Finding Top-k Frequent Balls in High Dimensional Spaces

Liu Zheng

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Systems Engineering and Engineering Management

Supervised by

Prof. Jeffrey Xu Yu

©The Chinese University of Hong Kong
July 2004

中文摘要

在股票交易市场中，一个广泛使用的投资策略是参考股票过去的价格升跌模式。这种过去的价格升跌模式和当前的股票价格走势有很强的相似性。这种策略的根据是这种标准的价格模式往往会在未来重复出现。已经有很多学者专注于如何有效的监视，匹配这种价格模式，而本文则给出了如何在大量的时间序列数据中去识别出这种模式的几种有效的方法，以帮助用户去发现这些以前未知的价格模式。

本文所阐述的新问题是在大量的高维数据中发现 k 个有显著区分的球形簇。而这些球形簇的半径要小于预先给定的 r。我们把这些球形簇看成是更有意义，真正的频繁模式。

我们基于创新的树结构给出了两种新的搜索剪枝技巧，能够有效的增加算法的速度。同时，树结构也使得只输出最大球形簇成为可能。本文中的算法包括一个确定算法和几个启发式的近似算法。

在几个时间序列的数据集上的试验结果显示我们的方法能在高维空间中有效的，正确的发现真正的频繁模式。

Abstract of thesis entitled:
Finding Top-k Frequent Balls in High Dimensional Spaces
Submitted by Zheng Liu
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in July 2004

In stock markets, one of the widely-used investment strategies is to respond quickly when there is a trend that certain stock price shows strong similarity to one of the preselected price patterns. This strategy is based on the observation that similar patterns will repeat from time to time. While many papers concentrate on how to monitor stock price trends and how to find matches between stock price trends and preselected price patterns. We present approaches on how to find the most frequent patterns in large time-series data sets, or in other words in a high dimensional real-valued space in this dissertation. Our techniques help users identify previously unknown patterns to use as price patterns in stock price time-series.

The new problem we study attempts to find k distinctive balls that contain the largest numbers of data points in a given data set $\mathcal{S}$. The radius of a ball to be found needs to be less than or equal to a given radius r. These balls are true frequent patterns for their density is higher than ones generated by other existing algorithms.

We present two new pruning techniques. Novel tree-structures for indexing are introduced to speed up the computation. The first one reduces computational cost and the second removes those frequent balls, which are enclosed in other frequent balls. Our proposal includes an exact solution and two approximate solutions.

We also conducted performance study on several time series data sets. The experiment results show our approach gives true frequent patterns in a high-dimensional real-value space and can be processed efficiently.

Acknowledgement

There are thanks to many people, with the great favors of whom can I finish this dissertation in time.

To my supervisor Prof. Jeffrey Xu Yu, for both the academic guidance and the living attitude throughout my two-year-long study. He also confirm my belief to pursue a PhD for my ultimate goal of an academic career.

To Prof. Xuemin Lin and Prof. Hongjun Lu, for their kindly direction, support and discussion of what this research should be. They shared their bright thoughts with me which improved the work in the dissertation.

To the members of our database group, Fiona Choi and Gabriel Fung for their encouragement and helps. To Zhiheng Li and Wei Gao, for sharing their work with me. Special thanks to Jianghua Lv for the discussion of the work and proposition of some important issues.

To Many others, who deserve my thanks for various reasons. They are: Prof. Flip Korn in New York University, Prof. Eamonn Keogh in University of California at Riverside, Prof. Wei Lam and Prof. Helen Meng of our department. My friends: Xiaolei Yuan, Arber Xu, Qi Wang and Bin Zhang.

To my parents and Cora finally, who always support what I do.

Zheng Liu

The Chinese University of Hong Kong
July 2004

To my parents and Cora, for many wonderful things of the world.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

One of the widely-used investment strategies in stock markets is to respond quickly when a stock price trend shows a strong similarity to one of the preselected price patterns. This strategy is based on the observation that similar patterns repeat from time to time. There are reported studies concentrating on how to monitor the stock price trends and how to find matches between stock price trends and preselected price patterns. In this thesis, as general techniques, we present approaches on how to find the most frequent patterns in large time-series data sets, or in other words in a high dimensional real-valued space. Based on our solutions, we can determine real representative patterns for traders to monitor if they appear again. These patterns could help traders maximize their profits. In essence, this technique helps users identify previously unknown patterns.

Data mining is to discover previously unknown yet interesting patterns. There are three main data mining techniques, namely association rule mining, clustering and classification. The key step in association rule mining is to find frequent itemsets in a given transaction database with respect to a given support threshold, τ. Recall a transaction database is a set of transactions and each of them contains a set of items in a domain of items. A set of items is called an itemset. Clustering is to cluster data objects in a large data set into clusters such as the distances between objects within a cluster are as small as possible, whereas the distances between clusters are as large as possible. Classification is to build a classifier, based on a training data set, and to classify unseen data objects into classes.

The problem we study is similar to frequent itemsets mining problem in the sense that it attempts to find frequent patterns. But, unlike frequent itemsets mining, the frequent patterns to be found are patterns in a high-dimensional real-valued space. It is important to know that an itemset is a set of categorical data objects. A pattern in a real-valued m-dimensional space is a set of m-dimensional data points. In frequent itemsets mining, it counts itemsets based on whether they are the same or not. In our problem, two m-dimensional data points are seldom exactly the same. Data points in an m-dimensional data points are considered same if they are close to each other, by a given radius r. This makes the problem unique and challenging. The problem of finding frequent patterns by a given radius r is similar to clustering, which considers similarity among data to clusters. However, the former is to find a maximum space under the constraint of r with high density, whereas clustering is to partition data in the whole space into different clusters.

1.1 Contributions

In this dissertation, we study a new problem called top-k balls, denoted k-ball, in a given data set $\mathcal{S}$, by a user given radius r. The data set, $\mathcal{S}$, is a subset of the m-dimensional data space R^m. In brief, we attempt to find k distinctive balls that contain the largest numbers of data points in $\mathcal{S}$. The radius of a ball to be found needs to be smaller than or equal to the given radius r. The main contributions of our work are summarized below.

1. We observe that it is unsatisfactory to identify k-balls by enumerating every data point in $\mathcal{S}$ and treating it as a center in order to count data points in a ball of radius r. We propose a set of solutions to find such k-balls by considering possible

data points in the space R^m as centers. In other words, we will take a data point which may not be in $\mathcal{S}$ as a center to find frequent patterns. This would be challenging, as the search space is much larger.

2. The set of our solutions includes an exact solution and several approximate solutions. We present two new pruning techniques. Two novel tree-structures for indexing is introduced for this purpose. The first one reduces computational cost making the problem solvable in reasonable time. The second removes those frequent balls that are enclosed in other frequent balls making the collection of frequent balls compact and complete.

3. We conducted performance study. Our performance study showed that our approach gives the true frequent patterns in a high-dimensional real-value space and can be processed efficiently.

1.2 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, we will give an overview of the current methods of pattern discovery and analyze some previous related work also with the formal problem statement. Chapter 3 presents our ball discovery algorithm step by step from small data set to large data set, from exact algorithm to approximate heuristic greedy algorithms. Experiment results are shown in Chapter 4 and Chapter 5 discusses some important issues. The dissertation ends at Chapter 6 with conclusion and future directions.

