are not pruned, the number of candidates to check increased, leading to an overall increasing #distcomp. BFW incurs the most #distcomp for small *k* (< 16) due to the waves that it needs to process, whereas BFCE-PB incurs the most #distcomp for large *k* (> 16) due to the extra processing in maintaining the pruned set. Had we not modified the pruned set to drop some candidates, BFCE-PB would be doing the most #distcomp for any *k*. This result tallies with the TPL algorithm (for exact RNN results) which is known to be distcomp intensive. BFCE-CR is the clear winner with a speed-up of 3.31 to 4.12 times over its closest rival, BFCE.

6.3.4 Effect of Disk Page Size

This section details the effect of disk page size for the RNN-Grid. Figure 82 shows that the disk page size seems to have little effect on all four algorithms, except that the BFCE-CR sees an increase in the average number of I/Os. The grid file structure guarantees at most two I/Os for any bucket retrieval; one I/O each to access the partition index and the actual bucket. The grid file actually requires more disk accesses during construction than query. During construction, when a bucket is full, the grid file will try to split the bucket in half in one of the dimensions, selecting the dimension that gives the least data points movement (re-distributing overflowing points into other buckets).

During query, both the BFW and BFCE algorithms discover data points outwards toward the edges of the plane, with the cell where the query point hits as the centre. Since they depend on the probability values table, they need to return more candidates than BFCE-CR. BFCE-CR only requires 6*k* candidates; hence it will at most incur 6 disk accesses. For the #distcomp, BFCE-PB reacts to the increasing disk page size likely because of the pruned set that it maintains. The overall performance in Section 6.3.7 details more.





(a) avg # I/Os vs disk page size (b) avg # dist comp vs disk page size

Figure 82. Effect of disk page size with 100K dataset, bucket size 16K and *k*=1

6.3.5 Precision and Recall Analysis

Precision and recall are two measures of a search algorithm's sensitivity in an approximate query. In the RNN-Grid algorithms, due to the fact that we estimate the top k_1 NNs to contain Rk_2NN of q, the results retrieved from such a query is bound to contain a small number of false positives (FP), in addition to the true RNN results (true positives, TP). FP refers to the number of data points included in the RNN results, but should not. Figure 83 depicts the RNN result set from a RNN-Grid query. Oftentimes, in the larger picture, there may be some correct data points not included in the RNN-Grid query result, as they may lie beyond the top k_1 NN of q, yet are indeed the RNNs of q. These data points are wrongly regarded as non-result, therefore labelled as false negatives (FN).

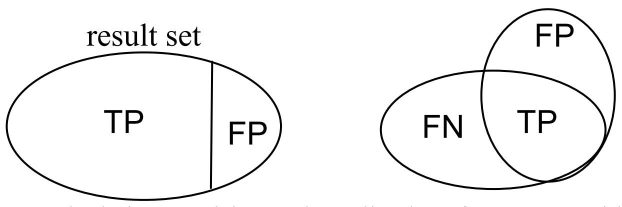


Figure 83. Calculating precision and recall values from true positives (TP), false negatives (FN) and false positives (FP).

Using the true results known *a priori* that are also used to calculate the table of true results (Table 20), the FP and FN values can be determined from any result set of any of the RNN-Grid algorithms. The precision and recall values are computed as follows

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

A high precision value indicates that there are very few false RNN results that were included by the algorithm, which is good. In fact, a precision value of 1.0

means that every result retrieved was correct but did not tell whether all correct results were retrieved. A high recall value means that there are very few true RNN results that were missed by the algorithm, which is good. In fact, a recall value of 1.0 means that all correct results were retrieved but did not tell how many incorrect results were also retrieved.

Table 23.The precision and recall values of the two best RNN-Grid algorithms compared to the ERkNN algorithm.

k	Precision			Recall		
	BFCE-PB	BFCE-CR	ERkNN	BFCE-PB	BFCE-CR	ERkNN
1	0.635	0.535	0.823	0.986	0.998	0.994
2	0.511	0.383	0.827	0.970	0.999	0.992
4	0.465	0.378	0.818	0.948	0.994	0.990
8	0.455	0.374	0.834	0.953	0.996	0.983
16	0.445	0.343	0.842	0.958	0.988	0.982
32	0.468	0.310	0.810	0.987	0.990	0.980

In our experiments, we used the 100K dataset, a bucket size of 64, disk page size 4K, and confidence values of 0.99 for BFW, BFCE and BFCE-PB algorithms. The experiments were conducted for various R*k*NN queries (varying *k*). First of all, we note that the ERkNN algorithm does produce high recall values for our experimental dataset. The recall decreases as *k* increases, but it is still above 0.98 for $k \leq 32$, which is considerably high.

In all our RNN-Grid algorithms, the recall values are very high, above 0.94. This is because our statistical analysis method works well for the random data distribution. Comparing BFW and BFCE, the BFCE has a slightly higher recall than BFW. The recall values of BFCE and BFCE-PB are similar because the latter actually added an extra method of pruning on top of the former. This merely affects the order in which candidate points are found and inserted into the result set, but the final candidates set is still the same. In fact, in the next section, we showed that BFCE-PB does not add value to BFCE. In fact, the recall for BFCE is not as good as ERkNN's, which means the statistical analysis alone may not be sufficient to compete with the local *k*NN-distance estimator in the ERkNN algorithm, except when *k* is very large ($k \geq 32$). The local *k*NN-distance estimator is merely estimating the number of candidates from a small sample of data points around the query point *q* and works only for random data distribution. In the case when *k* is small, it can predict the number of candidates to retrieve from its small samples pool. When *k* is large, the prediction might be off. BFCE and BFCE-PB outperform ERkNN for large *k* as the statistical analysis is done on a much larger pool of data points, therefore increasing the effectiveness of the method.

On the other hand, BFCE-CR consistently produces a higher recall rate than ERkNN for all *k*. The simple reason is that BFCE-CR returns larger number of candidates (6*k*) than the ERkNN algorithm. So the RNN results are almost complete in the BFCE-CR algorithm, at the expense of additional running time spent in the refinement step to verify the candidates.

The precision values for BFCE-PB and BFCE-CR, however, were consistently lower than ERkNN. This is because BFCE-CR has expanded and included too many false positives. As each cell in the grid file is most likely not a square, many results may have been included during cell expansion. This can be improved by reducing the bucket size of the grid file (hence effectively reducing density) and employing distance calculations in the refinement step to filter the data points in each cell. We note that BFCE-PB has better precision than BFCE-CR as the method keeps track of pruned data points to act as bisector pruning for future data points. This indirectly cuts away a lot of false positives data points that share a cell with true positives during expansion.

6.3.6 High Dimensional Data

Although the grid file is a dynamic structure that is extensible to higher dimensions, it is known to degrade in performance for indexing data in very high dimensions (say, $d > 100$). Unlike the fixed grid which can be easily modified to any dimensions as long as the size of the hyperplane is known, the grid file needs to maintain an array of partition indices for each partition. This means that high dimensions actually results in costlier maintenance of partition index arrays, not to mention the algorithms for repartitioning of high-dimensional data points becomes inefficient, due to the 2^{d-1} possible ways to split the grid, when a bucket is full.

Similarly, the performance of the TPL algorithm also suffers from the curse of dimensionality, as acknowledged in the original paper. The performance of TPL degrades due to the underlying R-tree data structure that it uses. The bisector perpendicular line used for pruning has become a hyperplane ($d > 2$), and its coarse filtering encounters many more potential candidates which leads to a much costlier refinement step, as all points and MBRs in the refinement set S_{rfn} are used for pruning.

With this understanding, experiments were conducted to look at the performance of R*k*NN queries in an 8-d real-life dataset, a set of 40700 feature vectors of NASA images. The 2-d dataset are randomly generated. As the BFCE-CR is the best of all RNN-Grid algorithms, it is selected for comparison in this experiment. A bucket size of 64 and disk page size 4K for BFCE-CR were used. For the TPL algorithm, the disk page size used is also 4K. To be fair, we note that in this comparison, BFCE-CR is an estimated approach while TPL is an exact approach to answer the R*k*NN query. Therefore, BFCE-

CR is expected to show better performance as TPL needs to do much more work to ensure the results returned are accurate.

Table 24. Comparison of R*k*NN queries in 2-d and 8-d datasets. The number of distance computations of BFCE-CR and TPL are shown

	BFCE-CR	TPL	BFCE-CR	TPL
k	(2-d data)		(8-d data)	
1	226	1669	574	86653
2	266	2676	753	97149
4	355	4428	1109	167905
8	587	8420	1970	431335
16	1297	19969	3634	749178
32	3427	56477	6936	1342755

TPL performs 12.47 times more distcomp than BFCE-CR ($k = 4$), but it incurs 2-3 orders of magnitude more #distcomp than BFCE-CR in the 8-d dataset. In fact, the experiment results in Table 24 suggest that TPL has #distcomp increase of between 1.12 and 2.57 times, as k increases. The R-tree becomes less efficient with more overlaps in high dimensions [ThSe96] as it is impossible to construct an R-tree with only 10% overlap. Owing to this, TPL spends up to 98% of query cost spent on the filtering step.

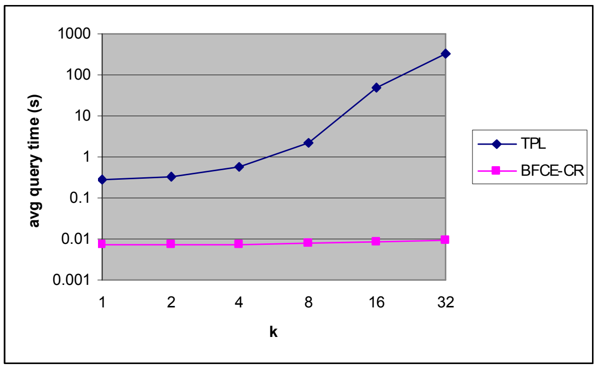


Figure 84. Comparison of R*k*NN queries in 8-d data. The average query time for BFCE-CR and TPL are shown

Next, we look at the query time used by both algorithms in Figure 84. At 8-d, the TPL algorithm is at least one order of magnitude slower than BFCE-CR and grows to 4 orders of magnitude, and still growing, for large *k*. Query costs generally explodes for both BFCE-CR and TPL in tandem with higher dimensions; as data points move to hyperplanes, the #distcomp in operations like "find enclosure" (the R-tree's MBR in TPL) increases tremendously.

6.3.7 Performance Comparisons

Finally, we compare all the RNN-Grid algorithms (BFW, BFCE, BFCE-PB and BFCE-CR) to ERkNN, in all three measures. The corresponding values for the same measures for TPL and TYM algorithms are also presented as a reference. The experiments are performed using the 100K dataset, a bucket size of 64, disk page size 4K, and confidence values of 0.99 for BFW, BFCE and BFCE-PB algorithms.

In Table 25, the number of disk accesses is compared. Note that the BFCE-CR started off with more disk accesses than BFW, BFCE and BFCE-PB. As *k* becomes large, it is evident that BFCE-CR requires less I/Os as it only need to retrieve O(6*k*) candidates, compared to up to 15*k* candidates needed for answering a R*k*NN query for large *k*. On the average, most RNN-Grid algorithms need fewer I/Os than ERkNN, except for $k \geq 32$ when BFW, BFCE and BFCE-PB begin to lose out to ERkNN's method of estimating the required NN candidates. This means that they sought more candidates than ERkNN at that point and hence requires more I/Os. However, this is actually in response to our stricter confidence level of 0.995 to discover all R*k*NN results; but then this results in a higher recall that outperforms the ERkNN's recall. This was presented in Table 23, when $k = 32$ the recalls of BFW, BFCE, BFCE-PB and BFCE-CR are higher than that of ERkNN's. We also showed that the grid file structure is as good as the R-tree (for ERkNN) for answering RNN queries. For reference, the TPL and TYM algorithms need much more disk accesses as they need to ensure exact R*k*NN results.

Table 25. Performance comparison (number of I/Os) of all RNN-Grid algorithms with ERkNN, TPL and TYM

k	BFW	BFCE	BFCE-PB	BFCE-CR	ERkNN	TPL	TYM
1	3.404	3.408	3.464	3.948	5.696	915	524
2	3.618	3.622	3.694	4.186	6.604	991	572
4	4.240	4.244	4.318	4.614	7.150	1103	610
8	5.384	5.436	5.570	5.544	7.948	1300	664
16	7.734	7.928	8.104	7.854	9.440	1584	751
32	11.966	13.06	13.358	11.160	11.364	2105	848

The performance in terms of #distcomp is shown in Table 26. The efficiency of the RNN-Grid algorithms are always BFCE-PB > BFW > BFCE > BFCE-CR. This is easily explained by looking at the BFCE-PB algorithm. It incurs the most #distcomp because it has to maintain a pruned set *PS*, at the cost of O(2|*PS*|) distcomp. The pruned set was originally designed to help in the coarse filtering by quickly pruning off large regions of space where any data point found in these regions is guaranteed as a non-result. However, this method introduces unavoidable distcomp necessary to maintain *PS* as well as using *PS* for coarse filtering.

Although BFW is a simple idea, it actually performs better than the BFCE-PB. It is about 1.3 to 1.7 times more efficient than BFCE-PB. As shown earlier in Section 6.3.2 where we need to decide between BFW and BFCE to extend, the BFCE is consistently faster than BFW except that for *k* = 32. Since a typical RNN query (say, a virtual reality shooting game) focuses on small *k*, the BFCE was chosen. The BFCE-CR is by far the fastest estimated algorithm, has 2.7 to 8.9 times fewer distcomp than ERkNN yet outperforming ERkNN with a higher recall. The BFW and BFCE made fewer #distcomp than ERkNN for $k \geq 4$, and BFCE-PB, $k \geq 8$. This indicates that the ERkNN algorithm is not efficient for large k, even though the query aggregation of ERkNN has helped to reduce up to 75% of distcomp. As expected, the exact algorithms (TPL and TYM) perform more distcomp than estimated algorithms. In particular, the TYM algorithm uses significantly more distcomp even for $k = 1$, because it is an algorithm for the general metric space, and it cannot take advantage of well-known Euclidean geometric properties for pruning.

Table 26. Performance comparison (number of distance computations) of all RNN-Grid algorithms with ERkNN, TPL and TYM

k	BFW	BFCE	BFCE-PB	BFCE-CR	ERkNN	TPL	TYM
1	1044.00	933.494	1348.04	226.640	614.086	1669.196	100529
2	1203.61	1078.00	1612.34	266.968	1017.184	2676.826	101053
4	1859.36	1677.69	2532.71	355.426	2038.812	4428.880	103673
8	3220.01	2955.66	4575.94	587.498	4608.038	8420.356	115201
16	6489.35	6204.03	10335.82	1297.317	11518.299	19969.190	163409
32	16670.00	17446.71	28360.53	3427.315	30177.252	56477.908	360433

Finally, we compare the performance of all the estimated algorithms in terms of query time. The trend is similar to the #distcomp. BFCE-PB takes the longest to run and the BFCE-CR is the fastest among all estimated algorithms, except when $k = 1$, ERkNN is equal. The TYM algorithm runs in the range of seconds, as it was contributed by the high #distcomp cost. The results of TPL and TYM are presented here to provide an idea of how long it takes to arrive at exact results.

Overall, we reiterate that the BFCE is consistently faster than BFW, hence it was chosen to be extended. The BFCE-PB is not a viable extension as it performed worse than BFCE. This shows that the perpendicular bisector pruning is not workable, because although finally we are looking for RNN results, the approach to obtain candidates is by NN. On the contrary, BFCE-CR is a very good improvement on the BFCE, as it not only runs fast, but manage to produce very high recall values (≥ 0.988, from Table 23).

Table 27. Performance comparison (query time in seconds) of all RNN-Grid algorithms against ERkNN, TPL and TYM

k	BFW	BFCE	BFCE-PB	BFCE-CR	ERkNN	TPL	TYM
1	0.00086	0.00064	0.00150	0.00032	0.00032	0.00096	3.93
2	0.00100	0.00076	0.00188	0.00038	0.00039	0.00136	5.35
4	0.00162	0.00124	0.00304	0.00044	0.00065	0.00286	5.41
8	0.00292	0.00244	0.00584	0.00060	0.00232	0.01034	5.52
16	0.00654	0.00592	0.01480	0.00098	0.00346	0.05076	5.86
32	0.01920	0.01994	0.04360	0.00184	0.00771	0.30570	6.21

6.3.8 Dataset Distributions

One lingering concern is the effectiveness and accuracy of RNN-Grid as it depends on a pre-computed table determined from statistical analysis. When an actual dataset is given, its distribution may be different from the data used in the statistical analysis. This section looks into the problem of using a different statistical table that we derived in Section 5.6.1 for crossed-value distributions of data. For the analysis, we studied three types of data distributions: uniform, normal (Gaussian) and real-life data from the TIGER/Line database [TIGER02].