□ **End of chapter.**

Chapter 2

Problem Statement and Background Study

Summary

In this chapter, we first present the formal statement of the k-ball problem and give an overview of current methods of pattern discovery followed by the analysis of some previous related work.

2.1 Problem Statement

In this paper, we study the problem of finding top-k frequent balls in a high dimensional real-valued space $\mathcal{S}$ with a user given radius, r. Here, a frequent ball is a maximal ball which has a radius smaller than or equal to r and contains as many data points as possible in $\mathcal{S}$. We call it the problem of top-k balls, denoted k-balls.

Distance (similarity) measure in the Euclidean space is Euclidean distance measure. For two points $s_i = (s_{i1}, ...s_{im}), s_j = (s_{j1}, s_{jm})$ in an m-dimensional space R^m. The Euclidean distance between the two points is defined as follows.

$$d(s_i, s_j) = \sqrt{\sum_{k=1}^{m}(s_{ik} - s_{jk})^2}.$$

Having defined the similarity of two points, we give the definition of *Ball*.

Definition 1. *(**Ball**) Given a set of m-dimensional data points $\mathcal{S}$ ($\subset R^m$) and a radius r of real number. A ball, in terms of $c \in R^m$, is denoted as $Ball(c) = \{s_j \mid d(c, s_j) \leq r \wedge s_j \in \mathcal{S}\}$. Here, c is the center of $Ball(c)$ and $Ball(c) \subseteq \mathcal{S}$ for any $c \in R^m$.*

Let B be a ball, we use $cent(B)$, $radius(B)$ and $boundary(B)$ to denote its center point c, radius and the set of points on the spherical surface of the ball respectively. Apparently $radius(B) \leq r$, the given radius.

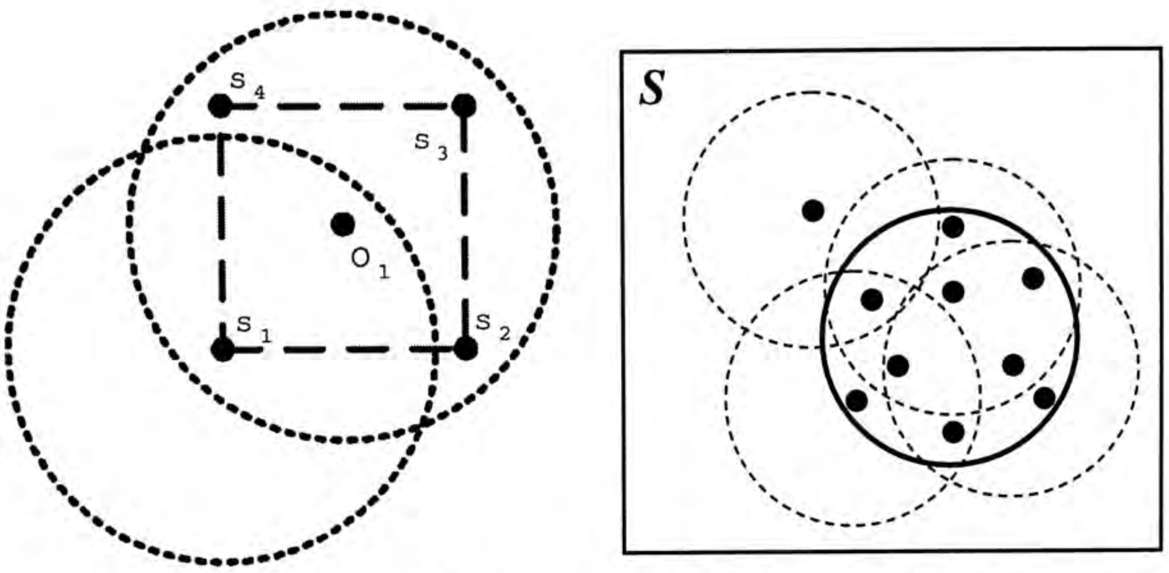
Definition 2. *(**k-ball**) Given a set of m-dimensional data points $\mathcal{S}$ ($\subset R^m$) and a range r of real number. The 1-st ball in S is the ball B_1 that has the highest count of all belonging points. The k-th ball in S is the ball that has the highest count of all belonging points and satisfies $d(cent(B_k), cent(B_i)) > 2r$, for all $1 \leq i < k$.*

The problem of top-k frequent balls (k-Ball) is to find k balls $Ball(c_1)$, $Ball(c_2)$, $\cdots$, $Ball(c_k)$, such that $radius(Ball(c_i)) \leq r$, for $Ball(c_i) \not\subseteq Ball(c_j)$ if $i \neq j$ and $|Ball(c_i)| \geq |Ball(c_j)|$ if $i > j$. Here, $c_i \in R^m$, for $i = 1, 2, \cdots, k$.

It is important to note that $Ball(c)$ is defined for a point c which may not be in $\mathcal{S}$. The main differences between finding $Ball(c)$ for $c \in \mathcal{S}$ and finding $Ball(c)$ for $c \in R^m$ but $c \notin \mathcal{S}$ are illustrated below.

Figure 2.1(a) shows an example where $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$. Assume that the distances, $d(s_1, s_2)$, $d(s_2, s_3)$, $d(s_3, s_4)$ and $d(s_4, s_1)$, are the same, $r + \epsilon$, for a very small $\epsilon > 0$. There does not exist a ball, $Ball(s_i)$, for any $s_i \in \mathcal{S}$, which contains more than one point (itself). Given $c \notin \mathcal{S}$ as the center of the four points. Obviously, $d(c, s_i) < r$ for $s_i \in \mathcal{S}$ and $|Ball(c)| = 4$. Such a $Ball(c)$ is the most frequent ball to be found.

Figure 2.1(b) illustrates another example on a simple set in R^m projected into 2-*d* space. In this example, there are 10 points in the



(a) There is no ball found if c in $\mathcal{S}$ (b) The quality of the most frequent balls

Figure 2.1: Difference between finding $Ball(c)$ for $c \in \mathcal{S}$ and $c \in R^m$

set $\mathcal{S}$. If we attempt to find $Ball(s_i)$ by enumerating points $s_i \in \mathcal{S}$, the most frequent ball to be found includes 6 points. The circles of dotted line are some examples of the above balls. But if we attempt to find a $Ball(c)$ for a c which is possible not in $\mathcal{S}$, the most frequent ball to be found includes 9 points as shown of circle of the solid line. Now we can see that the count of a ball whose center is an exist point is often smaller than the count of a ball whose center is in R^m. More results will be shown in the experiments section.

As mentioned above, the difficulty of finding the frequent balls, for $Ball(c)$, also comes from the fact that c does not necessarily be an existing point in the given set of points, $\mathcal{S}$ and the search space could be very large. We will present some efficient algorithms in the next chapter.

2.2 Background Study

In this section, we will first give an overview on related methods of pattern discovery, and then analyze in detail several cases of previous work in different applications.

2.2.1 Overview of Pattern Discovery Methods

There are two common ways to identify those previously unknown frequent occurring patterns in a m-dimensional Euclidean space R^m using either a k as the number of nearest neighbors or clusters or a r as a radius of the patterns. Both k and r are previously defined. Some of the existing works are summarized below. Let $\mathcal{S}$ be a set of points in R^m.

- k-approaches
  Clustering is a widely used method to separate the data into distinct groups that are often treated as patterns. Following are some partitioning clustering methods that are based on a previously defined number k.