Table 28. The value of k_1 for P(Rk_2NN(q) $\subseteq$ k_1NN(q)) > 0.9 for different dataset distributions

k	Real-life	Normal	Uniform
1	2	3	3
2	4	4	4
4	6	8	6
8	11	15	11
16	20	21	20
32	39	44	42

We selected the probability value of 0.9 for a more realistic comparison between the different dataset distributions. The results show that the values of k_1 is very similar for the three distributions, which suggest that the robustness of the accuracy of RNN-Grid across different data distributions.

6.4 Summary

In this chapter, several ideas based on the grid file for estimating R*k*NN results were explored. As a result, the RNN-Grid algorithms based on the grid file data structure were developed. The RNN-Grid is a very fast alternative for answering the RNN query where full accurate results are not desired, in exchange for speed in query response time. Experiments showed that the RNN-Grid outperforms other estimated RNN approaches such as the ERkNN and SFT. Not only is RNN-Grid faster than ERkNN, it also has better recall in the results it returned.

The best-first cell expansion algorithm, combined with constrained region pruning technique (BFCE-CR), is shown to be a promising approach resulting in fast execution and very high recall. BFCE-CR is almost similar to ERkNN in terms of running time when the dimension *k* is small. For larger *k*, BFCE-CR is much faster than ERkNN, retaining the same high recall value that ERkNN does.

In terms of implementation, we believe RNN-Grid algorithms are easier to implement, and the underlying grid file data structure also has the same advantage of insertion, deletion and update (point query) as does the R-tree that ERkNN was based upon.

Chapter 7 RNN-C Tree: An Exact Approach for RNN Query

To answer the RNN query with certainty is a much more challenging and harder problem. The main challenge is how to process minimal data points, be sure that the RNN results are correct and terminate the query. As the NN and RNN are asymmetrical, we cannot use distance from *q* as the terminating condition. Worse, some techniques described for RNN processing only work on the assumption of a certain distance metric or data dimensionality.

In this section, we propose a novel hierarchical data structure and corresponding query algorithm for answering the RNN query, called RNN-C (C for cluster) tree. We chose to design a RNN algorithm for the general metric distance, which work as long as a distance function is defined between two data points (or objects) that conform to the triangle inequality principle. In general metric distance indexing, our algorithm cannot assume any location information on the data points, therefore pruning techniques that make use of absolute coordinates cannot be used. This makes it all the more challenging, but results in an algorithm that works across all distance metrics.

RNN-C tree has several advantages. It is designed especially for finding *exact* RNN results. It is also simple to understand and implement. Our experimental results show that RNN-C tree outperforms the current state-of-the-art algorithm for metric distance RNN query [TaYM06]. The RNN-C is based on the concept of *k*NN graphs, first introduced in [SeKi02] for pattern recognition research, which proposed that data points be linked to their 1NN, which results in cyclic graphs of disjoint components we call clusters. The topology of the *k*NN graph is deterministic and inherent from the position of data points, regardless of the order in which they are presented (unlike the R-tree which is dependent on presentation order of data points, but not including those built by bulk-loading techniques).

7.1 Preliminaries

The key design concept of the RNN-C tree data structure is to construct a data structure that satisfies the following conditions: (i) be able to answer R*k*NN queries, (ii) be able to index metric distances and make use of them for pruning, (iii) hierarchical so that pruning a node will ensure that that branch of child nodes do not contain valid RNN results, and (iv) easy to implement and understand.

The RNN-C tree is based on the idea of *k*NN graph. We took the idea one step further by regenerating the *k*NN graphs on multiple levels and linked them up to form a hierarchical tree structure. The main reason for doing so is because the *k*NN graph represents a forest of clusters in which each cluster is a minimum spanning tree. This is a direct consequence of each data point linking to its 1NN. All points in a cluster are stored as a node in the RNN-C tree as they are closely related for answering the R*k*NN query.

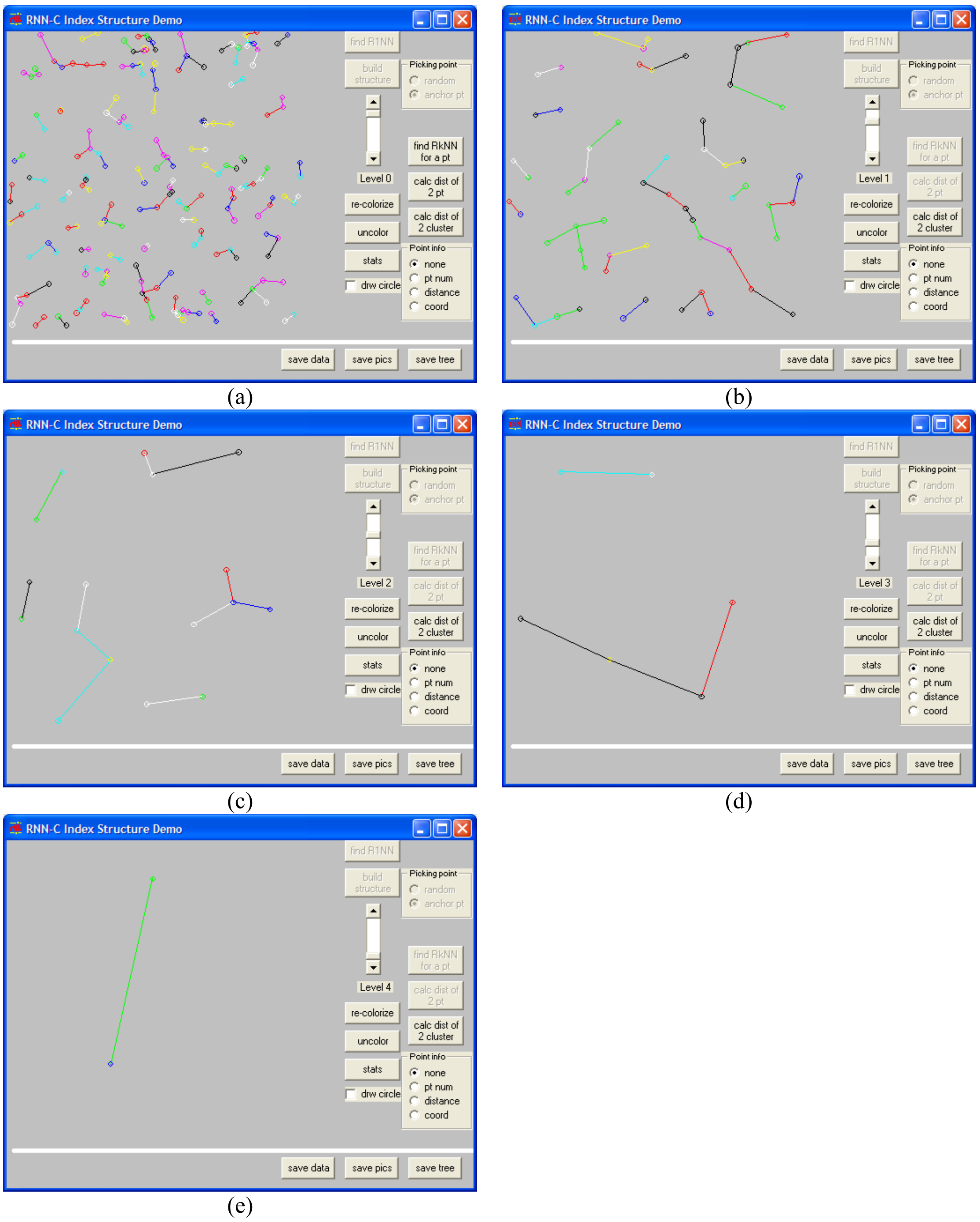


Figure 85. An example of the RNN-C tree hierarchical index data structure of 200 data points. The tree is built from bottom-up. At each level, clusters are formed by the data points' inherent position. One way to build the tree is by selecting a representative point from each cluster to become a data point in the next level

The RNN-C tree is built one level at a time from bottom up. At the bottom level (leaf), the *k*NN graph is computed from the dataset *SDB*, where clusters are formed. This is essentially the same as a *k*NN graph. To construct the next level, the centroid $c_i^{\,j}$ (*j*-th centroid at the *i*-th level) of a cluster $C_i^{\,j}$ is computed and it will become a data point for the construction of the *k*NN graph at the (*i*+1)-th level. Note that $c_i^j \notin C_i^j$, therefore $c_i^j \notin SDB$. c_i^j is merely a representation of C_i^j. The reason a centroid is computed for any C_i^j is due to the fact that the RNN-C tree uses a minimum bounding circle (MBC) to represent C_i^j and the pruning is based on MBC. This process is repeated until the root level where < 3 points is left. Table 29 lists the notations related to the RNN-C tree.

Table 29. Notations used in the RNN-C tree

Notation	Definition
SDB	dataset
q	the query point
k	the number of R*k*NN
C_i	all clusters at level *i* (at leaf level, *i* = 0)
$\lvert C_i \rvert$	number of clusters at level *i*
C_i^j	cluster *j* at level *i*
c_i^j	the centroid for cluster C_i^j
r_i^j	radius for MBC of C_i^j centred at centroid c_i^j
σ_i^j	the population for cluster C_i^j, defined as $\sigma_i^j = \begin{cases} \sum_{m=1}^{\lvert C_{i-1} \rvert} \sigma_{i-1}^m \ \ if\ i > 0 \\ \lvert C_i^j \rvert \ \ if\ i = 0 \end{cases}$
n_i	total number of points at level *i*. $n_0 = \lvert SDB \rvert$. $n_{i+1} = \lvert C_i \rvert$. note that $n_i >> \lvert C_i \rvert$
$\lvert C_i^j \rvert$	the size of C_i^j, excluding centroid. $\sum \lvert C_i^j \rvert = n_0$, for $i = 0$
h	height of RNN-C tree, $1 \leq j \leq h$

7.2 RNN-C Tree Construction

For our RNN-C tree, initially questions were abound and there were three distinct directions to pursue. First, to go with 1NN linkage (at leaf level) and check whether it is even possible, let alone sufficient, to answer a R*k*NN query for any *k*. Second, if the first method is implausible, to explore whether it is possible to expand a RNN-C tree (based on 1NN) dynamically via computation during query execution to answer a R*k*NN query for any *k*. Third, to exploit a RNN-C tree based on *k*NN (*k* to be determined) that could answer a R*k*NN query for any *k*. Eventually, our research had proven that the first direction was plausible and adheres beautifully to our aim of an algorithm that is simple to implement.

Figure 86 lists the algorithm for constructing a RNN-C tree. We highlight three important areas in the algorithm, namely (i) an algorithm needed to find the 1NN of a data point, (ii) computing the radius $r_i^{\,j}$ for a cluster $C_i^{\,j}$, and (iii) computing the population of a cluster. To tackle the first problem, we adopted the fast branch-and-bound NN algorithm of [RoKV95], but any implementation of exact NN algorithms can be used. To compute the radius, we redefined a data point to include a link (index) to the clusters that it represents as a centroid (in addition to its coordinates), for all non-leaf data points. The population of a cluster is the number of data points contained in the cluster *including* all clusters at the lower levels. At level 0, the population $\sigma_i^{\,j}$ of a cluster $C_i^{\,j}$ is simply the size of the cluster $|C_i^{\,j}|$. At intermediate levels ($i > 0$), the population $\sigma_i^{\,j}$ of a cluster $C_i^{\,j}$ is $\sigma_i^{\,j} = \sum|C_{i\text{-}1}^{\,j'}|\ \forall p \in C_i^{\,j}$ where $p \Leftrightarrow c_{i\text{-}1}^{\,j'}$. The purpose of $\sigma_i^{\,j}$ is for pruning in R*k*NN query, where $k > 1$.

In the algorithm, `whichCluster` is a straightforward implementation of the member function of a set, so it is not shown. `calcCentroid` is also not presented because it is also a straightforward computation of the centre of a cluster across all data points in the cluster, averaged for each dimension.

```
RNN-C-tree-build(SDB, T)
// Input:  spatial database SDB
// Output: RNN-C tree T
begin
  h ← 0  // height of final RNN-C tree
  n ← |SDB|  // number of points at this level
  repeat
     h ← h + 1
     RNN-C-tree-build-level(SDB, C[])
     SDB ← ∅
     for i = 1 to c do
        p.x ← C[i].centroid.x
        p.y ← C[i].centroid.y
        p.link ← i
        SDB ← SDB ∪ {p} // each point links back to its cluster
     endfor
     n ← |SDB|
     T[h] ← C[]  // building of RNN-C tree level by level
  until n ≤ 3
end; {procedure RNN-C-tree-build}

RNN-C-tree-build-level(SDB, C[])
// Input:  spatial database SDB
// Output: array of c sets of data points C[1..c]
begin
  c ← 0  // number of clusters at this level

  //generate clusters
  forall p in SDB do
     p’ ← NN(p, 1)  // find 1NN(p) from SDB
     if p.inserted = false and p’.inserted = true then
        temp ← whichCluster(p’, C[]) // add to same cluster as p’
        C[temp] ← C[temp] ∪ {p}
        p.inserted ← true
     elseif p.inserted = false and p’.inserted = false then
        c ← c + 1  // form a new cluster
        C[c] ← C[c] ∪ {p,p’}
        p.inserted ← true
        p’.inserted ← true
     elseif p.inserted = true and p’.inserted = true then
        temp ← whichCluster(p, C[]) // link both clusters together
        temp2 ← whichCluster(p’, C[])
        C[temp] ← C[temp] ∪ C[temp2]
        delete C[temp2]
        c ← c - 1
     elseif p.inserted = true and p’.inserted = false then
        temp ← whichCluster(p, C[]) // add to same cluster as p
        C[temp] ← C[temp] ∪ {p’}
        p’.inserted ← true
     endif
  endfor

  //compute the centroid and radius for all clusters at this level
  for i = 1 to c do
     C[i].centroid ← calcCentroid(C[i])
     C[i].radius ← calcRadius(C[i], i)
     C[i].population ← calcPopulation(C[i], i)
  endfor
```

```
end; {procedure RNN-C-tree-build-level}

calcRadius(C, i)
// Input:   a cluster C, level where C is
// Ouptput: the radius that covers the cluster C
begin
  radius ← 0  // the radius of the cluster to compute
  forall p in C do
     dist ← dist(p, C.centroid)
     if dist > radius then
        radius ← dist
     endif
  endfor

  // recursively expand the cluster radius to cover clusters below
  if i > 0 then
     forall p in C do
        dist ← calcRadius(C[p.link], i-1)
        if dist > radius then
           radius ← dist
        endif
     endfor
  endif

  return radius;
end; {procedure calcRadius}

calcPopulation(C, i)
// Input:  a cluster C, level where C is
// Output: total population of the cluster, including its children
begin
  if i = 0 then  // leaf level
     total ← |C|
  else  // intermediate level
     total ← 0
     forall p in C do
        total ← total + calcPopulation(C[p.link], i-1)
     endfor
  endif
  return total;
end; {procedure calcPopulation}
```

Figure 86. The RNN-C tree construction algorithm

To build the RNN-C tree data structure, the initial dataset *SDB* is read. Then the algorithm attempts to build a *k*NN graph (with 1NN relation) from the data points, which will results in $|C_0|$ clusters at the leaf level (level 0). During cluster construction, the data points are processed sequentially to find their 1NN. Each data point *p* also has a Boolean value to indicate whether it has been previously inserted into a cluster, or newly discovered. In actual implementation, a bit vector of size $|SDB|$ can be used. Let NN(p) be p'. There are 4 possible outcomes for p and p'. If p is a new data point but p' is not, p will join the cluster of p', and vice versa. If both p and p' are new data points, then a new cluster is born, as neither p nor p' is connected to other clusters. If both p and p' were inserted before (p must have been the NN of some other point, and now p's NN is p' which belongs to another cluster), so both clusters are linked up. At the end of the procedure, each data point in SDB would be inserted into one and only one cluster. Finally, the centroid $c_i^{\,j}$ for each discovered cluster is computed. To compute the radius, for the case where the level is 0 (leaf), $r_i^{\,j}$ is $\max\{d(c_i^{\,j}, p)\}$ $\forall p \in C_i^{\,j}$. For cases of intermediate nodes, $r_i^{\,j}$ is recursively grown to cover the MBC of all clusters in each point.

The centroid is extracted from each cluster and a link (index) is added to the cluster which the centroid represents, to form the new dataset (of smaller size, which is equal to $|C_0|$). The process is repeated until the dataset size $|C_h|$ is ≤ 3. One concern about the RNN-C is whether the construction algorithm will terminate.

Lemma 4 proves that the RNN-C tree will not result in an unbalanced or skewed tree. Furthermore, we had empirically shown in our statistical analysis (Section 5.6.2) that $|C_i|$ reduces to approximately 0.20 to 0.32 of its size at the next level, both for random dataset and real-life GIS dataset. In theory, the reduction is at most 0.5. Figure 87(a) illustrates how the RNN-C tree is constructed on a dataset of 12 data points, $n_0 = |SDB| = 12$.