  1) The k-Nearest Neighbor (k-NN) clustering problem is to find the k nearest neighbors, for an existing point, in $\mathcal{S}$. Here, k is the number of nearest neighbors. The goal of this clustering method is to simply separate the data based on the assumed similarities between various clusters. Thus, the clusters can be differentiated from one another by searching for similarities between the data provided. k-NN clustering is a discriminant clustering method to discover the most distinct clusters in the given $\mathcal{S}$.

  2) The goal of K-Median clustering, like k-NN clustering, is to separate the data into distinct groups based on the differences in the data. Here, k is the number of clusters to be partitioned. Thus, upon completion, the analyst will be left with k-distinct clusters with distinctive characteristics. The metric k-median problem is to select most k data points in $\mathcal{S}$ to be cluster centers initially, and repeat assign non-selected data points to their closest cluster centers until no data points will be reassigned any more.

  3) The k-center clustering problem is the min-max analogue of the k-median problem, while k-median which cannot obtain

the globe optimal is a greedy solution for the k-center problem. k is also the number of clusters. k-center clustering minimizes the size of the clusters where the *size* is defined to be the maximum distance between the center of a cluster and a point in the cluster. k-center is a *NP-Complete* problem[1] and have some approximation algorithms [1, 8, 16].

- r-approaches:

  1) The $motif$ discovery problem for time-series data is to find previously unknown frequent occurring subsequences. The straightforward method is to compute the number of points (subsequences) within a radius r of every existing data point (subsequences) and choose the subsequence with the largest number as the first significant pattern [5, 19, 22, 24].

  2) The dual clustering problem tries to cover each point in R^m with a unit ball and minimize the total number of balls used [4]. This is also a *NP-Complete* problem. [4] give a performance ratio of the problem.

  3)The r-dominating set problem is much like dual clustering which covers the points in R^m with balls whose radius is r and tries to minimize the number of centers to cover all data points. [2, 20] study this problem in a $2d$ plane.

- k/r-approaches:
  In density-based clustering, both k and r are used to identify clusters that have a range smaller than or equal to r and more data points than k [9].

All of the above methods use existing data points in the data set as a center of clusters to measure the radius or size of a cluster, except in the dual clustering and r-dominating problems. Based on

[1]*NP-Complete* problem is problem the answer of which cannot be computed and validated in polynomial time.

the definition of *ball*, we adopt a generalized r approach where the center of a cluster called ball may not be in the existing data set.

2.2.2 Applications

Some previous related work of pattern discovery in several different applications are analyzed here which can be regarded as the motivation of our work.

- Time-series clustering has been extensively studied. Clustering time series subsequences is an important subroutine in many other time series algorithms including association rule discovery, indexing, classification, prediction and anomaly detection etc.. In [18], Keogh et al. pointed out an important issue that clustering of time series subsequences is meaningless. Their claim is based on such fact that a data point, at the relative position i, a window starting at the same position, and will appear in $i - j$-th value in the following j-th sliding window. The mean of all such data points will be an approximately constant vector, which makes any clustering approach meaningless.

  Several papers studied *motifs* [5, 19, 22] which can be treated as one way to solve the above problem by clustering just the motifs which assume that a time series is mainly made of motifs. On the other hand, a motif itself can be considered as a substitute of sequence clustering method in many time series algorithms. A motif is a previously unknown frequent occurring subsequence after the continuous adjacent similar subsequences, called trivial matches, are removed. Let T_i be a subsequence starting at position i. T_i is a trivial match to T_j of a given radius r, if $T_i = T_j$ or there does not exist T_k such as the distance between T_k and T_j is greater than r and T_k is in between T_i and T_j. Figure 2.2 gives an example of a motif, which only appears 3 times (trivial matches have been removed) in the short series. The first motif is the most frequent subsequence in a given subset of m-dimensional subspace. Here, a subsequence of length m is mapped to a data point in an m-dimensional space. It can be considered as an application of k-balls that will find balls that include all possible trivial matches for a given radius r. [2]

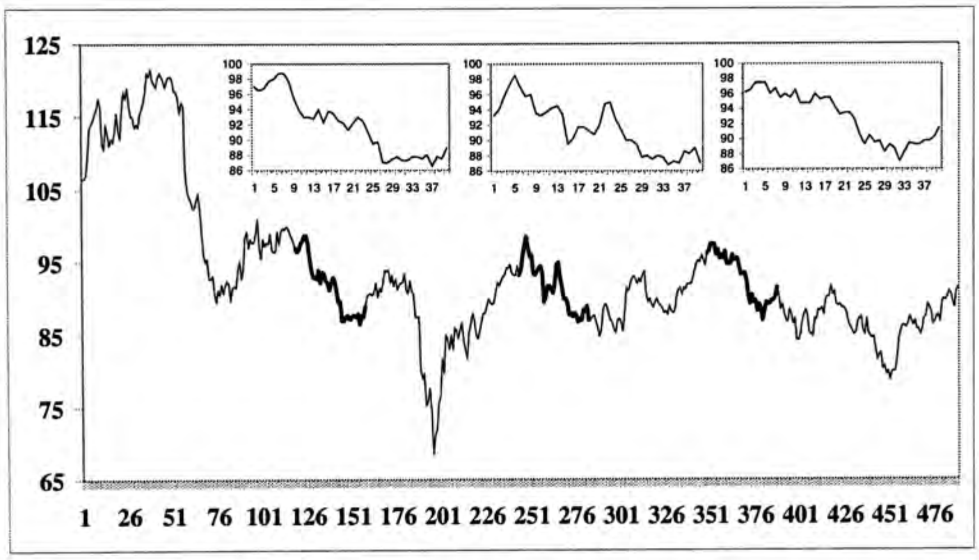


Figure 2.2: A motif in a time-series data

- Identifying preselected time-series (stock prices) patterns is an important issue, because one important scenario is that there are needs to respond immediately when the stock price trend shows a strong similarity to one of the preselected price patterns (a time-series) [11]. The centers of a top-k balls are good candidates for the representative price patterns if time-series data of length m are treated as data points in an m-dimensional space, though the finial patterns are often decided by domain experts.
- Allocating network centers in a distributed network such as mobile stations studied in [2] is a simplified k-ball problem

[2]One can always remove trivial-match subsequences by removing the continuous subsequences but one in the same motif.

in a $2d$ plane with the constrain that the center of radius r has an upper bound of total clients.

- There are many similar applications in sensor network, spatial database, etc.

The uniqueness of the k-ball problem is discussed below. First, the center of a k-ball is not necessarily an existing data point in $\mathcal{S}$ but could be any data point in R^m. Therefore, the search space becomes so huge that any trivial enumeration approach do not work for even a moderate data size. Note that when existing data points are the candidate centers, such enumeration approaches can enumerate all these existing data points to find patterns. The approach taken in k-means clustering could not be used in k-balls. That is to compute the mean of data points in a candidate set and use the mean as the center for iterations. You need to set k as the total number of possible patterns and choose the initial points very well. It is important to note that the problem of k-balls is a problem in a real-valued m-dimensional space. Second, k-balls shares similarity with data clustering problem. But, k-balls is not a data clustering problem. The data clustering problem is to cluster $\mathcal{S}$ into clusters to minimum the distances between data points in a cluster and maximize the distances between clusters. The emphasis of the data clustering problem is to partition the whole set of data points. But, k-balls is to find k subsets of the data points that with largest numbers of data points in a radius slight smaller than or equal to the given range r. Third, as a r-approach, it differs from other r-approaches. In the dual clustering problem [4] and r-dominating set problem [2, 20], the radius r is fixed, the number of balls needs to be minimized while covering all the points. In k-balls, the number of balls is given, the radius r' of a ball needs to be maximized within the constraint of the given radius r[3]. Figure 2.3 gives an intuitional example of k-balls that we are trying to find.