Firstly, for all $p_i \in SDB$, $1 \leq i \leq 12$, find NN(p_i). The directed edges in the diagram is merely a visual representation of p_i with NN(p_i), in actual implementation, an array of pointers of size n_0 would suffice. Note that there are 4 clusters formed (C_0^1 to C_0^4, and $|C_0| = 4$). Secondly, for C_0^i, $1 \leq i \leq 4$, calculate the centroids of each cluster (c_0^1 to c_0^4, respectively). They are denoted as white dots in the figure. Thirdly, using the centroids for each cluster, determine a radius large enough to cover the cluster, i.e. computing r_0^1 to r_0^4. Finally, compute the population of the cluster. Then we are done for this level. Repeat all the steps above recursively with the 4 centroids assumed as data points on the next level. In Figure 87, horizontal dotted lines connecting from white dots in (a) to black dots in (b) indicate this (similarly, from white dots in (b) to black dots in (c)). The construction algorithm terminates when $n_h \leq 3$, where h is the height of the resulting RNN-C tree. In Figure 87(b), $n_1 = 4$ and $|C_1| = 2$. In Figure 87(c), $n_2 = 2$, $|C_2| = 1$ and the construction algorithm terminates with $h = 2$ since the condition $n_2 \leq 3$ is met.

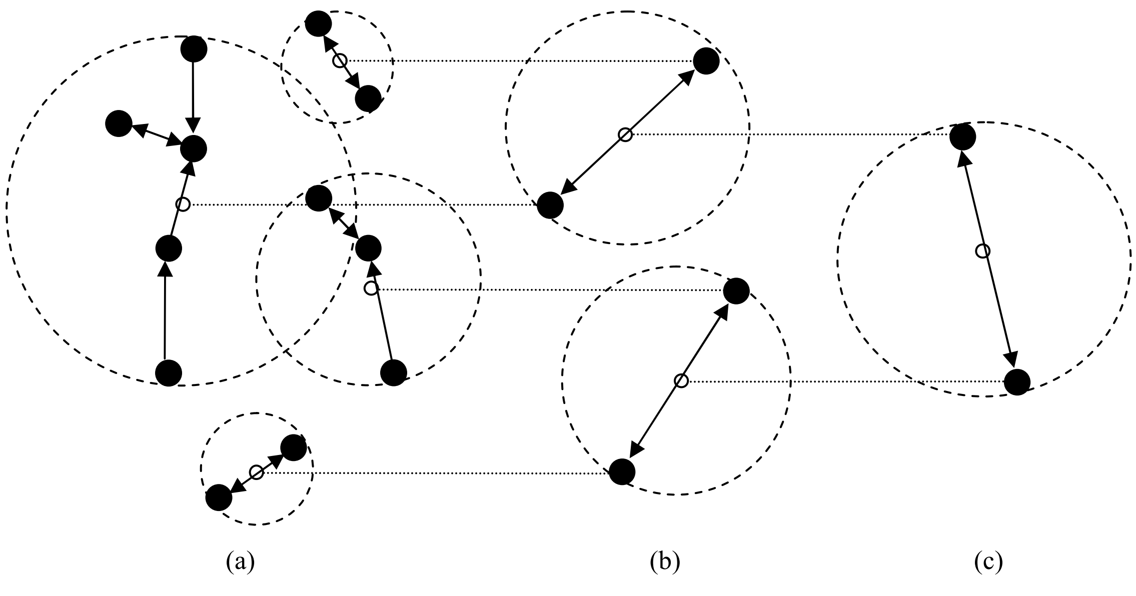


Figure 87. Constructing the RNN-C tree for a dataset of 12 points. Note that $x \rightarrow y$ denotes NN(x) is y. (a) find each point's 1NN and calculate the centroid (white point) for each resulting cluster, (b) the centroid becomes a data point on the next level; repeat the same process as in (a) at this level, (c) stop when 3 or less data points remain

Lemma 4. The algorithm to construct RNN-C tree will always terminate with the finite height h of $O(\log_2 n)$.

Proof. The kNN graph is based on the 1NN relationship of all the points in the dataset to their 1NN. Let $\{p_1, p_2, p_3\} \in SDB$, $n = |SDB|$, NN(p_1) be p_2 and 2NN(p_2) = $\{p_1, p_3\}$. By definition of kNN graph, an edge always connects from p_1 to p_2. In this situation, there are 2 possibilities for p_2, either $d(p_2, p_1) < d(p_2, p_3)$ which means NN(p_2) is p_1, or $d(p_2, p_1) \geq d(p_2, p_3)$ which means NN(p_2) is p_3. For the former, there will be an edge from p_2 to p_1, and assuming all other clusters have the same conditions, there will be $n/2$ clusters, which means $n/2$ points on the upper level. Assuming the same conditions, eventually we will arrive at a RNN-C tree of height $\log_2 n$. ■

7.3 R1NN Queries with RNN-C Tree

Having presented the RNN-C tree construction algorithm, we now discuss and prove two lemmas, presented below, which are used by the query algorithms to traverse the RNN-C tree and prune away points during traversal.

Lemma 5. A cluster C_i^j with centroid c_i^j and radius r_i^j does not have a RNN of any query point q if $d(c_i^j, q) > 2r_i^j$.

Proof. To prune off the whole cluster C_i^j we need to show that $q \notin$ NN(p_i) for all $p_i \in C_i^j$. Since all p_i are enclosed by MBC(c_i^j, r_i^j), $d(c_i^j, p_i)$ is at most r_i^j. In the worst case, $\exists p_j$ where $d(c_i^j, p_j) = r_i^j$. This p_j could become the NN of q if $d(q, p_j) \leq d(p_j, c_i^j)$ (which is r_i^j). Therefore, if $d(q, p_j) > d(p_j, c_i^j)$, q cannot be the NN of p_j. Since $d(q, p_j) > r_i^j$ and $d(c_i^j, p_j) = r_i^j$, then when $d(c_i^j, q) > 2r_i^j$ none of p_i can be the NN of q. ■

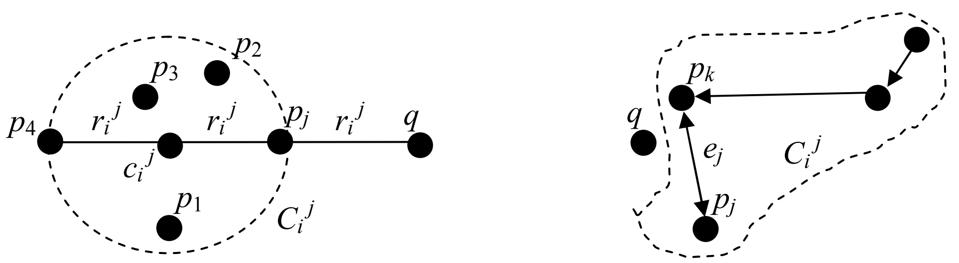


Figure 88. An example illustrating the conditions for Lemma 5 (left) and Lemma 6 (right)

Lemma 6. In a cluster C_i^j where the longest edge is e with length $|e|$, C_i^j does not have a RNN of any query point q if $d(q, (p_i \in C_i^j)) > |e|$.

Proof. To prune off the whole cluster C_i^j in this situation, we need to show that $d(q, p_i) > |e|$ for all $p_i \in C_i^j$. Let us pick two random points p_j, $p_k \in C_i^j$ which is connected by an edge e_j. If $d(q, p_j) \leq d(p_j, \text{NN}(p_j))$ then p_j would be a NN of q. Note that NN(p_j) could be p_k or some points. So when $d(q, p_j) > d(p_j, \text{NN}(p_j))$, it follows that $q \notin \text{NN}(p_j)$. However, it may happen that $d(q, p_j) < d(p_j, p_k)$ because $|e_j| < d(p_k, \text{NN}(p_k))$. This can be avoided if e_j is the longest edge e in C_i^j. Therefore none of the p_i's can be the NN of q if $d(q, p_i) > |e|$. ■

Two pruning rules are used during top-down traversal of RNN-C tree when answering R1NN queries. Recall that each cluster in the RNN-C tree has a centroid and radius that defines a MBC that covers all clusters in its subtree. When traversing down the RNN-C tree, for each cluster, we first determine whether we can prune off a cluster C_i^j by using the two lemmas in this section. Let edge $e_{\max} \in C_i^j$ such that $\forall e_m \in C_i^j$, $e_{\max} \geq e_m$, if $d(q, c_i^j) > e_{\max}$ and $d(q, c_i^j) > 2r_i^j$ then C_i^j can be pruned. If C_i^j does not meet either one of conditions, we recursively traverse each $p \in C_i^j$ which represents $C_{i-1}^{j'}$ where $j' = 1$ to $|C_i^j|$. A point to note is the need to compute $d(p, \text{NN}(p))$ $\forall p \in C_i^j$, which could best be implemented as a hash table lookup (using p's index for j-th cluster at level i) since it has been pre-computed and stored during tree construction. This would take O(1) time. Alternatively, if such information is not available, $|NN(p)| << n_i$ as $NN(p) \in C_i^j$, therefore we only need to search within the same cluster, which is typically < 10 points.

```
R1NN-C-tree-query(q, i, j, R)
// Input:  query point q, current tree level i, cluster index j
// Output: R - the R1NN results of q
begin
  if i = 0 then  // leaf level
     forall p in C[0,j] do
        if dist(q, p) ≤ dist(p, NN(p)) then
           R ← R ∪ {p}
        endif
     endfor
  else  // intermediate level
     // find max edge length in the cluster
     maxEdgeLen ← 0
     forall p in C[i,j] do
        dist ← dist(p, NN(p))
        // NN(p) is searched within C only
        if dist > maxEdgeLen then
           maxEdgeLen ← dist
        endif
     endfor

     // 2*radius and max edge length pruning rules
     if dist ≤ 2*C[i,j].radius or dist ≤ maxEdgeLen then
        for t = 1 to |C[i,j]| do
           R1NN-C-tree-query(q, i-1, t, R)
        endfor
     endif
  endif
end; {procedure R1NN-C-tree-query}
```

Figure 89. RNN-C tree query algorithm for k=1

When the traversal reaches a cluster at the leaf level, all the data points in the cluster will be checked to determine the correct R1NN results for q. Recall that the RNN-C tree uses pruning techniques for metric space, therefore we can only prune based on distance function alone and there must be no assumption made on the coordinates or relative positions of a cluster to another.

7.4 R*k*NN Queries with RNN-C Tree

The logic for pruning intermediate clusters is similar to MBR of R-tree. After much hard work, we were able to generalise our query algorithm for $k > 1$. Although the query algorithms for the case of $k = 1$ and $k > 1$ are presented separately, it is easy to combine both algorithms to provide R*k*NN query using RNN-C for any $k > 0$.

To answer R*k*NN queries for $k > 1$, we propose a technique called *the sum of clusters*. The key idea is to exploit the relationship between clusters and make use of the cluster population to prune off a cluster. This is also the key difference between RNN-C tree and TYM. TYM was not able to make use of its node size for pruning; it merely uses the distance from a node to its parent to save on computation cost. The following lemma describes the sum of clusters technique, and Figure 90 provides a sketch of the proof.

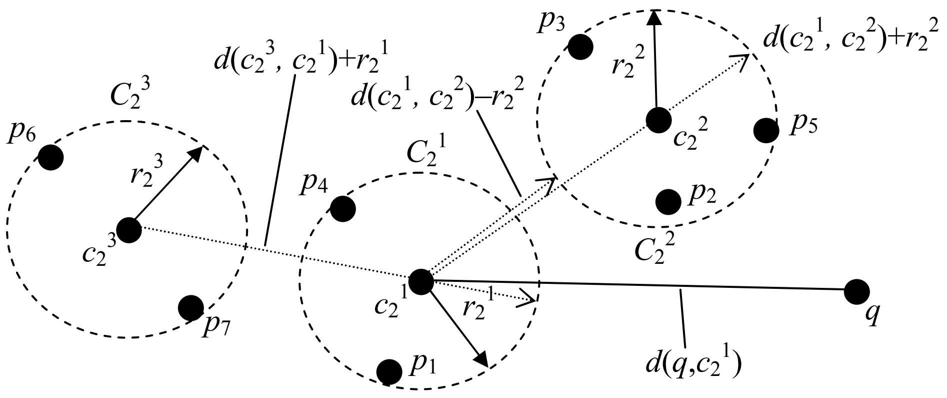


Figure 90. A sketch for the proof for Lemma 7. Dotted straight lines represent the distance between 2 cluster centroids plus a radius. C_2^1 can be pruned if $k \geq \sigma_2^2$. Note that data points may not be accurately represented within a cluster

Lemma 7. Let q be a query point for a RkNN query ($k > 1$). Let $C_i^{\,z} \in C_i$ be any cluster with centroid $c_i^{\,z}$, radius $r_i^{\,z}$ and population $\sigma_i^{\,j}$ at level i of a RNN-C tree. Let $S = \{C_i^{\,j} \in C_i \mid d(c_i^{\,z}, c_i^{\,j}) - r_i^{\,j} > r_i^{\,z}$ *and* $d(c_i^{\,z}, c_i^{\,j}) + r_i^{\,z} + r_i^{\,j} < d(q, c_i^{\,z}), j \neq z\}$ be a set of clusters which met the condition. If $\sum_{C_i^j \in S} \sigma_i^j \geq k$ then $C_i^{\,z}$ can be pruned.

Proof. S is derived from clusters meeting two conditions. The first condition $d(c_i^{\,z}, c_i^{\,j}) - r_i^{\,j} > r_i^{\,z}$ (can be rewritten as $d(c_i^{\,z}, c_i^{\,j}) > r_i^{\,z} + r_i^{\,j}$) means that $C_i^{\,z} \cap C_i^{\,j} = \varnothing$. The second condition $d(c_i^{\,z}, c_i^{\,j}) + r_i^{\,z} + r_i^{\,j} < d(q, c_i^{\,z})$ means that the furthest possible data point in $C_i^{\,j}$ w.r.t. $c_i^{\,z}$ is closer to $c_i^{\,z}$ than $c_i^{\,z}$ is to q. Note that since $d(c_i^{\,z}, c_i^{\,j}) > r_i^{\,z} + r_i^{\,j}$, so $2(r_i^{\,z} + r_i^{\,j}) < d(q, c_i^{\,z})$, and it follows that $d(q, c_i^{\,z}) \pm r_i^{\,z} > 2r_i^{\,j}$. Thus, a cluster $C_u \in S$ with corresponding population σ_u means that there are at least σ_u points closer to any point $p \in C_i^{\,z}$ than q, because $C_i^{\,z} \cap C_i^{\,j} = \varnothing$. Since all $C_u \in S$ satisfy the two conditions, therefore if $\sum\sigma_u \geq k$, no points in $C_i^{\,z}$ can be a RkNN of q. ■

Notice that the sum of clusters rule requires that we find all the clusters in the "band" outside $C_i^{\,z}$ but inside q, i.e. MBC($c_i^{\,z}$, $d(c_i^{\,z}, q)$) – $C_i^{\,z}$. As a matter of fact, the sum of clusters is a two-pronged approach. Besides looking for clusters $C_u \in S$, at the same time the query algorithm determines whether C_u can be pruned w.r.t. $C_i^{\,z}$ (the current cluster under processing). This is called the *mirror pruning rule*. The mirror pruning rule identifies in advance the clusters that can be pruned, so that they are bypassed straight away in the main processing loop. The mirror rule works on the conjecture that C_u in at least half the search space could be pruned, if the sheer size of $C_i^{\,z}$ satisfies $|C_i^{\,z}| \geq k$.

In the example of Figure 90, both C_2^2 and C_2^3 are in the set S. Assuming $\sigma_2^1 \geq k$, the mirror pruning rule causes C_2^3 to be marked as pruned, because $d(c_2^3, c_2^1)+r_2^1+r_2^3 < d(c_2^3, q)$ which means C_2^1 (with population σ_2^1) lies between C_2^3 and q, so no points in C_2^3 can be RkNN(q). However, C_2^2 is not marked as pruned as it does not satisfy the distance condition.