[3]The exact algorithm of this paper tries to find all possible balls which may share some common points, but one can easily generate the top-k balls by selecting largest ball and removing the sharing points in other balls.

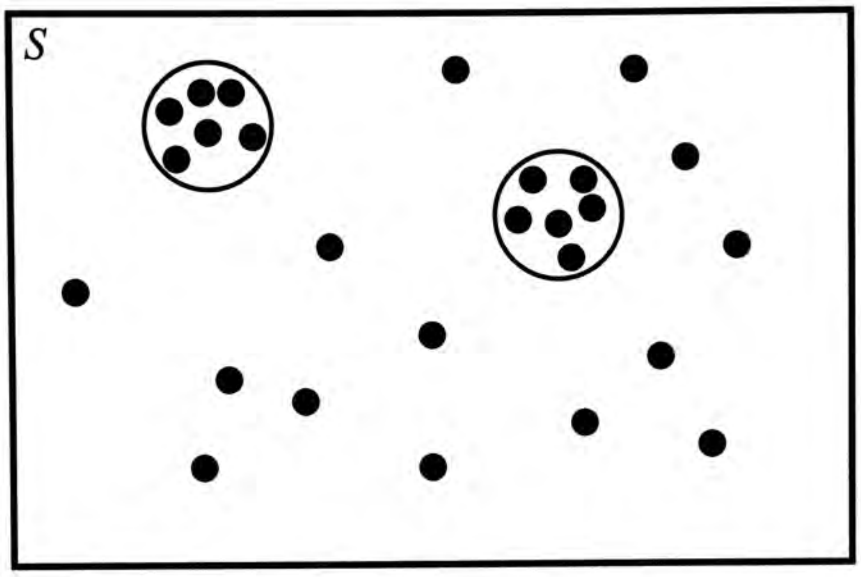


Figure 2.3: Example of balls to be found

□ End of chapter.

Chapter 3

Ball Discovery Algorithms

Summary

In this chapter, we will introduce our algorithms for ball discovery step by step from small data set to large data set, from exact algorithm to approximate heuristic greedy algorithms.

3.1 Brute-force Method for Ball Discovery

The problem of finding top-k balls is that given a set of m-dimensional data points $\mathcal{S}$ ($\subset R^m$) and a range r of real number, finding k balls $Ball(c_1)$, $Ball(c_2)$, $\cdots$, $Ball(c_k)$, such as $radius(Ball(c_i)) \leq r$, for $Ball(c_i) \not\subseteq Ball(c_j)$ if $i \neq j$ and $|Ball(c_i)| \geq |Ball(c_j)|$ if $i > j$. Here, $c_i \in R^m$, for $i = 1, 2, \cdots, k$.

Now, we will introduce a brute-force iterative Algorithm 1 for finding exactly all balls with a given point set $\mathcal{S}$ in R^m and a radius r. Once all balls are found, then the top-k balls can easily be obtained.

The function *miniball* here is to compute the smallest enclosed ball of a given point set. The smallest enclosed ball is the ball that contains the given point set in m-dimensional Euclidean space when

Algorithm 1 *Find-All-Ball-Brute-Force*$(\mathcal{S}, r)$

```
1: $miniball(\mathcal{S})$;
2: $r' = radius(\mathcal{S})$
3: let $P = boundary(\mathcal{S})$ be a set of points on the spherical surface of the small-
   est enclosed ball with a radius $r'$;
4: if $r' \leq r$ then
5:     output $\mathcal{S}$;
6: else
7:     for each $s_i \in P$ do
8:         $\mathcal{S}' = \mathcal{S} \setminus \{s_i\}$;
9:         Find-All-Ball-Brute-Force$(\mathcal{S}', r)$;
10:    end for
11: end if
```

the radius is smallest. The algorithm to compute a smallest enclosed ball is first introduced by Welzl [26]. Welzl adopts a move-to-front strategy to speed the computation and Berne Gätner implemented a slightly modified algorithm in C++ based on his paper [12]. We skip the detailed procedure of the smallest enclosed ball algorithm here for brevity which could be found at Appendix A.1.1. $miniball$ returns three values, the center point of the smallest enclosed ball, the radius of the smallest enclosed ball and the point set P at the spherical surface of the smallest enclosed ball. In Line 1 of Algorithm 1, $miniball$ is called, so we can get the radius r' and the boundary point set P of S at Line 2 and 3 separately.

This brute-force algorithm iteratively checks whether the radius of the smallest enclosed ball of the current point set is less than or equal to r and outputs the current set of points as a *ball* if it is true. Otherwise it will remove one of the points at the spherical surface of the smallest enclosed ball and check again. By removing one point s_i in P, $radius(\mathcal{S} \setminus \{s_i\})$ must be smaller than $radius(\mathcal{S})$, so the iteration would stop at the end. Here, $radius(\mathcal{S})$ denotes the radius of the smallest enclosed ball of $\mathcal{S}$.

Such an algorithm is extremely slow for the function $miniball$[1] is

[1]The algorithm of computing smallest enclosed ball is based on linear programming, and thus,

repeatedly called for many times and also the search space is large.

3.2 Ball Discovery with Small Point Sets

What we will discuss in this section is to find balls in a given point set which is small. Here, the small set $\mathcal{S}$ means that radius of $\mathcal{S}$ is just slightly larger than the previous defined range r, that is $r + \epsilon$. Apparently that this assumption is not held for most data sets of ball discovery problem and we will see in Section 3.3 that once the problem of finding balls in small sets is solved, the problem of finding balls in large data set is solved. The radius of a large point set is much large than the previously defined range r.

One may claim that the brute-force Algorithm 1 could work well if there are not many points in the data set. But the fact is that the brute-force algorithm cannot work well even with a few points. In the next two subsection, we will see how to prune the search space efficiently and collect the balls found in a compact and complete form.

3.2.1 Pruning the Search Space Using RP-tree

From Algorithm 1 we find that there are a lot of duplicated computations in the iterative procedure. Such duplication should be avoided for $miniball$ is computationally intensive time-consuming.

Figure 3.1 gives an example. Let's simulate how Algorithm 1 works. Suppose we have a small point set of five points $\mathcal{S} = \{1, 2, 3, 4, 5\}$ and a range r. At first, we call $miniball(\mathcal{S})$ to find that $\mathcal{S}$ is not a ball and the boundary point set is $\{1, 2, 3\}$. In iteration 1, we remove point 1 first, and still find the radius of the smallest enclosed ball of $S_1 = \{2, 3, 4, 5\}$ is larger than r that means S_1 is not a ball. Then come to the next iteration 1.1, we remove point 2. Suppose $S_2 = \{3, 4, 5\}$ is a ball, iteration 1.1 stops and the algorithm will

it is not very fast. For details, please refer to Appendix A.1.1.

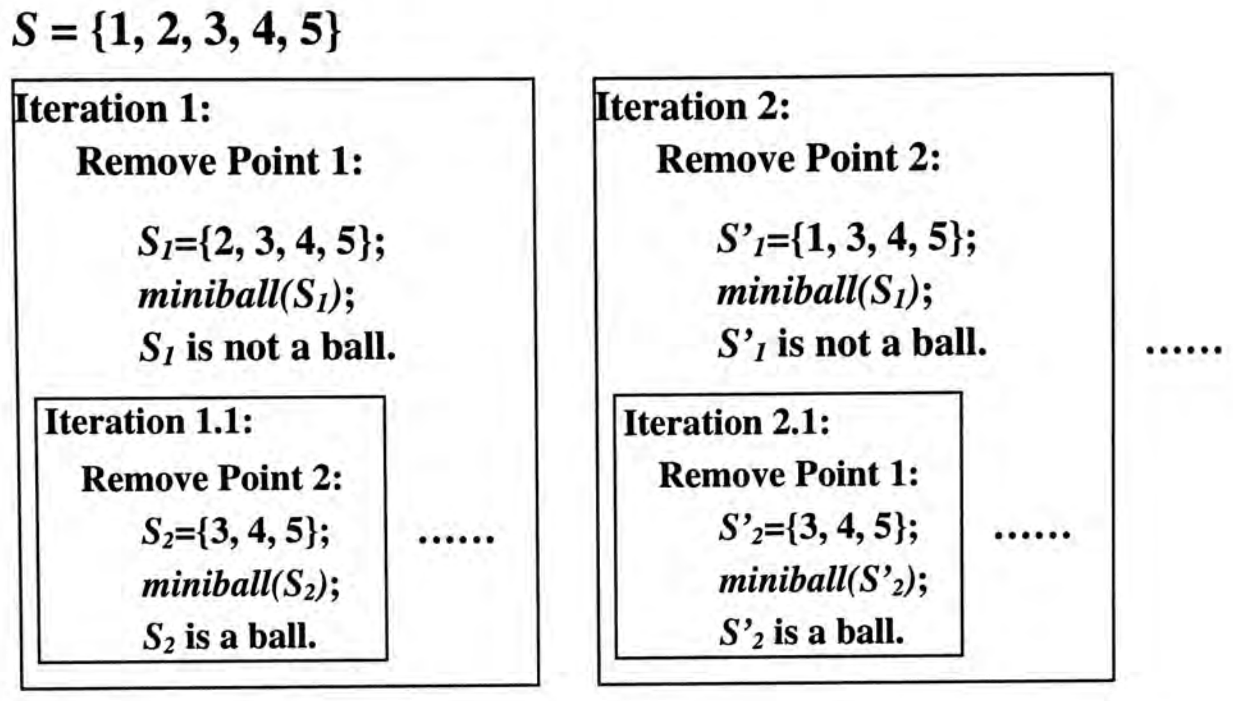


Figure 3.1: Duplicate Computation of the Algorithm 1

continue to the iteration 1.2,... . When the algorithm comes back to the top level iteration, it will continue the search in iteration 2. The algorithm removes point 2, and suppose we find $S'_1 = \{1, 3, 4, 5\}$ is not a ball. In the next iteration 2.1, the algorithm removes point 1, and finds that $S'_2 = \{3, 4, 5\}$ is a ball, so iteration 2.1 stops. Now we notice that the ball $S_2 = S'_2 = \{3, 4, 5\}$ has been found at the iteration 1.1, and there is no need to compute it again in the iteration 2.1.

Based on this observation, RP-tree (Removed Point Tree) is proposed to prune the unnecessary computation. Figure 3.2 shows a RP-tree.

RP-Tree is a prefix tree that stores the removed point sets in the iterations. The node of a RP-tree has two parts: the removed point at this node and the links to its children. The children are the points to be removed in the following iterations. A path of a RP-tree is a sequence of points encountered when going from the root of a RP-tree to a leaf. Suppose $(s_1, ..., s_n)$ is a path in the RP-tree from root

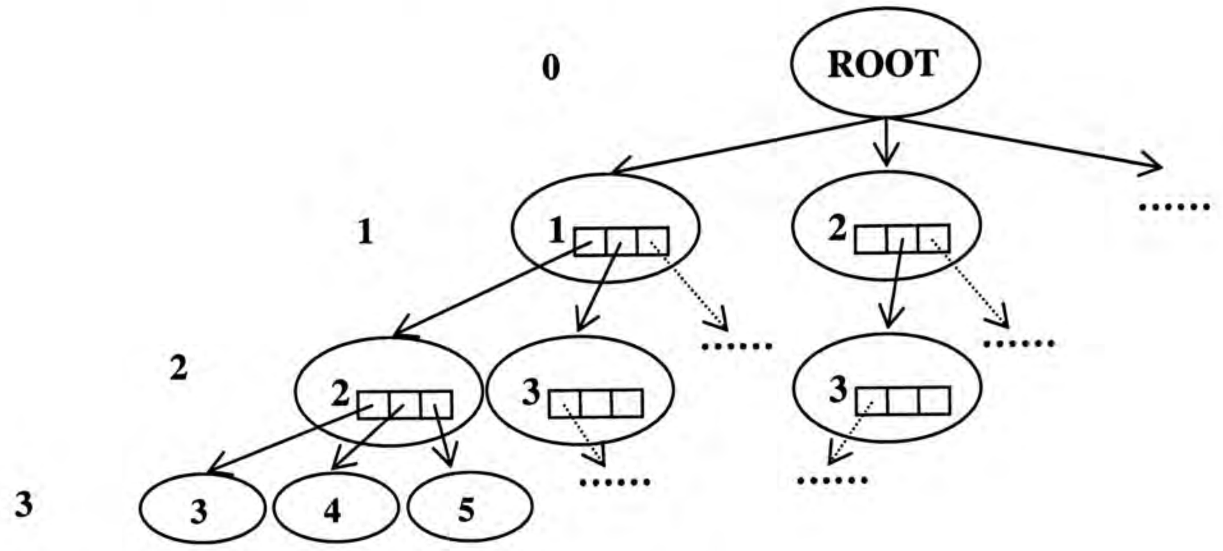


Figure 3.2: A RP-Tree

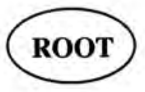
to the leaves, then a path point set P is the set of the points in the path, that is, $P = \{s_1, ..., s_n\}$. If the given point set is $\mathcal{S}$ and $\mathcal{S} \setminus P$ is a ball, for some P, we link the boundary points of the ball to the node containing the end point of the path, s_n.

With a RP-tree, calling $miniball(P)$, we check the RP-tree first to see whether the removed point set can form a prefix of a path in the tree. If so, we know that the point set P have been computed, then we directly go to the next iteration.

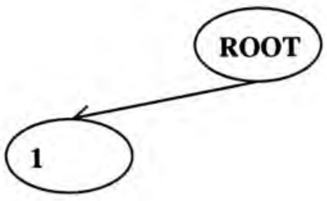
Let's re-visit the example in Figure 3.1. We will explain how to construct a RP-tree incrementally while searching balls as shown in Figure 3.3.

There are three point sets in Figure 3.3, point set P, boundary set B and removed set R. point set P is the current point set to be computed with $miniball()$, boundary set B is the boundary point set of P and removed set R is $\mathcal{S} \setminus P$, $\mathcal{S}$ is the given point set initially.