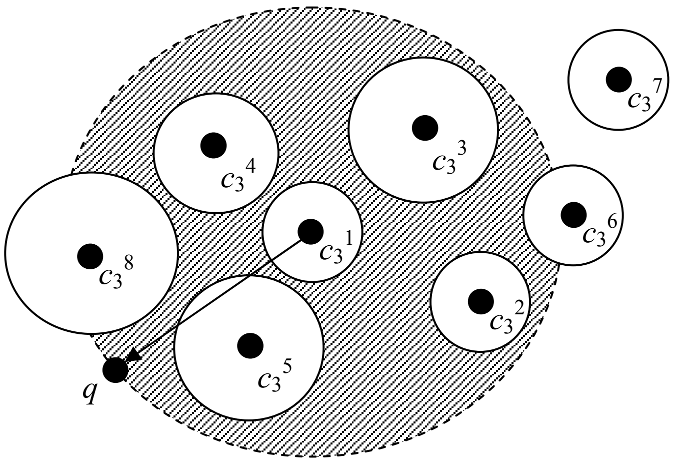


Figure 91. Illustration of the band (shaded area) between C_3^1 and q. Three clusters are disqualified by the sum of clusters rule testing. Four clusters exist within this band and therefore eligible for mirror pruning rule testing (eventually C_3^5 failed but the rest passed)

```
RkNN-C-tree-query(q, i, k, R)
// Input:  query point q, current tree level i, the number of RkNN k
// Output: R - the RkNN results of q
begin
  forall C ∈ Ci do
     C.pruned ← false
  endfor

  forall C ∈ Ci do
     if C.pruned = true then continue;  // C was pruned by mirror rule
     S ← ∅  // remember all clusters in C' lying between C and q
     sum ← 0   // sum of points closer to cluster C than q
     dist ← d(q, C.centroid)
     forall C' ∈ Ci do
        if C = C' then continue;  // skip same cluster
        dist2 ← d(C.centroid, C'.centroid)
        if dist2 - C'.radius > C.radius and
           dist2 + C'.radius + C.radius < dist then
           // mirror pruning rule
           if C'.pruned = false and C.population ≥ k and
              dist2 + C.radius + C'.radius < d(q, C'.centroid) then
              C'.pruned ← true
           endif

           // sum of clusters pruning rule
           S ← S ∪ C'
```

```
            sum ← sum + C'.population
            if sum ≥ k then
               C.pruned ← true
               break;  // C is pruned, sum need not be fully updated
            endif
         endif
      endfor
      if C.pruned = false then  // sum fully updated if C is unpruned
         if i = 0 then
            if sum + C.population ≤ k then
               R ← R ∪ C  // all points in C are valid results
            else
               refineCluster(q, k, C, S, R)
            endif
         else
            RkNN-C-tree-query(q, i-1, k, R)  // traverse the cluster
         endif
      endif
   endfor
end; {procedure RkNN-C-tree-query}

refineCluster(q, k, C, S, R)
// Input:  query point q, the number of RkNN k, current cluster C,
//         set of clusters in band S
// Output: R - the RkNN results of q
begin
   forall p in C do
      count ← 0  // count the number of points nearer to p than q
      dist ← d(p, q)

      // process points within cluster C first
      forall p' in C do
         if p = p' then continue;  // skip same point
         if d(p, p') < dist then
            count ← count + 1
            if count ≥ k then goto next p;  // continue main loop
         endif
      endfor
      // process the clusters in band
      forall T ∈ S do
         if d(T.centroid, q) + T.radius < dist then
            count ← count + T.population
            if count ≥ k then goto next p;  // continue main loop
         else  // look into individual points in T
            forall p' ∈ T do
               if d(p, p') < dist then
                  count ← count + 1
                  if count ≥ k then goto next p;  // continue main loop
               endif
            endfor
         endif
      endfor
      // p is a result since count < k
      R ← R ∪ {p}
   endfor
end; {procedure refineCluster}
```

Figure 92. RNN-C tree query algorithm for *k*>1

During query processing, the running population sum (i.e. $\sum\sigma_u \mid C_u \in S$) is kept. As soon as we encountered k points, $C_i^{\,z}$ is pruned and we proceed to the next point immediately. Suppose at the end of the loop, less than k points are encountered, it means that there are either no clusters within the band, or their combined population sum cannot lead us to conclude that there are at least k points closer to the cluster $C_i^{\,z}$ than q. In this case, for intermediate levels, we will traverse down the RNN-C tree recursively and process all the clusters under $C_i^{\,z}$. At the leaf level, if $\sum\sigma_u + \sigma_i^{\,z} \leq \mathrm{k}$ then all points in $C_i^{\,z}$ qualifies as the RkNN of q. Otherwise, we know that only a partial set of points in $C_i^{\,z}$ qualifies. To find out which, we had to refine the cluster $C_i^{\,z}$.

The `refineCluster` step is necessary because if $C_i^{\,z}$ is not pruned, it means that there are not enough points ($\sum\sigma_u < k$, where σ_u is the corresponding population of $C_u \in S$) found in between $C_i^{\,z}$ and q. However, the sum of clusters technique only applies on $C_i^{\,z}$'s surrounding clusters. When this technique fails to prune $C_i^{\,z}$, we will have to consider the data points within because now for the cluster $C_i^{\,z}$, $\min\{|C_u|\} < k < |C_i^{\,z}|+\sum\sigma_u$ is true. This means that in the best case, a data point $p \in C_i^{\,z}$ is in RkNN(q) because $\min\{|C_u|\} < k$ for p with $\min\{d(p, q)\}$. In the worst case, p is not in RkNN(q) since $|C_i^{\,z}|+\sum\sigma_u > k$ for p with $\max\{d(p, q)\}$. The `refineCluster` procedure filters off those $p \notin$ RkNN(q) by counting whether enough siblings p' of p exist between p and q (that is, p' exist such that $d(p, p') < d(p, q)$). Next, we proceed to the clusters in the band. Here we segregate those clusters C_u which satisfy the condition $d(C_u, q) + r_u < d(p, q)$ and those that do not. For those that satisfy, we could merely add σ_u to the count and save the computation cost for each member of C_u.

7.5 Experiments and Results

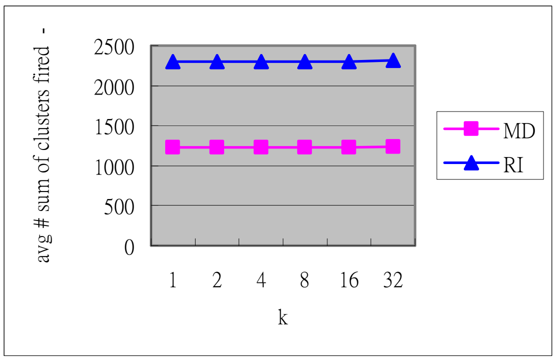
The TPL and TYM algorithms were chosen for the exact approach category, to pit against the RNN-C tree. The former uses the well-known half-plane pruning technique and is extremely fast. The latter is the only approach to solve the R*k*NN problem in metric space, to the best of our knowledge. As the RNN-C tree is also designed for answering R*k*NN in metric space, TYM is the only true state-of-the-art competitor for RNN-C tree at present.

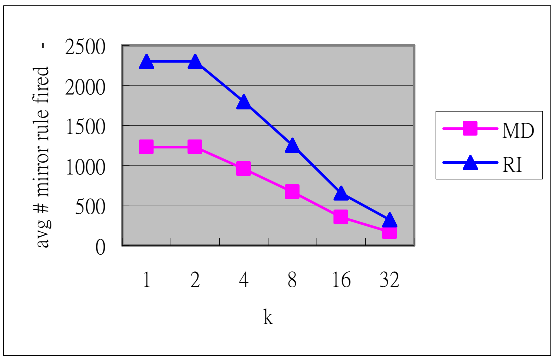
The experiment settings are similar to those described in the experiments for RNN-Grid algorithms, found in Section 6.3.1.

7.5.1 Effect of Pruning Rules

Firstly, we check the effect of the sum of clusters pruning rule and the mirror pruning rule. Counters were used to capture the number of times the two rules fired, and they are enabled only for the purpose of counting in this subsection. We used two real-life datasets, MD and RI, from the TIGER/Line database, with 4K disk page and 500 R*k*NN queries to obtain the average.

Figure 93(a) shows the number of sum of clusters being activated to prune away the current cluster under investigation. On the average, the figures decrease slightly with *k*, but very slowly. The average number of sum of clusters is directly related to the number of clusters available at any particular level of the RNN-C tree. This rule is fired when the total sum of the population for those clusters that satisfy the conditions of Lemma 7, is greater than *k*. In actual implementation, the rule is fired as soon as the total sum exceeds *k*, whereby the current cluster is pruned off and processing continues with the next cluster. For any *k*, the total sum for the same dataset is constant, so as *k* increases, the rule is fired less. Our empirical results suggest that it takes a very large *k* to reduce the effect of this pruning rule.





(a) (b)

Figure 93. The average number of (a) sum of clusters rule and (b) mirror pruning rule fired in MD and RI datasets

Figure 93(b) shows the number of mirror pruning rules fired for the two datasets. Here it is more evident that the number of times it fired decreases as *k* increases. A pre-requisite for this pruning rule to fire is $\sigma \geq k$, where *C* is the current cluster under investigation and σ is *C*'s population (Figure 92). Typically, $|C| \leq 4$ at the leaf level. Hence, when $k > 4$, the incidence of this rule firing is significantly reduced especially at higher levels of the tree. When processing the RNN-C tree at higher levels (nearer to the root), the mirror pruning rule is more useful. Note that the mirror pruning rule does not reduce the #distcomp directly, as it still incurs one distcomp per cluster. However, it helps prune off clusters earlier so that subsequent processing in the main loop is able to skip the pruned clusters.

Table 30. The average number of pruning rules fired at different levels of the RNN-C tree for MD dataset across $1 \leq k \leq 32$

h	Avg # of sum of clusters	Avg # of mirror pruning rule
0	609.8	230.0
1	406.0	277.5
2	133.8	103.2
3	56.0	48.8
4	25.0	18.0

Table 30 depicts the average number of incidence of both pruning rules at different levels of the RNN-C tree for the MD dataset. Note that $h = 0$ is the leaf (data) level, and $h = 4$ is the root. At high levels, the number of clusters is small and σ is large. Assuming $|C| = 4$, the probability of sum of clusters firing is 0.348% and mirror pruning is 0.252%. At the leaf level, however, the probability increased to 8.48% and 3.2% respectively.

7.5.2 Performance Comparisons

The RNN-C tree is compared against the TPL and TYM algorithms, both exact R*k*NN algorithms. We used the real-life MD dataset from the TIGER/Line database, with 4K disk page and 500 R*k*NN queries to obtain the average.

In Figure 94, the #distcomp is compared. TPL has the lowest #distcomp among the three algorithms. In fact, at $k = 1$, it is approximately two orders of magnitude smaller than RNN-C tree and TYM. However, the growth of TPL is huge. When $k = 32$, it incurs 56477.91 distcomp, which is 33.84 times the distcomp when $k = 1$. Even so, it is just half of the #distcomp of RNN-C tree. The lower #distcomp of the TPL algorithm is explained by the use of geometrical properties (bisector pruning) in the Euclidean metric space to filter candidates. Bisector pruning reduces the #distcomp between subsequent data points by a very large factor. RNN-C tree and TYM, both being generic metric space algorithm, do not benefit from any properties that only work in one distance metric.

The RNN-C tree generally sees a growth in the #distcomp as well, but relatively constant for $k \leq 32$. The TYM, on the other hand, grows streadily, and it is almost 4 times larger than RNN-C tree when k is large.

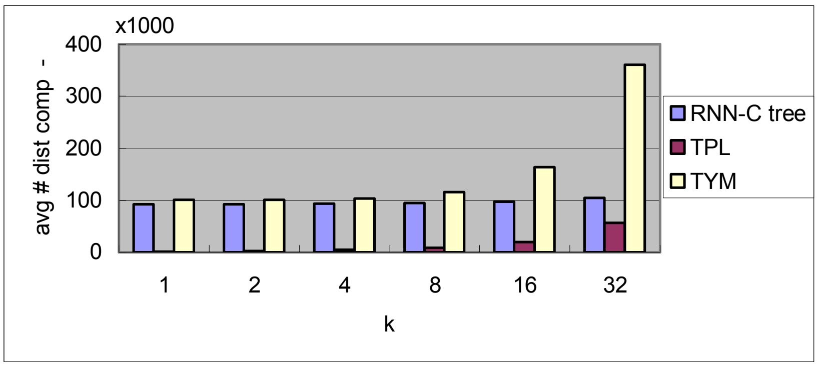


Figure 94. Comparison of number of distance computations in TIGER/Line MD dataset of RNN-C tree, TPL and TYM

In Figure 95, the average number of disk access indicates that the TPL algorithm's disk access grows as k increases. This is because the TPL algorithm is based on the R-tree index, so the larger the k, the more objects will have to be accessed for use in refining the candidate set. Our results show that the R-tree does not support RNN type of queries efficiently, as even the average number of I/Os needed to answer R1NN is almost 1.8 times that of RNN-C tree. The R-tree was originally designed for answering NN queries. Owing to the nature of RNN, where an answer can be very far away from a query point q, the MBR of an R-tree is not a good choice (since it minimises the solution space which is of course good for NN queries). The RNN-C tree and the TYM algorithm require almost the same number of disk accesses, which is expected because both structures store data points in at most one branch of the tree and both use similar pruning techniques involving the radius of a cluster (for RNN-C tree) or node (TYM).

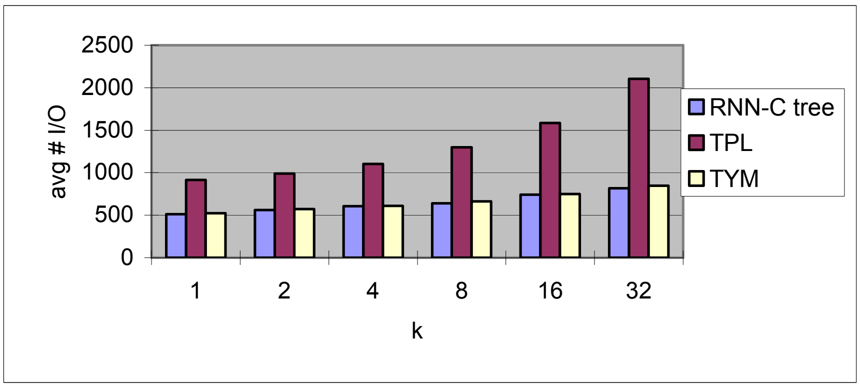


Figure 95. Comparison of # I/Os in TIGER/Line MD dataset of RNN-C tree, TPL and TYM

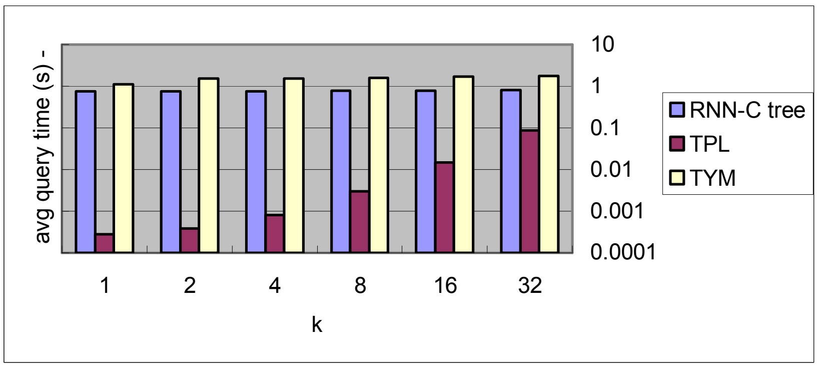


Figure 96. Comparison of query cost (s) in TIGER/Line MD dataset of RNN-C tree, TPL and TYM

Finally, the overall cost of a query is tied to the number of disk accesses and #distcomp. For TPL, its query time increases as *k* increases but it is still faster than both RNN-C tree and TYM. Although TPL is shown to have a higher disk access cost than the other two, it actually incurs far fewer #distcomp due to its bisector pruning of candidates. Adding to the fact that the gap between I/O cost and CPU cost is closing, TPL stands to benefit since it is disk access intensive. Overall, TPL is still the fastest of the three algorithms. The major disadvantage for TPL is that it only works for RNN queries in Euclidean metric. When k >> 32, one can see the benefits of a generic metric distance algorithm. The average query time will be lower than TPL.

For TYM and RNN-C tree, both incurs high "startup" cost whereby any RNN query will be sure to incur. Beyond the initial startup cost, the growth in query time is actually very small and negligible. This high cost is attributed to the minimum #distcomp and disk accesses needed in order to traverse down the RNN-C tree structure to look for the correct results, as data points are stored at the leaf level. The M-tree in which TYM is based on, also stores data points at leaf level, thereby exhibiting the same high startup cost.

The maximum values for the performance comparisons for all 3 metrics are shown in Table 31. The trends are similar to the average cases. The number of max disk accesses is very similar for both RNN-C tree and TYM, but the max query time for RNN-C tree is about half of TYM's. For $k \leq 8$, the maximum #distcomp and disk accesses for RNN-C tree and TYM are comparable, but RNN-C is 2.02 times faster.

Table 31. Performance comparison (max values) of RNN-C tree, TPL and TYM for the TIGER/Line MD dataset

k	max #distcomp			max #I/Os			max query time (s)		
	RNN-C tree	TPL	TYM	RNN-C tree	TPL	TYM	RNN-C tree	TPL	TYM
1	94721.8	2338.4	101216	543	941	554	0.75	0.00260	1.14
2	95109.3	3015.2	101356	572	1010	610	0.78	0.01236	1.54
4	102511.2	5536.1	104513	622	1142	622	0.78	0.01493	1.56
8	106326.9	9472.9	115848	663	1309	689	0.79	0.02102	1.61
16	110808.6	21217.3	163492	753	1607	775	0.79	0.04768	1.69
32	112400.3	58242.8	361101	854	2125	860	0.83	0.09225	1.79

7.6 Summary

In this chapter, the RNN problem, a relatively new kind of query, was explored. We proposed a novel data structure to solve the exact RNN problem for any *k*, giving full accurate results. The RNN-C tree is a unique tree based on the *k*NN graph, where the dataset is pre-processed and connected as a forest of very small subgraphs with the 1NN relation. We showed that the RNN-C tree can be used to answer R*k*NN queries efficiently.

The RNN-C tree is compared to other algorithms in the same class (exact results), such as TPL and TYM. RNN-C tree and TYM are the only two algorithms designed to work in metric space, as long as a distance function is defined between two objects. RNN-C tree is shown to be faster and more efficient than the TYM, because it prunes effectively based on both inter-cluster distances as well as cluster population.

We strongly believe that RNN-C tree has potential in RNN queries and it has wide applications because of its minimal requirements (only a distance function). In the coming chapter, we propose several further problems for RNN-C tree.

Chapter 8 Conclusion and Future Work

8.1 Conclusion

In conclusion, this research has addressed two major issues in MPRQ. Firstly, we researched into various techniques used to solve MPRQ. As a result, we discovered three approaches that can be used, presented their algorithms and analysed each of them in detail. Intelligent pruning rules form the key for the good query time that MPRQ enjoys. Extensive experiments were carried out to understand the MPRQ in a wide variety of problem parameters and MPRQ performs well in all of them against the conventional technique RRQ and state-of-the-art spatial join algorithms used in many proximity queries today. Secondly, we adapted the best results from our study into an application of a vastly different domain of computer science: bioinformatics.

MPRQ can be solved with the MPRQ-MinMax, MPRQ-Sorted Path or MPRQ-Rectangle Intersection approach. The most straightforward method is MPRQ-MinMax which is easy to implement and deploy in any applications that do proximity queries. We showed that MPRQ does even better as search distance and the number of query points increase, and the overall total query time grows very slowly. We investigated the effect of applying different combinations of pruning rules, and found out the reasons behind MPRQ's good performance and the effect of pruning rules have on MPRQ.

It is also shown that MPRQ can be used with other structures such as SOM to perform sequence similarity search to identify peptides in the bioinformatics domain. The results we obtained are very encouraging – our

PepSOM algorithm (which contains MPRQ) is as fast as the best *de novo*-database hybrid approach at present, and PepSOM database filtration rate is high without sacrificing peptide similarity accuracies.

For the RNN problem, we proposed two different approaches which are highly effective. For an everyday application that does not require accurate results (approximate RNNs will do) but does require fast response time, we proposed the RNN-Grid that is very efficient and has very high recall. We have shown that the RNN-Grid is fast even for solving RNN of high-dimensional datasets. We proposed three algorithms for the RNN-Grid and conducted an in-depth study of their performances, as well as when compared to other estimated RNN algorithms.

Applications that require exact answers for a RNN query will benefit from our proposed novel data structure, called the RNN-C tree, which is able to answer R*k*NN queries in any metric space. The RNN-C tree is useful in many applications such as decision making, outlier detection, data mining, data retrieval, etc. As long as there is a defined distance function between any two objects in a dataset, and it satisfies the triangle inequality principle, the RNN-C tree can be used to solve R*k*NN queries given any query object. To the best of our knowledge, the RNN-C tree is one of only two RNN algorithms that work with data points in metric space, the other being the TYM algorithm. And RNN-C is shown to outperform TYM.

8.2 Future Work for MPRQ

This thesis leaves a number of topics unexplored and the issues highlighted here can be further pursued in future as possible extensions or new research directions. They are broadly classified as (i) considering velocity and trajectory in the input, and (ii) finding *k*NN for MPRQ.

8.2.1 Velocity and Trajectory

One area of further work could be to extend the model of MPRQ to include the ability to process temporal information in addition to spatial information.

It was explained in our research scope that the time domain will not be considered in this research because results can be easily processed with time information when the spatial query is done. Our experiments also showed that including time specific pruning into our MPRQ algorithms does not make much sense as the number of pruned spatial points is not significant since the MPRQ results contain points that are relatively static. However, this is not true if the spatial points move.

Moving spatial points (where each point has assigned velocity and direction) in a spatial index might make more sense to pursue research in this direction. For example, in addition to the relatively static spatial index of events, there exists another index for moving points (say, vehicles), then work can be done to answer MPRQ w.r.t. both indexes. This area of research is new and there are many work done [ŠJLL00, AgAE00, TPZL05] on a single moving query point but not a single moving query point in pre-planned path.

8.2.2 *k*-Nearest Neighbour MPRQ

Can the MPRQ be used to answer *k*NN queries? In this research, we stated that the main objective is to find *all* the events that are close to a given planned route in the fastest time. It might be possible to apply ranking to all the points within the result set such that, when given a value *k*, we are able to find the top *k*-nearest points closest to the path. Or perhaps, a further extension would be to find the top *k*-nearest points closest to each and every query point.

The ability to answer *k*NN queries using the multi-point range query is useful in many ways. Suppose the MPRQ represents a particular line of telephone poles running through a residential area, the events being the nearby houses whose telephones are connected to it. It is very common task to identify the top *k* houses that lie closest to the poles because telephone lines run through them before reaching their neighbours next door. This can help facilitate the maintenance and troubleshooting of faulty or noisy lines.

8.3 Future Work for RNN-C Tree

It is believed that the RNN-C holds immense potential to solve other variants of the RNN problem. Possible future new research directions in this area can be broadly classified as follows: (i) extension for processing multiple R*k*NN queries simultaneously in one tree traversal, (ii) designing the RNN-C tree to be a dynamic structure, (iii) using RNN-C tree to solve the bichromatic version of the RNN problem, and (iv) continuous tracking of a moving query point.

8.3.1 Multi-point R*k*NN Problem

In the spirit of MPRQ, a possible future extension to the RNN problem is to design algorithms to answer R*k*NN for multiple query points simultaneously.

This is a more challenging problem than a single query point. Since the notion of R*k*NN(q) represents the influence of q as within the top k NN of some data point $p \in \mathrm{R}k\mathrm{NN}(q)$, the motivation for the RNN problem has always been the belief that any changes in q will affect p. So it isn't hard to imagine that given Q as a set of query points and k, find RkNN(Q) efficiently. It can also be argued, like the RRQ, that we execute the RkNN(q) query separately $\forall q \in Q$ and join the outcome results. However, optimisations in the query might be possible in the RNN-C tree if we know in advance that there is more than one query point.

Possible uses include diverse applications in decision support systems, continuous referral systems and maintaining document repositories. For example, in a document repository, the NN relationship is based on similarities between two technical documents already filed. When a batch of new technical documents in the same category are filed, the repository can execute a RkNN(Q) to retrieve the authors of all similar documents and let them know of the possibly interesting new entries.

8.3.2 Dynamic RNN-C Tree Structure

In this research work, we focused on solving the RNN problem with the assumption that the underlying dataset is static. It will be highly convincing to claim that a dataset of spatial nature will require far fewer updates than when the dataset is of another domain, say data mining or information retrieval. At this point, the construction algorithm for the RNN-C tree, described in Section 7.2, does not delve into methods for inserting a new data point or for deleting existing data points in an already constructed RNN-C tree. In order to do so, as the structure of the RNN-C tree is dependent on the 1NN graph derived from the coordinates of the data points (instead of the order in which they are encountered), efficient techniques similar to [SPKS03] can be discovered.

An initial strategy to answer the RNN query when it was first proposed involved constructing a duplicate R-tree (the original is used for NN queries) called RNN-tree [KoMu00] where the leaf nodes store vicinity circles (VC) instead of the point (the RNN-tree and VCs were mentioned in Section 5.3). This is obviously not efficient because two R-trees have to be maintained. Hence, future work to make the RNN-C tree structure dynamic must be directly effected on the data structure itself. We also believe that lazy deletion of data points is possible with the RNN-C tree, especially when the MBC on intermediate nodes of the RNN-C tree is not affected. These two issues make a good direction to explore.

8.3.3 Bichromatic RNN and Beyond

Can the RNN-C tree be adapted to solve the bichromatic RNN problem? Given a set *TDB* of sites, a set *SDB* of points, and a query site q, B-RNN(q) = $\{p \in SDB \mid \forall s \in TDB, d(q, p) \leq d(p, s)\}$. Currently the RNN-C tree is constructed from a 1NN graph of single-coloured points. There are at least two possible techniques to extend the RNN-C tree for B-RNN: (i) via the *k*NN graph, and (ii) by constructing one RNN-C tree each for *SDB* and *TDB*. In the first method, all points and sites are set on the plane and we "capture and label" the colour of the points before building a RNN-C tree. Extra information about the minimum distance between a point and a site might have to be stored in the MBC on intermediate levels of the tree. For the second method, two RNN-C trees may be constructed and the traversal proceeds in tandem for both trees, for pruning conditions checking.

A more daring proposition is to solve the *k*-chromatic RNN challenge, for any general *k* number of sites. To the best of our knowledge, there is no research work that addresses trichromatic RNN and beyond.

8.3.4 Moving Query Point

Tracking of a continuously moving query point q to answer RNN queries has received some attention recently [XiZh06, BJKS07, KMSX07, WYCT08]. For the continuous-RNN problem, given a set SDB of points, some time interval T_j and moving query point q, the goal is to keep track of $\mathrm{RNN}_j(q)$ where $\mathrm{RNN}_j(q) = \{p \in SDB \mid \forall o \in SDB, d(q, p) \leq d(p, o)\}$ at time interval T_j. The assumption is that all $p \in SDB$ are continuously moving in non-predictable fashion, in addition to the moving query point q. A variant of the continuous-RNN is where the input is a set of query points Q.

Continuous-RNN queries are useful for location-aware applications such as mixed-reality games and vehicle traffic monitoring systems where positions of objects and query points are frequently updated. For example, in a battlefield, all soldiers may be issued a GPS device each that not only can pinpoint their location, but also perform a continuous-RNN query for a particular soldier to monitor other nearby comrades who might be wounded and require help.

Let us focus on the case for a single moving query point. For continuous-RNN queries, the RNN-C tree must be sensitive to monitoring regions. These regions are defined around the query point, so that when the underlying dataset points change outside of them, it guarantees that the query results will not be affected (that is, updated correctly).

Bibliography

[AFST07] Apaydin T., Ferhatosmanoglu H., Singh A. & Tosun A.S., 2007, “A Unified Method for Multi-dimensional NN, RNN, and Matching Queries”, *Technical Report OSU-CISRC-8/07-TR58*, Ohio State University.

[AgAE00] Agarwal P.K., Arge L. & Erickson J., 2000, “Indexing Moving Points”, *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS 2000)*, pp.175-186.

[AHVV99] Arge L., Hinrichs K.H., Vahrenhold J. & Vitter J.S., 1999, “Efficient Bulk Operations on Dynamic R-Trees”, *Proceedings of the 1st Workshop on Algorithms Engineering & Experimentation (ALENEX 99)*, LNCS 1619: 328-348.

[AnSa96] Ang C.H. & Samet H., 1996, “Approximate Average Storage Utilization of Bucket Methods with Arbitrary Fanout”, *Nordic Journal of Computing*, 3(3): 280-291.

[AnTa97] Ang C.H. & Tan T.C., 1997, “New Linear Node Splitting Algorithm for R-Trees”, *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD 97)*, LNCS 1262: 339-349.

[APRS98] Arge L., Procopiuc O., Ramaswamy S., Suel T. & Vitter J.S., 1998, “Scalable Sweeping-Based Spatial Join”, *Proceedings of the 24th International Conference on Very Large Databases (VLDB 98)*, pp. 570-581.

[ASKK06] Abe T., Sugawara H., Kanaya S., Kinouchi M. & Ikemura T., 2006, “Self-Organizing Map (SOM) Unveils and Visualizes Hidden Sequence Characteristics of a Wide Range of Eukaryote Genomes”, *Gene*, 365: 27-34.

[BaMc72] Bayer R. & McCreight E.M., 1972, “Organization and Maintenance of Large Ordered Indexes”, *Acta Informatica*, 1: 173-189.

[BeGe01] Bertone P. & Gerstein M., 2001, “Integrative Data Mining: The New Direction in Bioinformatics”, *IEEE Engineering in Medicine and Biology Magazine*, 20(4): 33-40.

[BJKS07] Benetis R., Jensen C.S., Karĉiauskas G. & Ŝaltenis S., 2007, “Nearest and Reverse Nearest Neighbor Queries for Moving Objects”, *The International Journal on Very Large Data Bases*, 15(3): 229-249.

[BKSS90] Beckmann N., Kriegel H.P., Schneider R. & Seeger B., 1990, “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles”, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 19(2): 322-331.

[BrKS93] Brinkhoff T., Kriegel H.P. & Seeger B., 1993, “Efficient Processing of Spatial Joins using R-Trees”, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 237-246.

[CaWe07] Cannon W. R. & Webb-Robertson B., 2007, “Computational Proteomics: High-Throughput Analysis for Systems Biology”, *Pacific Symposium on Biocomputing*, 12: 403-408.

[Chan01] Chan E.P.F., 2001, “Evaluation of Buffer Queries in Spatial Databases”, *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*, LNCS 2121: 197-216.

[CiPZ97] Ciaccia P., Patella M. & Zezula P., 1997, “M-tree: An Efficient Access Method for Similarity Search in Metric Spaces”, *Proceedings of the 23rd International Conference on Very Large Databases (VLDB 97)*, pp. 426-435.

[CMAB98] Crauser A., Mehlhorn K., Althaus E., Brengel K., Buchheit T., Keller J., Krone H., Lambert O., Schulte R., Thiel S., Westphal M. & Wirth R., 1998, “On the performance of LEDA-SM”, *Technical Report*, Max-Planck Institut für Informatik, Germany.

[CNLP06] Chong K.F., Ning K., Leong H.W. & Pevzner P., 2006, “Characterization of Multi-Charge Mass Spectra for Peptide Sequencing”, *Proceedings of the 4th Asia-Pacific Bioinformatics Conference (APBC 2006)*, pp. 109-119.

[Corr02] Corral, A., 2002, “Algorithms for Processing of Spatial Queries using R-trees. The Closest Pairs Query and its Application on Spatial Databases”, *Ph.D. Thesis*, Department of Languages and Computation, University of Almeria, Spain.

[CrMe99] Crauser A. & Mehlhorn K., 1999, "LEDA-SM: A Platform for Secondary Memory Computation", *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE 99)*, LNCS 1668: 228-242.

[DACV99] Dancik V., Addona T., Clauser K.R., Vath J.E. & Pevzner P.A., 1999, "De novo Protein Sequencing via Tandem Mass-Spectrometry", *Journal of Computational Biology*, 6(3-4): 327-342.

[DDKN06] Desiere F., Deutsch E.W., King N.L., Nesvizhskii A.I., Mallick P., Eng J., Chen S., Eddes J., Loevenich S.N. & Aebersold R., 2006, "The PeptideAtlas Project", *Nucleic Acids Research*, 34: D655-D658.

[DeKS05] Dementiev R., Kettner L. & Sanders P., 2005, "Stxxl: Standard Template Library for XXL Data Sets", *Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005)*, LNCS 3669: 640-651.

[DoC98] "Global Positioning System Market Projections and Trends in the Newest Global Information Utility", International Trade Administration, Office of Telecommunications, U.S. Department of Commerce, September 1998.

[DoC01] "Trends in Space Commerce", Office of Space Commercialization, U.S. Department of Commerce, June 2001.

[Doli01] Dolin R., 2001, "TIGER Files and Their Applications", Department of Urban & Regional Planning, University of Illinois at Urbana-Champaign, http://www.urban.uiuc.edu/Courses/up330/old_essays/2001_Dolin_w04.pdf

[EnMY94] Eng J.K., McCormack A.L. & Yates J.R., 1994, "An Approach to Correlate Tandem Mass Spectral Data of Peptides with Amino Acid Sequences in a Protein Database", *Journal of the American Society for Mass Spectrometry*, 5(11): 976-989.

[EpPY97] Eppstein D., Paterson M.S. & Yao F.F., 1997, "On Nearest-Neighbor Graphs", LNCS 623: 416-426.

[FiBe74] Finkel R.A. & Bentley J.L., 1974, "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, 4: 1-9.

[FLLL99] Foo H.M, Lao Y.Z., Leong H.W. & Lau H.C., 1999, "A Multi-Criteria, Multi-Modal Passenger Router Advisory System", *Proceedings of the IES-CTR International Symposium on Advanced Technologies in Transportation*.

[FrPe05] Frank A. & Pevzner P., 2005, "PepNovo: De Novo Peptide Sequencing via Probabilistic Network Modeling", *Analytical Chemistry*, 77(4): 964 -973.

[FTBP05] Frank A., Tanner S., Bafna V. & Pevzner P., 2005, "Peptide Sequence Tags for Fast Database Search in Mass Spectrometry", *Journal of Proteome Research*, 4(4): 1287-1295.

[Fuku90] Fukunaga K., 1990, "Introduction to Statistical Pattern Recognition", 2nd ed., Academic Press, October 1990.

[GaGü98] Gaede V. & Günther O., 1998, "Multidimensional Access Methods", *ACM Computing Surverys*, 30(2): 170-231.