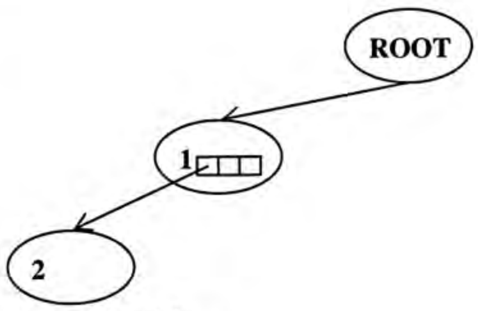
In iteration 0, the tree only consists of a *root* node. Since $P_0 = \{1, 2, 3, 4, 5\}$ is not a ball, we call $miniball(P_0)$ and get $B_0 = \{1, 2, 3\}$. In iteration 1, the removed set R_1 is $\{1\}$, we check the RP-tree to see whether the R_1 can be mapped to a prefix of any path in the RP-tree before calling the $miniball(P_0)$. Apparently the answer is negative



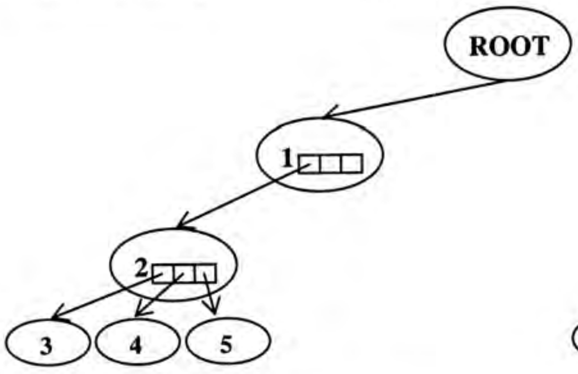
Iteration 0:
Point Set: {1, 2, 3, 4, 5}
Boundary Set: { 1, 2, 3}
Removed Set: { }



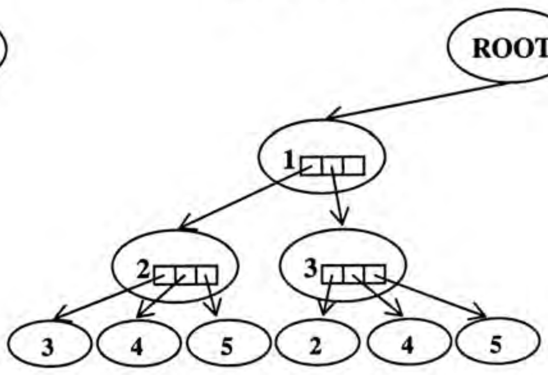
Iteration 1:
Point Set: {2, 3, 4, 5}
Boundary Set: {2, 3}
Removed Set: {1}



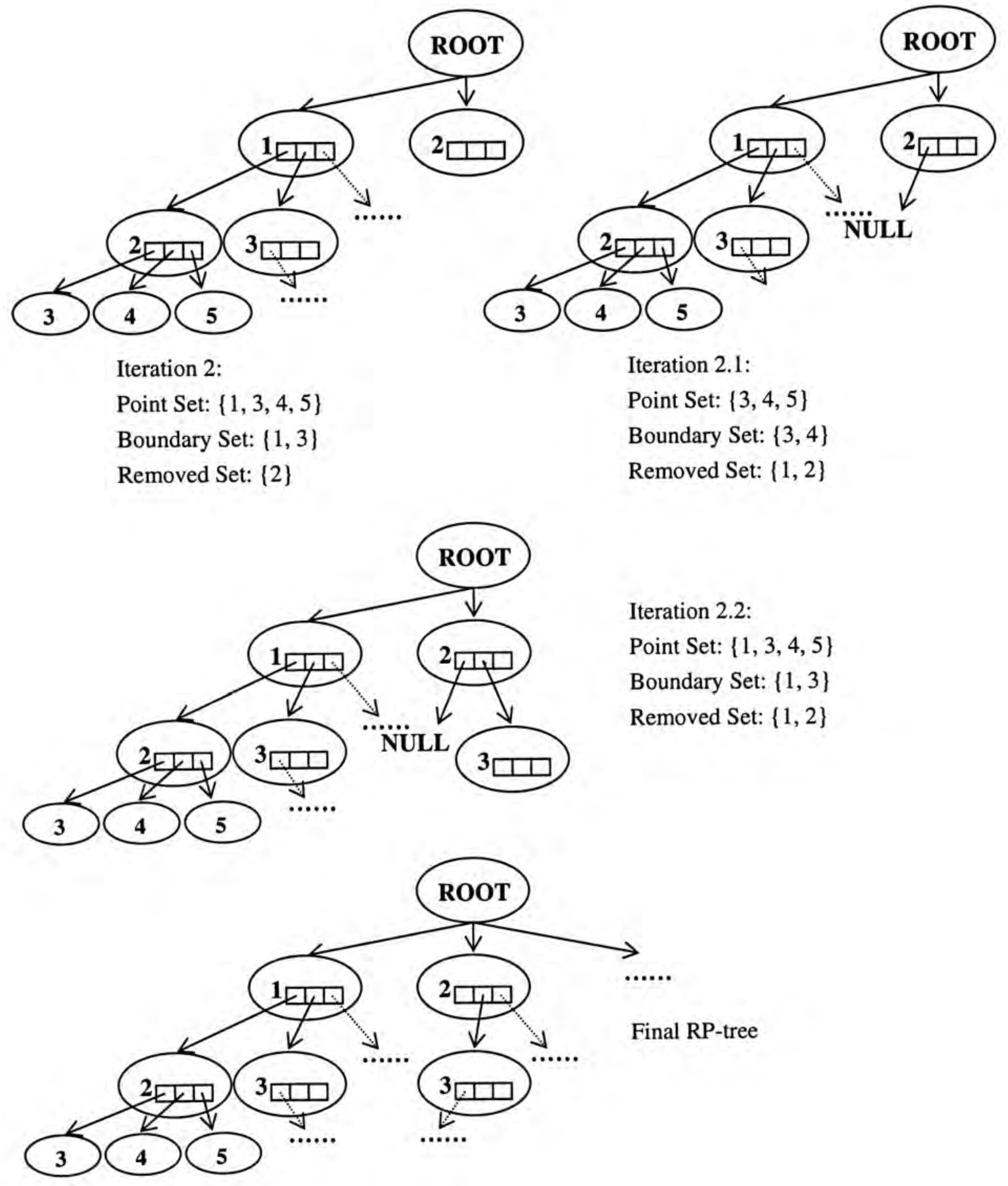
Iteration 1.1:
Point Set: {3, 4, 5}
Boundary Set: {3, 5}
Removed Set: {1, 2}



Link the Boundary Points Set.