[GaLL98] García Y.J, López M.A. & Leutenegger S.T., 1998, "A Greedy Algorithm for Bulk Loading R-Trees", *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS 98)*, pp. 163-164.

[GBDJ94] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. & Sunderam V.S., 1994, "PVM: Parallel Virtual Machine", MIT Press, November 1994.

[Gold99] Goldwasser M., 1999, "The Sixth DIMACS Implementation Challenge", Center for Discrete Mathematics & Theoretical Computer Science, Rutgers The State University of New Jersey, http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html

[GrLS94] Gropp W., Lusk E. & Skjellum A., 1994, "Using MPI: Portable Parallel Programming with the Message-Passing Interface", MIT Press, Cambridge, October 1994.

[GSR01] "GIS Software and Resources", GeoPlan Center, University of Florida, http://www.geoplan.ufl.edu/software.html

[GüSh87] Güting R.H. & Schilling, W., 1987, "A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem", *Information Sciences*, 42(2): 95-112.

[Gutt84] Guttman A., 1984, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 14(2): 47-57.

[Hilb91] Hilbert D., 1891, "Ueber stetige Abbildung einer Linie auf ein Flächenstück", *Mathematische Annalen*, 38: 459-460.

[HjSa98] Hjaltason G.R. & Samet H., 1993, "Incremental Distance Join Algorithms for Spatial Databases", *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 27(2): 237-248.

[HKCL03] Hwang S., Kwon K, Cha S.K. & Lee B.S., 2003, "Performance Evaluation of Main-Memory R-tree Variants", LNCS 2750: 10-27.

[Ho00] Ho N.L., 2000, "Proximity Search for Route Advisory Systems", *Honours Year Project Report*, Dept. of Computer Science, School of Computing, National University of Singapore.

[JOTY05] Jagadish H.V., Ooi B.C., Tan K.L., Yu C. & Zhang R., 2005, "iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search", *ACM Transactions on Database Systems*, 30(2): 364-397.

[KaFa93] Kamel I. & Faloutsos C., 1993, "On Packing R-trees", *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM 93)*, pp. 490-499.

[KaFa94] Kamel I. & Faloutsos C., 1994, "Hilbert R-tree: An Improved R-tree Using Fractals", *Proceedings of the 20th International Conference on Very Large Databases (VLDB 94)*, pp. 500-509.

[KaKK98] Kaski S., Kangas J. & Kohonen T., 1998, "Bibliography of Self-Organizing Map (SOM) Papers: 1981-1997", *Neural Computing Surveys*, 1: 102-350.

[KaSa01] Katayama N. & Satoh S., 2001, "Distinctiveness-Sensitive Nearest-Neighbor Search for Efficient Similarity Retrieval of Multimedia Information", *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, pp. 493-502.

[KHKL96] Kohonen T., Hynninen J., Kangas J. & Laaksonen J., 1996, “SOM_PAK: The Self-Organizing Map Program Package”, *Technical Report A31*, Laboratory of Computer and Information Science, Helsinki University of Technology.

[KMNP99] Kanungo T., Mount D.M., Netanyahu N.S., Piatko C., Silverman R. & Wu A., 1999, “Computing Nearest Neighbors for Moving Points and Applications to Clustering”, *Proceedings of 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 99)*, pp. 931-932.

[KMSX07] Kang J.M., Mokbel M.F., Shekhar S., Xia T. & Zhang D., 2007, “Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors”, *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pp. 806-815.

[Knot71] Knott G.D., 1971, “Expandable Open Addressing Hash Table Storage and Retrieval”, *Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pp. 186-206.

[Knut98] Knuth D.E., 1998, “The Art of Computer Programming”, 2nd ed., Addison-Wesley, Redwood City, California, April 1998.

[Koho01] Kohonen T., 2001, “Self-Organizing Maps”, *Springer Series in Information Sciences*, Vol. 30, 3rd ed., Springer-Verlag, Berlin, January 2001.

[KoMu00] Korn F. & Muthukrishnan S., 2000, “Influence Sets Based on Reverse Nearest Neighbor Queries”, *Proceedings of the 2000 ACM SIGMOD Iinternational Conference on Management of Data*, pp. 201-212.

[KPNS02] Keller A., Purvine S., Nesvizhskii A.I., Stolyar S., Goodlett D.R. & Kolker E., 2002, “Experimental Protein Mixture for Validating Tandem Mass Spectral Analysis”, *Omics*, 6(2): 207-212.

[KrHS91] Kriegel H.P., Horn H. & Schiwietz M., 1991, “The Performance of Object Decomposition Techniques for Spatial Query Processing”, *Proceedings of the Second International Symposium on Advances in Spatial Databases* (*SSD 91)*, LNCS 525: 257-276.

[KrSB93] Kriegel H.P., Schneider R. & Brinkhoff T., 1993, “Potentials for Improving Query Processing in Spatial Database Systems”, *Neuvièmes Journées Bases de Données Avancées*, pp. 11-35.

[Lao99] Lao Y.Z., 1999, “Multi-Modal Route Planning in Public Land Transportation”, *Honours Year Project Report*, Dept. of Computer Science, School of Computing, National University of Singapore.

[LaTh92] Laurini R. & Thompson D., 1992, “Fundamentals of Spatial Information Systems”, *APIC Series in Data Processing*, Vol. 37, Academic Press, London, March 1992.

[LeEL97] Leutenegger S., Edgington J. & Lopez M., 1997, “STR: A Simple and Efficient Algorithm for R-Tree Packing”, *Proceedings of the 13th International Conference on Data Engineering (ICDE 97)*, pp. 497-506.

[LeLo00] Leutenegger S. & Lopez M.A., 2000, “The Effect of Buffering on the Performance of R-Trees”, *IEEE Transactions on Knowledge and Data Engineering*, 12(1): pp 33-44.

[LiNY03] Lin K., Nolen M. & Yang C., 2003, “Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems”, *Proceedings of the 7th International Database Engineering & Applications Symposium (IDEAS 2003)*, pp. 290-297.

[LuOo93] Lu H.J. & Ooi B.C., 1993, “Spatial Indexing: Past and Future”, *IEEE Data Engineering Bulletin*, 16(3): 16-21.

[MaDo94] Martin J.L. & Dongarra J., 1994, “MPI: A Message-Passing Interface Standard”, *International Journal of Supercomputer Applications and High Performance Engineering*, 8(3-4): 165-416.

[MaHN84] Matsuyama T., Hao L.V. & Nagao M., 1984, “A File Organisation for Geographic Information Systems Based on Spatial Proximity”, *Computer Vision, Graphics and Image Processing*, 26(3): 303-318.

[MaMo01] Maneewongvatana S. & Mount D.M., 2001, “An Empirical Study of a New Approach to Nearest Neighbor Searching”, *Proceedings of the 3rd International Workshop on Algorithm Engineering and Experimentation (ALENEX 2001)*, LNCS 2153: 172-187.

[MaPa03] Mamoulis N. & Papadias D., 2003, “Slot Index Spatial Join”, *IEEE Transactions on Knowledge and Data Engineering*, 15(1): 211-231.

[MaVZ02] Maheshwari A., Vahrenhold J. & Zeh N., 2002, "On Reverse Nearest Neighbor Queries (Extended Abstract)", *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG 2002).*

[MeNä95] Mehlhorn K. & Näher S., 1995, "LEDA: A Platform for Combinatorial and Geometric Computing", *Communications of the ACM*, 38(1): 96-102.

[MMSG04] Mahony S., McInerney J.O., Smith T.J. & Golden A., 2004, "Gene Prediction using the Self-Organizing Map: Automatic Generation of Multiple Gene Models", *BMC Bioinformatics*, 5(23).

[MTI09] "Singapore Resident Population 1990-2009" and "Land Transport", 2009, Department of Statistics, Ministry of Trade and Industry, Singapore, http://www.singstat.gov.sg/keystats/people.html

[MZHL03] Ma B., Zhang K., Hendrie C., Liang C., Li M., Doherty-Kirby A. & Lajoie G., 2003, "PEAKS: Powerful Software for Peptide De Novo Sequencing by MS/MS", *Rapid Communications in Mass Spectrometry*, 17(20): 2337-2342.

[NgLe04] Ng H.K. & Leong H.W., 2004, "Path-Based Range Query Processing Using Sorted Path and Rectangle Intersection Approach", *Proceedings of the 9th International Conference on Database Systems for Advanced Applications (DASFAA 2004)*, LNCS 2973: 184-189.

[NgLe07] Ng H.K. & Leong H.W., 2007, "Multi Point Queries in Large Spatial Databases", *Proceedings of the 3rd IASTED International Conference on Advances in Computer Science and Technology (ACST 2007)*, pp. 408-413.

[NgLH04] Ng H.K., Leong H.W. & Ho N.L., 2004, "Efficient Algorithm for Path-Based Range Query in Spatial Databases", *Proceedings of the 8th International Database Engineering & Applications Symposium (IDEAS 2004)*, pp. 334-343.

[NgNL07] Ng H.K., Ning K. & Leong H.W., 2007, "A New Approach for Similarity Queries of Biological Sequences in Databases", *Proceedings of the 11th Pacific-Asia Knowledge Discovery and Data Mining conference (PAKDD 2007)*, pp. 728-736.

[NiHS84] Nievergelt J., Hinterberger H. & Sevcik K.C., 1984, "The grid file: An adaptable, symmetric multikey file structure", *ACM Transactions on Database Systems*, 9(1): 38-71.

[NiNL06] Ning K., Ng H.K. & Leong H.W., 2006, "PepSOM: An Algorithm for Peptide Identification by Tandem Mass Spectrometry Based on SOM", *Genome Informatics*, 17(2): 194-205.

[NiWi97] Nievergelt J. & Widmayer P., 1997, "Spatial Data Structure: Concepts and Design Choices", *Proceedings of Algorithmic Foundations of Geographic Information Systems*, LNCS 1340: 153-197.

[OjKK03] Oja M., Kaski S. & Kohonen T., 2003, "Bibliography of Self-Organizing Map (SOM) Papers: 1998-2001 Addendum", *Neural Computing Surveys*, 3: 1-156.

[Oren82] Orenstein J.A., 1982, "Multidimensional Tries Used For Associative Searching", *Information Processing Letters*, 14(4): 150-157.

[PaMa96] Papadopoulos A. & Manolopoulos Y., 1996, "Parallel Processing of Nearest Neighbor Queries in Declustered Spatial Data", *Proceedings of the 4th ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS 96)*, pp. 35-43.

[PaMa98] Papadopoulos A. & Manolopoulos Y., 1998, "Multiple Range Query Optimization in Spatial Databases", *Proceedings of the 2nd East European Symposium on Advances in Databases and Information Systems (ADBIS 98)*, LNCS 1475: 71-82.

[PCWL04] Prince J.T., Carlson M.W., Wang R., Lu P. & Marcotte E.M., 2004, "The Need for a Public Proteomics Repository", *Nature Biotechnology*, 22(4): 471-472.

[PeDT00] Pevzner P.A., Dancik V. & Tang C.L., 2000, "Mutation-Tolerant Protein Identification by Mass Spectrometry", *Journal of Computational Biology*, 7(6): 777-787.

[PPCC99] Perkins, D.N., Pappin, D.J.C., Creasy, D.M., & Cottrell, J.S., 1999, "Probability-based Protein Identification by Searching Sequence Databases using Mass Spectrometry Data", *Electrophoresis*, 20(18): 3551-3567.

[PrSh85] Preparata F.P. & Shamos M.I., 1985, "Computational Geometry: An Introduction", pp. 351-355, Springer-Verlag, New York, August 1985.

[PSTM04] Papadias D., Shen Q., Tao Y. & Mouratidis K., 2004, “Group Nearest Neighbor Queries”, *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, pp. 301-312.

[RMNP06] Ramakrishnan S.R., Mao R., Nakorchevskiy A.A., Prince J.T., Willard W.S., Xu W., Marcotte E.M. & Miranker D.P., 2006, “A Fast Coarse Filtering Method for Peptide Identification by Mass Spectrometry”, *Bioinformatics*, 22(12): 1524-1531.

[RoKV95] Roussopoulos N., Kelly S. & Vincent F., 1995, “Nearest Neighbor Queries”, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 71-79.

[Same89] Samet H., 1989, “The Design And Analysis of Spatial Data Structure”, Addison-Wesley, Reading, August 1989.

[Same06] Samet H., 2006, “Foundations of Multidimensional and Metric Data Structures”, Morgan Kaufmann Publishers, San Francisco, 2006.

[SCRF99] Shekhar S., Chawla S., Ravada S., Fetterer A., Liu X. & Lu. C.T., 1999, “Spatial Databases: Accomplishments and Research Needs”, *IEEE Transactions on Knowledge and Data Engineering*, 11(1): 45-55.

[SeKi02] Sebastian T.B. & Kimia B.B., 2002, “Metric-based shape retrieval in large databases”, *Proceedings of the 16th International Conference on Pattern Recognition (ICPR 2002)*, pp. 291-296.

[SeKr90] Seeger B. & Kriegel H.P., 1990, “The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems”, *Proceedings of the 16th International Conference on Very Large Databases (VLDB 90)*, pp. 590-601.

[SeKr98] Seidl T. & Kriegel H.P., 1998, “Optimal Multi-Step k-Nearest Neighbor Search”, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 27(2): 154-165.

[SeRF87] Sellis T.K., Roussopoulos N. & Faloutsos C., 1987, “The R^+-tree: A Dynamic Index for Multi-Dimensional Objects”, *Proceedings of the 13th International Conference on Very Large Databases (VLDB 87)*, pp. 507-518.

[ShLi97] Shekhar S. & Liu D.R., 1997, "CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations", *IEEE Transactions on Knowledge and Data Engineering*, 9(1): 102-119.

[ShML02] Shin H., Moon B. & Lee S., 2002, "Adaptive and Incremental Processing for Distance Join Queries", *Technical report 02-03*, Department of Computer Science, The University of Arizona.

[SiFT03] Singh A., Ferhatosmanoglu H. & Tosun A.S., 2003, "High Dimensional Reverse Nearest Neighbor Queries", *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM 2003)*, pp. 91-98.

[ŠJLL00] Šaltenis S., Jensen C.S., Leutenegger S.T. & Lopez M.A., 2000, "Indexing the Positions of Continuously Moving Objects", *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 29(2): 331-342.

[SoRo01] Song Z. & Roussopoulos N., 2001, "K-Nearest Neighbor Search for Moving Query Point", *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*, LNCS 2121: 79-96.

[SPKS03] Skopal T., Pokorný J., Krátký M. & Snášel V., 2003, "Revisiting M-Tree Building Principles", *Proceedings of the 7th East European Conference on Advances in Databases and Information Systems (ADBIS 2003)*, LNCS 2798: 148-162.

[SRAE01] Stanoi I., Riedewald M., Agrawal D. & El Abbadi A., 2001, "Discovery of Influence Sets in Frequently Updated Databases", *Proceedings of the 27th International Conference on Very Large Data Bases*, pp. 99-108.

[StAE00] Stanoi I., Agrawal D. & El Abbadi A., 2000, "Reverse Nearest Neighbor Queries for Dynamic Databases", *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pp. 44-53.

[Sund90] Sunderam V.S., 1990, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice & Experience*, 2(4): 315-339.

[TaJo97] Taylor J.A. & Johnson R.S., 1997, "Sequence Database Searches via De Novo Peptide Sequencing by Tandem Mass Spectrometry", *Rapid Communications in Mass Spectrometry*, 11(9): 1067-1075.

[TaJo01] Taylor J.A. & Johnson R.S., 2001, “Implementation and Uses of Automated De Novo Peptide Sequencing by Tandem Mass Spectrometry”, *Analytical Chemistry*, 73(11): 2594-2604.

[TaLe04] Tan J.S. & Leong H. W., 2004, “Least-Cost Path in Public Transportation Systems with Fare Rebates That Are Path- and Time-Dependent”, *Proceedings of the 7th International IEEE Conference on Intelligent Transportation Systems (ITSC 2004)*, pp. 1000-1005.

[TaPL04] Tao Y., Papadias D. & Lian X., 2004, “Reverse kNN Search in Arbitrary Dimensionality”, *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004)*, pp. 744-755.

[TaYM06] Tao Y., Yiu M.L. & Mamoulis N., 2006, “Reverse Nearest Neighbor Search in Metric Spaces”, *IEEE Transactions on Knowledge and Data Engineering*, 18(9): 1239-1252.

[ThSe96] Theodoridis Y. & Sellis T., 1996, “A Model for the Prediction of R-tree Performance”, *Proceedings of the 15th ACM Symposium on Principles of Database Systems (PODS 96)*, pp. 161-171.

[TIGER02] “Topologically Integrated Geographic Encoding and Referencing system – TIGER/Line®”, U.S. Census Bureau, February 2002, http://www.census.gov/geo/www/tiger/

[TPZL05] Tao Y., Papadias D., Zhai J. & Li Q., 2005, “Venn Sampling: A Novel Prediction Technique for Moving Objects”, *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pp. 680-691.