Iteration 1.2:
Point Set: {2, 4, 5}
Boundary Set: {4, 5}
Removed Set: {1, 3}



Iteration 2:
Point Set: {1, 3, 4, 5}
Boundary Set: {1, 3}
Removed Set: {2}

Iteration 2.1:
Point Set: {3, 4, 5}
Boundary Set: {3, 4}
Removed Set: {1, 2}

Iteration 2.2:
Point Set: {1, 3, 4, 5}
Boundary Set: {1, 3}
Removed Set: {1, 2}

Figure 3.3: The Construction Procedure of a RP-tree

because RP-tree has only the root. We then add the removed set R_1 to the RP-tree and initiate iteration 1.1. In iteration 1.1, the removed set $R_{1.2}$ is $\{1, 2\}$ and it still cannot be mapped to a prefix of any path in the RP-tree. We then add the removed set $R_{1.2}$ to the tree too. We do the same in other iterations at this level, i.e. iterations 1.2, 1.3, etc.. In iteration 2, the removed point set $R_2 = \{2\}$. We add it to the tree and initiate iteration 2.1, where the removed list $R_{2.1}$ is $\{2, 1\}$. We find that it can be mapped to the prefix of the path $(1, 2)$, so we know that the point set $\mathcal{S} \setminus \{1, 2\}$ has been computed. We just skip this iteration and continue to iteration 2.2. Iterations continue until finish.

From the above explanation, we can see that RP-tree is built in an incremental way during the iterations. The algorithm for construction of a RP-tree is highly integrated with the entire ball discovery algorithm. We will present it later together with the ball discovery algorithm.

Now we outline an algorithm for determining whether a removed point set has already been computed using the RP-tree. The key idea is whether the current point set P can be mapped to a prefix of a certain path in the RP-tree. The parameter $root$ in Algorithm 2 is the root node of the RP-tree initially and R is the current removed set to be tested. $Foundflag$ is a boolean variable outside *Mapping-RP-tree* and the function will set it to $true$ if the removed point set can be mapped successfully, otherwise *false*.

Algorithm 2 directly maps the removed set to the prefix of a path of the RP-tree.

When the height of the RP-tree is not so low as shown in our simple example, the pruning will be efficient. Consider the example in Figure 3.4. A and B are two pointers pointing to some nodes. Assume in the current iteration, A is pointing to the right side branch and the current removed set is $\{3, 1\}$, which is equal to the set of $\{1, 3\}$ as a prefix from the $root$ to B. With RP-tree, the large search space starts from B is pruned.

Algorithm 2 *Mapping-RP-tree*$(R, root)$

```
1: if $R$ is $\emptyset$ then
2:     $Foundflag = true$;
3: end if
4: if $(root! = NULL)$ and $(Foundflag == false)$ then
5:     for each child $s_i$ of $root$ do
6:         if $s_i$ in $R$ then
7:             Mapping-RP-tree$(R \setminus \{s_i\}, root = root \to s_i)$
8:         end if
9:     end for
10: end if
```

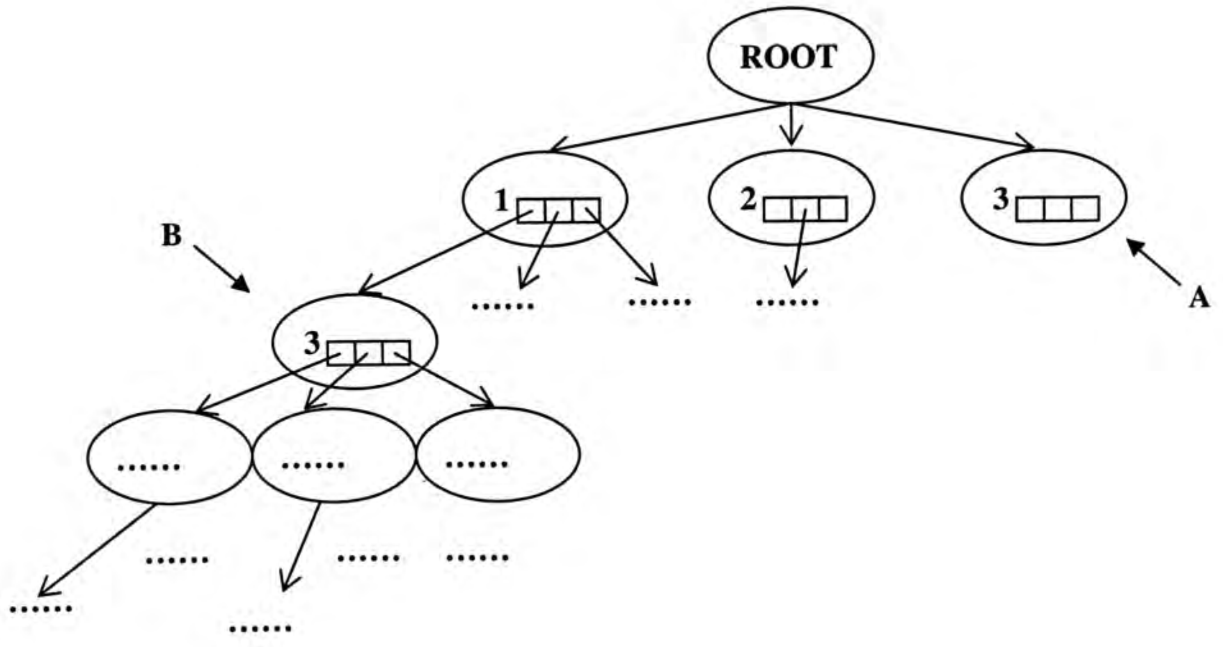


Figure 3.4: Pruning Ability of RP-tree

The Small point set
1, 2, 3, 4, 5, 6, 7, 8

Order	Balls
1	1 3 4
2	1 7
3	1 2 7
4	1 2 4
5	1 2 7 8
6	1 2 6 7
7	1 4 6
8	1 2 6 7 8
9	1 2 5 6 7 8
10	1 5
11	1 5 6
12	1 2 5 6

Table 3.1: An Output List of Balls

3.2.2 CB-tree - Collection of Balls in a Compact and Complete Form

Table 3.1 is a list of balls found in some small point set $\mathcal{S}$ by Algorithm 1 with the RP-tree pruning before the computation of *miniball*. The small point set is part of a real data set and the detailed coordinates of the points is given in Appendix A.2.

We carefully study the output list of balls, and come up with the following observations:

1. Point 1 is always at the beginning of a ball. We currently skip it and will talk about the reason in the next section.

2. If we store all balls found in some compact structure, it would be easier to determine the top-k balls.

3. Some balls found in the list are subsets of other balls. The structure for collecting balls should avoid such redundancy.

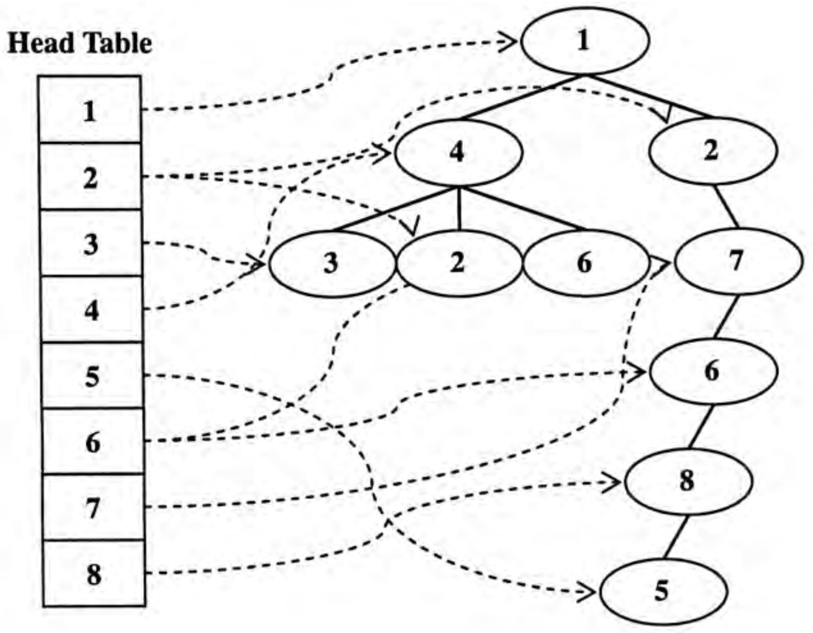


Figure 3.5: A CB-Tree

4. If two ball sets share a common prefix according to the output order, then the share parts can be merged using one prefix structure.