[Veng94] Vengroff D.E., 1994, “A Transparent Parallel I/O Environment”, *Proceedings of the 3rd DAGS Symposium on Parallel Computation and Problem Solving Environments (DAGS 94)*, pp.117-134.

[VTST93] Vonderohe A.P., Travis L., Smith R.L. & Tsai V., 1993, “Adaptation of Geographic Information Systems for Transportation”, *NCHRP Report 359*, Transportation Research Board, National Academy Press, Washington D.C.

[Will85] Willard D.E., 1985, “New Data Structures for Orthogonal Range Queries”, *SIAM Journal of Computing*, 14(1): 232-253.

[WYCT08] Wu W., Yang F., Chan C.Y. & Tan K.L., 2008, "Continuous Reverse k-Nearest-Neighbor Monitoring", *Proceedings of the 9th International Conference on Mobile Data Management (MDM 2008)*, pp. 132-139.

[XiHL05] Xia C., Hsu W. & Lee M.L., 2005, "ERkNN: Efficient Reverse kNearest Neighbors Retrieval with Local kNNDistance Estimation", *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM 05)*, pp. 533-540.

[XiZh06] Xia T. & Zhang D., "Continuous Reverse Nearest Neighbor Monitoring", *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006)*, pp. 77-86.

[YaLi01] Yang C. & Lin K., 2001, "An Index Structure for Efficient Reverse Nearest Neighbor Queries", *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, pp. 485-492.

[YOTJ01] Yu C., Ooi B.C., Tan K.L. & Jagadish H.V., 2001, "Indexing the Distance: An Efficient Method to KNN Processing", *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, pp. 421-430.

[YPMT06] Yiu M.L., Papadias D., Mamoulis N. & Tao Y., "Reverse Nearest Neighbors in Large Graphs", *IEEE Transactions on Knowledge and Data Engineering*, 18(4): 540-553.

[ZhZh95] Zhang H. & Zhong D., 1995, "A Scheme for Visual Feature-Based Image Indexing", *Proceedings of the 3rd Conference on Storage and Retrieval for Image and Video Databases (SPIE 95)*, 2420: 36-46.

[ZMPT04] Zhang J., Mamoulis N., Papadias D. & Tao Y., 2004, "All-Nearest-Neighbors Queries in Spatial Databases", *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004)*, pp. 297-306.

Appendix A PepSOM: An Application of MPRQ-Disk

We briefly describe an example of the real-life application of MPRQ in addition to RADS: the integration of MPRQ and the self-organising map (SOM) to serve as a coarse filter for identifying peptides (short proteins) as one of the most challenging problems in the bioinformatics (also known as computational biology) domain – peptide identification (and in general the biological sequence similarity problem [NgNL07]). A novel algorithm called PepSOM [NiNL06] is developed which provides for a non-trivial method for transforming spectrum similarity (a representation of peptides) to similarity of vectors, and then to neighbourhood similarity of points in 2-d plane.

A.1 Peptide Identification in Bioinformatics

Peptide identification by tandem mass spectrometry (MS/MS) is a challenging problem in proteomics. Current high throughput mass spectrometers [CaWe07] have generated a huge amount of spectra, and the analysis of these spectra must keep pace. Fast algorithms for peptide identification are crucial for such analysis.

Unfortunately, the process of analyzing these spectrum data is still slow and not accurate. Approaches for peptide identification can be categorized into database search algorithms [EnMY94, FTBP05, PPCC99] or *de novo* algorithms [DACV99, FrPe05, MZHL03, TaJo97]. The former are suitable for known peptide sequences that already exist in the database. However, they apparently do not perform well in discovering new peptide sequences not already available in database. For such peptide sequences, the *de novo* algorithms are the method of choice. *De novo* algorithms work from the ground up by interpreting peptide sequences from spectrum data purely by analyzing the intensity and correlation of the peaks in the spectrum data.

In the peptide identification problem, database search usually return the peptide sequences that match the parent mass of the spectrum. However, the accuracy depends on the quality of the database, and the process is slow (usually a few minutes). Typical analyses of an LC/LC/MS/MS experimental dataset using the popular BioWorks program by ThermoFinnigan with a single processor take several hours for computation (e.g. 30,000 scans against the *Escherichia coli* database). The *de novo* algorithm can find tags with high accuracy [CNLP06, FTBP05], and the process is fast (always within 1 minute) but tags are usually not complete sequences for the spectra.

Hence, how to achieve a *balance between identification completeness and efficiency yet manage reasonable accuracy* for peptide identification by tandem mass spectrum is an important consideration. This is where our proposed novel algorithm comes in. PepSOM is an algorithm using the database search approach but it is very fast, without the slow processing problems that plagued other database-based approach. It identifies candidate peptide sequences by selection from database via a technique by the combination of SOM [Koho01] and MPRQ (course filter), then scores and ranks these peptide sequences (fine filter) by comparing their theoretical spectrum with the experimental spectrum. Since the candidates are essentially found by database search algorithm, all the candidates in database that are similar (whose number are controlled with MPRQ's search distance *d*) to the experimental spectrum are retrieved. With this technique, completeness and efficiency are achieved with reasonable accuracy attained.

Recently, coarse and fine filtering methods commonly associated with database search techniques were introduced for peptide identification [RMNP06]. The spectra are mapped to vectors, and using a metric space indexing algorithm, initial candidates for later fine filtering were produced. A variant of shared peaks count (SPC) scoring function was used to compute the similarity among spectra. The coarse filtering can reduce the number of candidates to about 0.5% of the database and for fine filtering, a Bayesian scoring scheme is applied on candidate spectra to more accurately identify peptide sequences.

A.2 Problem Description

Proteomics is the study of proteins expressed by a genome. They are systematically studied by cataloguing and analysing proteins to determine when a particular protein is expressed, its expression level (amount expressed), and how proteins interact with one another. By studying proteins, we could determine the types of proteins present in normal vs diseased cells. We can also identify drug targets as well as discover new drugs for treatment of illnesses.

A typical MS/MS proteomics process calls for individual proteins to be separated via a process called 2-d PAGE (two-dimensional poly acrylamide gel electrophoresis). Proteins are first isolated and then sliced into parent peptides by enzymatic digestion, which usually involve the enzyme trypsin. The parent peptides are then ionised and isolated from each other. One of the methods to perform peptide isolation is by high-performance liquid chromatography (HPLC), and peptides are further separated by their mass-to-charge ratios (m/z). This forms the first stage of the mass spectrometry (MS) process. In tandem mass spectrometry (MS/MS), an isolated peptide (target) is then sent through collision-induced dissociation (CID) causing it to fragment into many pieces. The m/z of each and every piece is measured to obtain an MS/MS spectrum. Figure 97 illustrates.

Definition (Theoretical spectrum): *The ion fragmentation pattern of a particular peptide, usually stored on databases, derived from training data or expert opinion. Typically it is represented as a chart of peak intensity vs mass-to-charge ratio (m/z).*

Definition (Experimental spectrum): *The ion fragmentation pattern of a particular peptide derived from an MS/MS process, is a set of mass peak of fragment ions.*

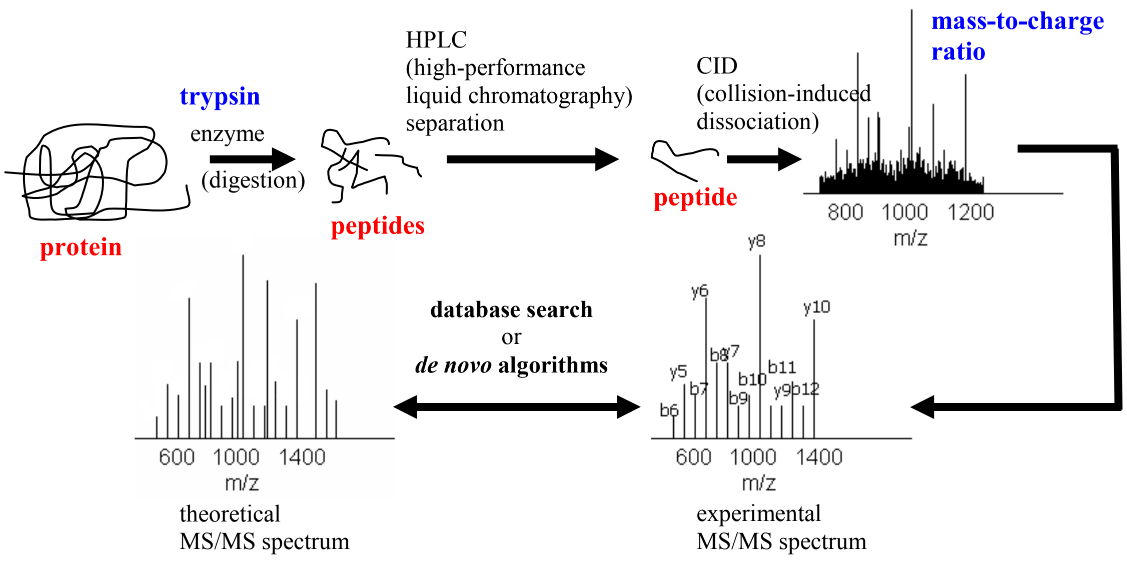


Figure 97: An example of LC/MS/MS peptide identification process

Peptide identification can be used to identify proteins present in a sample. In a perfect world, an oracle would be able to look at the sample and tell us exactly what proteins are contained therein. In reality, we must derive the experimental spectrum of a peptide via the MS/MS process. Unfortunately this process is not perfect, and it also introduces noise into the experimental spectrum, making it harder to compare with theoretical spectra to identify the correct peptide. Sources of noise include from MS instruments, the loss of water (H_2O) and ammonia (NH_3) during fragmentation and post-translation modifications (enzymes altering the protein after the translation process) such as phosphorylation, glycosylation, myristoylation or methylation. This is where algorithms like PepSOM fit in. PepSOM will efficiently process multiple experimental spectra and quickly derive peptides from databases that are similar to them.

A.3 PepSOM Algorithm

We first describe SOM and MPRQ followed by some notes on converting spectra to vectors (binning of peaks). Next, we present our novel peptide identification algorithm, PepSOM.

A.3.1 Self-Organising Map

SOM is a method for unsupervised learning, based on a grid of artificial neurons whose weights are adapted to match input vectors in a training set. In the training process, a SOM (map) is built and the neural network organises itself using a competitive process. The SOM usually consists of a two-dimensional regular grid of nodes. The node whose weights are closest to the input vector, termed the best-matching or winner node, is updated to be more similar to it while the winner's surrounding neighbours are also updated (to a smaller extent) to be more similar to the input vector. As a result, when a SOM is trained over a few thousand epochs, it gradually evolves into clusters whose data (peptides) are characterised by their similarity. Therefore, it is very suitable for analysis of the similarities among sequences and is very widely used [KaKK98, OjKK03]. Increasingly, SOM is used as an efficient and powerful tool for analysing and extracting a wide range of biological information as well as for gene prediction [BeGe01, MMSG04, ASKK06].

For spectrum data, each node represents an observation of the spectrum (converted to vector), and the distance between nodes represent their similarities. The closer two nodes are located to each other, the more similar they are. For a visual illustration, we give an example of SOM with 995 spectra (the ISB test dataset, which we will describe in Section A.4) on a 50×50 grid. Figure 98(a) illustrates the *relationship* among these spectra. Observe that some of the spectra (black dots) are clustered together and are hard to distinguish. Many spectra are surrounded by grey dots representing similar vectors (updated by SOM algorithm during training phase but not representing any spectrum in particular). It follows that spectrum similarities are represented by neighbourhoods of the points on SOM.

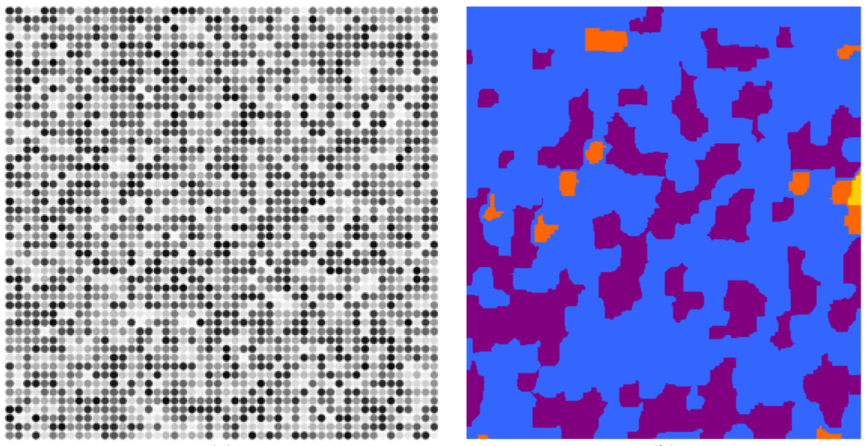


Figure 98: (a) In this example of SOM generated from spectra, each spectrum is represented by a grayscale dot. Notice that neighbouring dots have mutually similar shades of grey. (b) A sample of SOM training of *Escherichia coli* for a 100x100 orthogonal grid being visualized. Similar colours represent similarity of trained sequences

A.3.2 Multi-Point Range Query

MPRQ is an important component of the PepSOM algorithm. It provides a fast mechanism for peptide similarity queries.

Once the theoretical spectra for the peptide sequences in the database are mapped as 2-d points on a SOM, they are indexed with our KDTopDownPack bulk-loaded R-tree data structure since the peptide sequences database rarely change. The spatial index can then be reused many times. To perform similarity query, we transform the experimental spectra into query points in 2-d plane and proceed to query. At this point, it is possible to use many experimental spectra as the query simultaneously, which translates to multiple points as the input for MPRQ algorithm.

Experiments showed that a large input (up to 1000 experimental spectra or more) *does not* increase the overall query time by much. This phenomenon is due to the intelligent pruning rules NodeIn and PointOut embedded within the MPRQ algorithm. Apart from a set of query points, the MPRQ algorithm also accepts as input a parameter *d* that controls the radius of the search distance. The larger the value of *d*, the more candidate peptides will be returned. MPRQ can efficiently process the multiple input points *simultaneously* with respect to *d* and the MBRs during query, effectively performing multi-spectra similarity search (which is adjustable) on a database of known peptides.

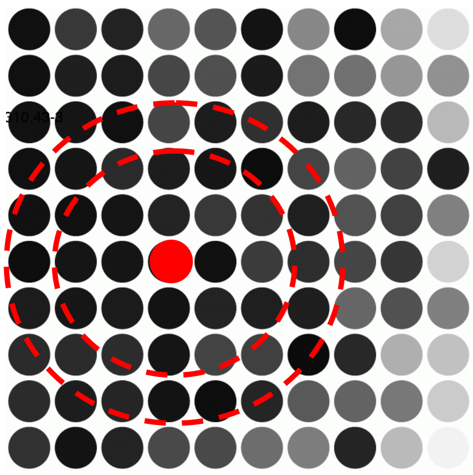


Figure 99: Applying MPRQ on the SOM map to retrieve peptide similarity candidates. The search distance *d* can be used to control the number of candidates desired to achieve a tradeoff balance between efficiency (query time) and accuracy

A.3.3 Converting Spectra to Vectors

The very first step of PepSOM is to convert spectra in database to high-dimensional vectors of the same dimension in vector space. The PepSOM algorithm requires both theoretical and experimental spectra to be converted to statistical vectors so that the SOM can be trained and queried. This is related to the binning of the peaks in spectrum. The binning idea was used in [PeDT00] for mass spectrum alignment. In [PeDT00], the intensity peaks of a spectrum are packed into many bins, and the spectrum was translated into sequences comprising 0's and 1's. We used a similar method for binning, except that our binning results are sequences of real numbers.

Binning is used to remove noisy peaks from a spectrum while converting them into vectors. A less noisy spectrum translates into more accurate identification results and faster processing time as fewer peaks are considered.

The important parameters for binning of peaks include the size of the bins, the amino acids interpretation of supporting peaks (bins), the mass tolerance value as well as the peaks intensity. For simplicity, it is suffice to say that given the properly set value of mass tolerance, binning can preserve the spectrum accuracies, while at the same time decrease the computational cost greatly, especially for noisy spectra. We refer the reader to our paper [NiNL06] for precise details and proofs.

The binning process also includes scoring of bins to eliminate bins with very low peak intensity. Based on domain knowledge, the important parameters for scoring should include peak intensity, the number of supporting peaks and mass error. Based on the analysis of the scores of peaks in the spectrum, the lowest 20% bins in scores ranking, or those bins with scores less than 1% of the highest rank are filtered out.

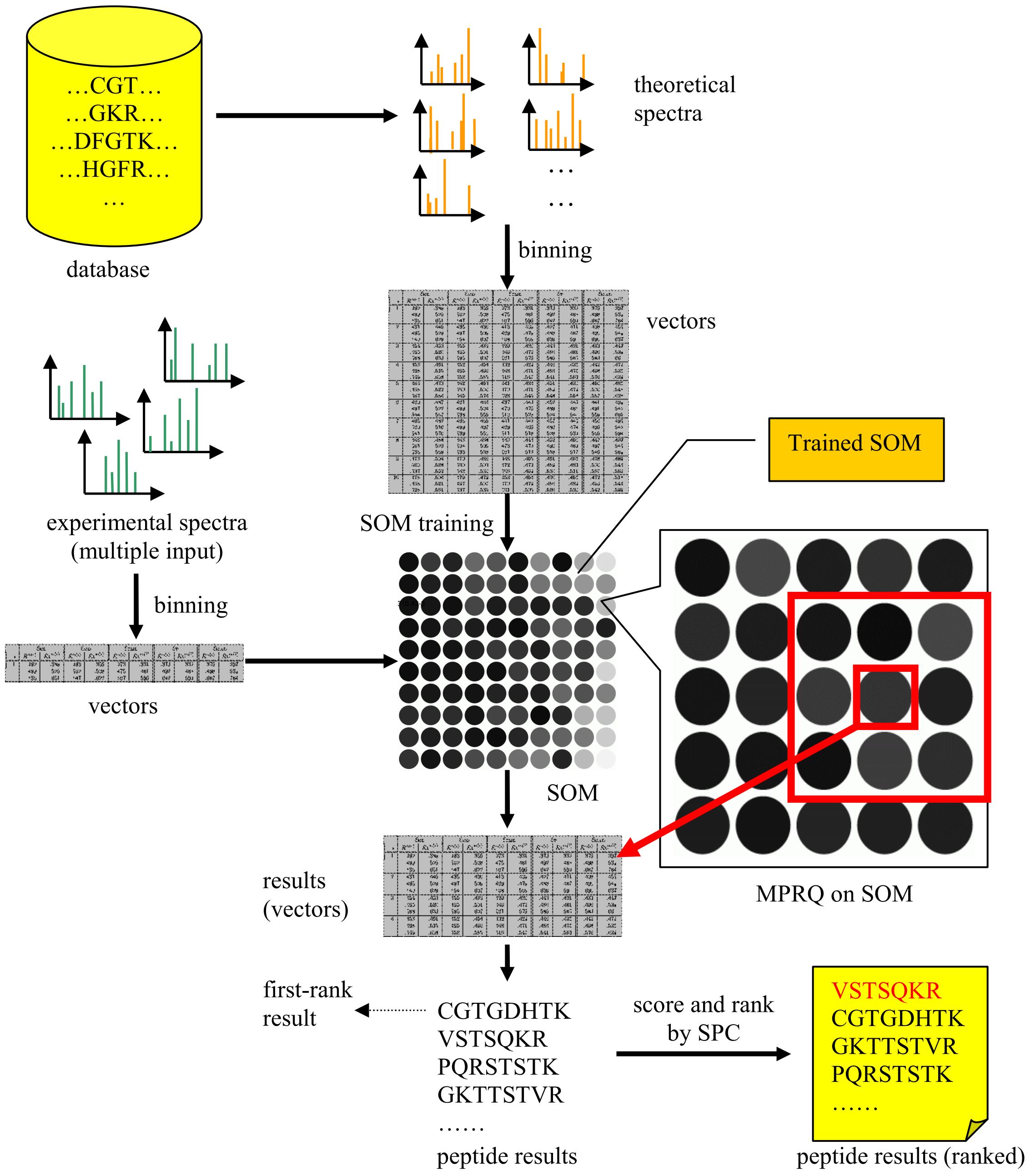


Figure 100: Diagram for the peptide identification with PepSOM

A.3.4 PepSOM

Figure 100 depicts how the PepSOM algorithm works as a coarse filtering step. Peptides from the database are converted to theoretical spectra which are further converted to high-dimensional vectors and then used to train a SOM (map). This only needs to be performed once unless the database changes.

In the query process stage, each experimental spectrum is converted to vector (via binning) and then matched with the trained SOM map to obtain its best-matching node (expressed in (x,y)-coordinates). The resulting coordinates form the basis input points for the MPRQ algorithm to perform a single, efficient similarity query. Candidate peptides are selected from the database this way, and then fine-filtered by comparing their theoretical spectrum with experimental spectrum by shared peaks count (SPC). The SPC score is computed as the number of shared peaks between experimental spectrum and theoretical spectrum of candidates (within tolerance). First rank result simply refers to the first result returned by MPRQ. While it is not necessarily the best, it gives an indication of the quality of results when a "quick result" is warranted.

```
PepSOM(DB, ES, d)
// input: peptide database DB, expt spectra ES, similarity d
// output: candidates results set C
begin
   TS ← bin all peaks of putative peptides in DB;
   V1 ← GenerateVectors(TS);
   som_map ← TrainSOM(V1);              // SOM training
   2d_map ← MapSOM(som_map, V1);        // map of (x,y)-coords
   ES ← bin all peaks of ES;  // bin ES if not previously done so
   V2 ← GenerateVectors(ES);
   Q ← MapSOM(som_map, V2);    // obtain multi points query set
   MPRQSearch(2d_map.root, Q, d, C);   // obtain candidates set C
   return C;
end; {procedure PepSOM}
```

Figure 101: Algorithm for PepSOM uses SOM and MPRQ for coarse filtering

Figure 101 lists the PepSOM algorithm. Although SOM has been used before to predict genes, this is the first attempt of its kind to combine SOM with spatial database query for peptide identification. Many efficient algorithms exist for spatial database queries in orthogonal 2-d grids or hierarchical data structures. SOM is useful because we believe it satisfies the condition that the distance on the map reflects the similarity of peptides.

A.4 Experiments

A.4.1 Experiment Settings and Datasets

Experiments were performed on a Linux machine with 3.0 GHz CPU and 1 GB RAM. PepSOM was implemented in C++ and Perl. SOM_PAK [KHKL96] was the SOM implementation used. We had selected two database search algorithms, Sequest [EnMY94] and InsPecT [FTBP05], as well as two *de novo* algorithms with freely available implementations, Lutefisk [TaJo01] and PepNovo [FrPe05], for comparison and analysis. We treated Sequest result with a cross-correlation score (Xcorr) above 2.5 as ground truth. In a typical setting, Xcorr ≥ 2.0 from Sequest is considered of good quality. We strived for more stringent results.

Spectrum datasets were obtained from the Open Proteomics Database (OPD) [PCWL04], PeptideAtlas database [DDKN06] and Institute for Systems Biology (ISB) [KPNS02]. The three datasets chosen are of vastly different sizes to enable us to examine the issue of scalability of PepSOM compared to other algorithms.

For OPD, the spectrum dataset used was opd00001_ECOLI, *Escherichia coli* spectra 021112.EcoliSol 37.1(000). The spectra were obtained from *E. coli* HMS 174 (DE3) cell, which is grown in LB medium until ~0.6 abs (OD 600). The spectra were generated by the ThermoFinnigan ESI-Ion Trap "Dexa XP Plus" and the sequences for these spectra were validated by Sequest. There are 3903 spectra in total; we chose all the 202 spectra that were identified with Xcorr ≥ 2.5.

Spectra from PeptideAtlas were also selected. The spectrum dataset A8_IP were obtained from Human *Erythroleukemia* K562 cell line. Electrospray ionization source of an LCQ Classic ion trap mass spectrometer (ThermoElectron, San Jose, CA) was used, and DTA files were generated from the MS/MS spectra using TurboSequest. The dataset consists of a total of 1564 spectra; we chose all the 44 spectra that were identified with Xcorr ≥ 2.5.

The ISB dataset was generated using an ESI source from a mixture of 18 proteins, obtained from ion trap mass spectrometry. The ISB dataset was of low quality, having between 200-700 peaks each with an average of 400 peaks. The entire dataset consists of a total of 37044 spectra; we chose all the 995 spectra that were identified with Xcorr ≥ 2.5.

The databases that we used were theoretical spectrum generated from the respective protein sequences dataset. Specifically, *E. coli* K12 protein sequences for OPD datasets, IPI HUMAN protein sequences for PeptideAtlas dataset and human plus control protein mixture for ISB dataset. As the number of protein sequences were very large for PeptideAtlas (60,090) and ISB (88,374) datasets, we used only the protein sequences corresponding to spectra identified with Xcorr $\geq$ 2.5 (our ground truth set). However, the sizes of databases were still very large because of many fragmentations.

The parameters for the generation of databases, the test datasets and theoretical spectra are shown in Table 32. Additionally, we use a search distance radius $d = 0.25$ as the MPRQ parameter.

Table 32: Parameters for the generation of databases and theoretical spectra

Parameters	Values		
	OPD	PeptideAtlas	ISB
No. of protein sequences	4,279	31	3,553
Total database size	494,049	9,421	1,248,212
Test dataset size	202	44	995
Fragments mass tolerance	0.5 Da		
Parent mass tolerance	1.0 Da		
Modifications	–		
Charge	+2, +3		
Ion type	a, b, y, $-H_2O$, $-NH_3$		
Missed cleavages	0		
Protease	Trypsin		
Mass range	0-6000 Da		

A.4.2 Accuracy Measures

The following accuracy measures were used to compare the different algorithms:

$$\text{Sensitivity} = \frac{\#correct}{|\rho|} \qquad \text{Specificity} = \frac{\#correct}{|P|}$$

where *#correct* is the *number of correctly identified amino acids*. It is computed as the longest common subsequence (LCS) of the actual correct peptide sequence ρ and the identification result P of the PepSOM algorithm. $|\rho|$ and $|P|$ depict the length of the respective peptide sequences.

Sensitivity indicates the quality of the identification result with respect to the actual correct peptide sequence – a high sensitivity being that the identification algorithm (in our experiments – InsPecT, Lutefisk, PepNovo and PepSOM) recovers a large portion of the correct peptide. For a fairer comparison with *de novo* algorithms like PepNovo that only outputs the highest scoring tags (subsequences), we also use a specificity measure, which measures the number of correctly identified amino acids within the identification result given by the algorithm (independent of the actual correct peptide sequence ρ).

A.4.3 Results and Analyses

A.4.3.1 Quality of PepSOM Results

We analyzed the quality of peptide sequences identified by PepSOM as candidates. These candidates would be tested against the experimental spectra (test size) to return the final results. Generally, the size of candidates set should be as small as possible (minimal false positives) yet able to yield the final results. The first among the results we obtained using the test set is labeled as first-rank peptide. Best-match peptide is the peptide from all candidates that match with "real" peptide with the highest specificity (and sensitivity). The latter can be thought of as an upper bound of the results obtained.

Table 33: Statistical results on the quality of candidates identification by PepSOM.
For specificity and sensitivity, the results for "first-rank peptide / best-match peptide" are shown

Datasets	Database Size	Test Size	No. of Complete Correct	Complete Correct Accuracy	Specificity	Sensitivity	Time (ms)
OPD	494,049	202	44	0.218	0.560 / 0.785	0.428 / 0.593	10.6
PeptideAtlas	9,421	44	10	0.227	0.334 / 0.377	0.445 / 0.637	10.5
ISB	1,248,212	995	116	0.117	0.529 / 0.895	0.680 / 0.726	10.8

From Table 33, it is clear that both sensitivity and specificity for PepSOM is high. For example, in the OPD dataset, both sensitivity and specificity are higher than 0.55 (best-match); as for the ISB dataset, the sensitivity is higher than 0.65 (both). There are also a significant number (10% to 25%) of completely correct peptide identifications among top-rank peptide sequences. The time taken for peptide identification is also very small; this is expected when using both SOM and MPRQ combined (more details will be provided later). The average query time per spectrum is approximately 11 ms. This is comparable to InsPecT (with average 10 ms search time per spectrum with default settings, but based on smaller database) which is one of the fastest database search algorithms because PepSOM is able to filter a small set of high quality candidates and yet keep the accuracy of the resulting set.

A.4.3.2 Performance of PepSOM

Next, we compared PepSOM with other well-known peptide identification algorithms, namely Sequest, Lutefisk, PepNovo and InsPecT among others. Recall that the Sequest algorithm provides the spectra identified with high Xcorr score (≥ 2.5). Therefore here we treated them as ground truth.

Table 34: Comparison of different algorithms on the accuracies of peptide identification. In each column, the "Specificity / Sensitivity" values are listed

Datasets	Database Size	Test Size	Sequest	InsPecT	Lutefisk	PepNovo	PepSOM
OPD	494,049	202	1.0 / 1.0	0.592 / 0.556	0.129 / 0.008	0.252 / 0.200	0.560 / 0.428
PeptideAtlas	9,421	44	1.0 / 1.0	0.811 / 0.402	0.162 / 0.063	0.291 / 0.135	0.334 / 0.445
ISB	1,248,212	995	1.0 / 1.0	0.602 / 0.633	0.032 / 0.032	0.563 / 0.593	0.529 / 0.680

We observe from Table 34 that both specificity and sensitivity of PepSOM are better than Lutefisk and PepNovo (both *de novo* algorithms), and they are comparable to InsPecT. Although InsPecT has higher specificity, our results outperform InsPecT in sensitivity. Specifically, for the OPD dataset, both the algorithms have specificity and sensitivity of about 0.55. For the PeptideAtlas dataset, the specificity of our algorithm is much worse than that of InsPecT, but the sensitivity is about 10% better. For the ISB dataset, PepSOM has lower specificity than InsPecT, but the sensitivity value is higher.

From these experiments, we note that the results for PepSOM are at best preliminary because of the use of conventional SPC scoring. We believe that by implementing an improved scoring function (e.g. incorporating statistical analysis or reliable tags generated by a *de novo* process), our results could be better. All in all, PepSOM's performance is comparable to InsPecT in both accuracy and efficiency.

A.4.3.3 Filtering Rate

One of the most important features of PepSOM is that it is very fast. For batch processing of multiple spectra query, Table 33 and Table 35 show that it can perform peptide identification for large spectrum datasets (> 500) in mere seconds (for example, 500 × 10.8 ms = 5.4 secs).

Table 35: PepSOM-generated candidates size, average query size and coarse filtering rate

Database	Database Size	Test Size	Candidates Size	Average Query Size	Coarse Filtering Rate
OPD	494,049	202	68,610	339.7	0.069%
PeptideAtlas	9,421	44	654	14.9	0.158%
ISB	1,248,212	995	101,443	102.0	0.008%

Traditional database search algorithms such as Sequest are much slower than PepSOM. Although *de novo* algorithms are usually faster than PepSOM, currently they cannot generate results with comparable accuracy. In Table 35, the candidates size represents the combined total results from coarse filtering of the database using the experimental spectra (test size) as the input query points for the MPRQ algorithm. The average query size represents the average peptide sequence candidates for each spectrum (query point). Coarse filtering rate is computed by averaging query size over the original database size. We only need to compare each spectrum against the candidates identified for it by MPRQ. Therefore, the coarse filtering rate is very low. Compared to the tandem cosine coarse filter used in [RMNP06] which filters to around 0.5% of the database, it is obvious PepSOM has a better filtering efficiency. This explains why PepSOM could achieve fast query time.

A.4.3.4 Effect of Search Distance

From Figure 102 we see that the larger search distance radius *d* that we use, the larger the average query size (due to the increased number of candidates), and the selection of *d* = 0.25 is a compromise between efficiency and accuracy. Accuracy generally improves by a little with larger *d* but it is not significant. In this application, the MPRQ input search distance *d* serves as a control mechanism for efficiency vs accuracy.

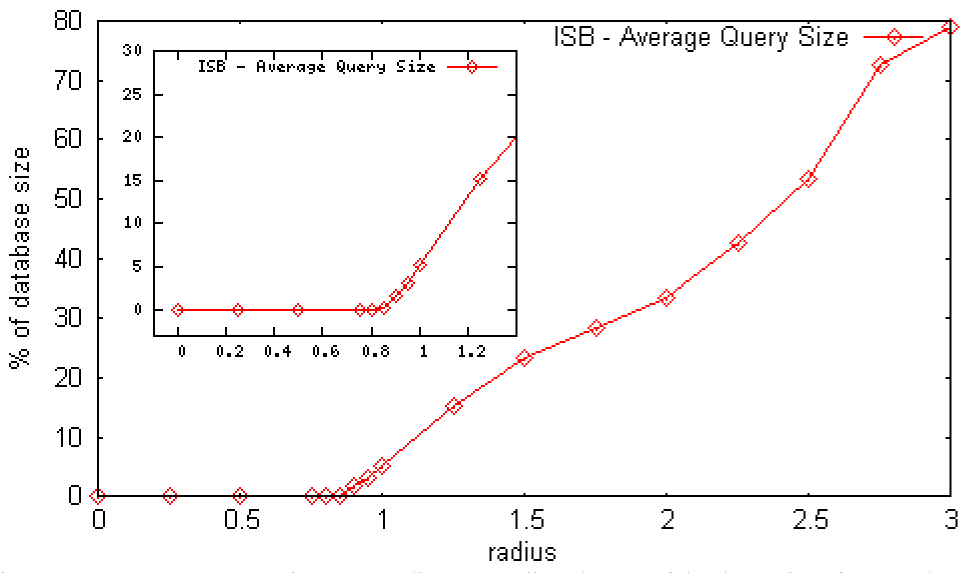


Figure 102: Average query size (query distance radius *d* vs % of database size) for ISB dataset