Recall the definition of *ball*, what we expect to find is *maximal ball*, i.e., every ball found should not be included by other balls. In the above output list, because the order of removed points is different, sometimes we get balls which are not maximal balls.

With these observations, we propose the CB-tree (Collecting Balls Tree), which is used to store balls in a complete and compact form as shown in Figure 3.5. CB-tree can also serve as a pruning tree in the ball discovery algorithm.

CB-Tree is a prefix, bi-direction linked tree which is used to store the balls. The node of a CB-tree contains three part: the point at this node, the links to its children and the link to its parent. One node could have more than one child but only one parent. A path of a CB-tree is a sequence of points encountered when going from the root of a CB-tree to a leaf. Suppose $(s_1, ..., s_n)$ is a path in the RP-tree from root to the leaves, then a path point set P is the set of points in the path, that is, $P = \{s_1, ..., s_n\}$ and P is a maximal ball.

Each CB-tree is accompanied with a head table, whose cells are made of points and each point is associated with a list of links to the occurring position of the point in the CB-tree.

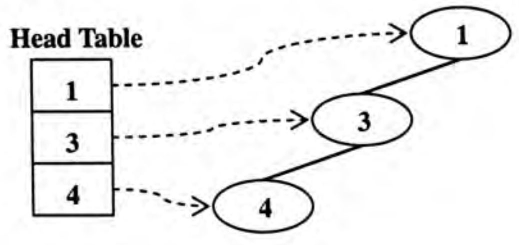
We will first introduce how to construct a CB-tree from the output list of balls in Table 3.1. There are two important operations on the CB-tree: $adjust$ and $closeset$. A step-by-step procedure for constructing a CB-tree based on Table 3.1 is shown in figure 3.6.

Initially, the CB-tree is *NULL*. In step 1, the ball B_1 is $\{1, 3, 4\}$, since the CB-tree is now empty we simply add ball 1 to the CB-tree by linking the points one by one then update the head table. Since point 1 occurs in every ball, we use it as the root of the CB-tree.

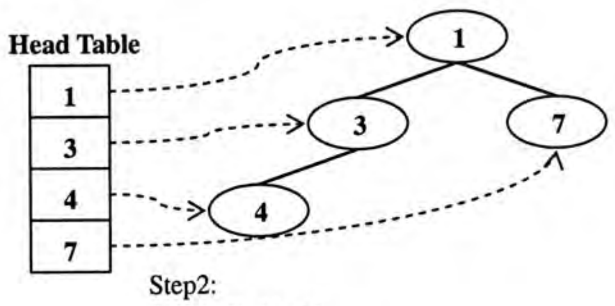
In step 2, the ball B_2 is $\{1, 7\}$. We search the tree to see if the ball is contained in any path point set from root to leaves. If so, that means the current ball is contained in some other balls which are already stored in the tree. Then we just skip the current ball and come to the next one. If not, We try to map the ball to the tree path to see whether a subset of the ball, assume P_2, can be mapped to a prefix of some path in the tree using depth-first strategy. Apparently we can see from Table 3.1, that a ball in the table is at least shared by the root point with any other ball in the tree. Ball 2 can only be partially mapped to the root point. Thus, we link the remaining of the ball $B_2 \setminus P_2$ one by one as the right most branch of the last common point in the prefix, i.e. $P_2 = \{1\}$ in this example. We do the same in step 3.

In step 4, the ball B_4 can only be linked to the $root$ point as the right most branch. But pay attention that the left most branch and the newly added branch at right most share some common point 4. To make the CB-tree as compact as possible, we need an operation called $adjust$ to compress the CB-tree. The algorithm of operation $adjust$ will be shown late, we now only give the result at step5.

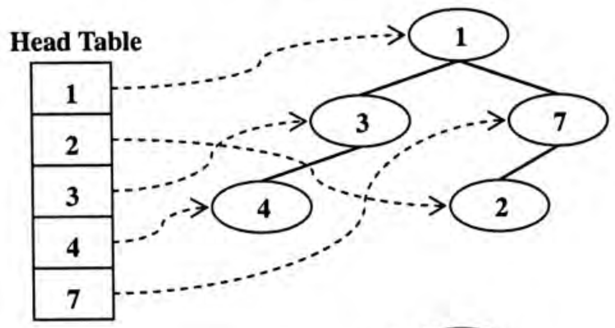
The remaining process is similar to step 2 to step 5.



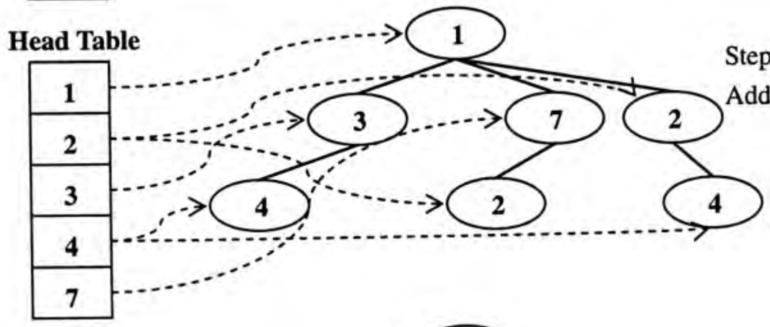
Step1:
Add ball: {1, 3, 4}



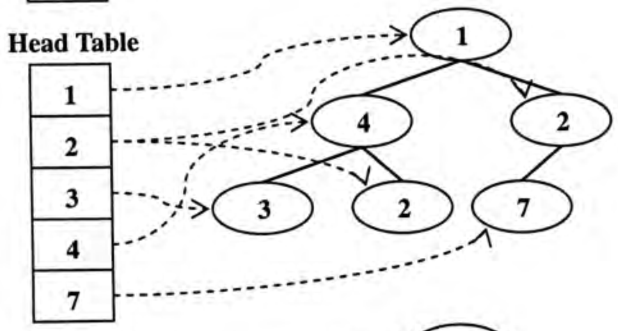
Step2:
Add ball: {1, 7}



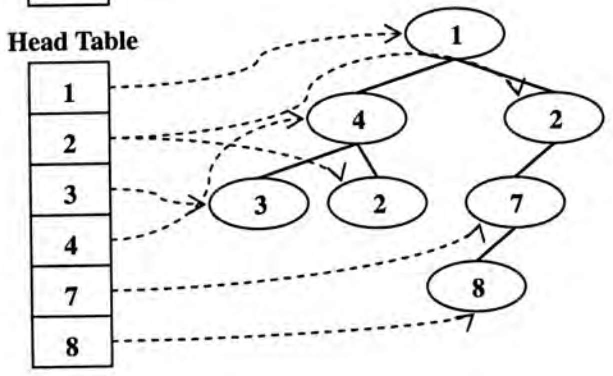
Step3:
Add ball: {1, 2, 7}



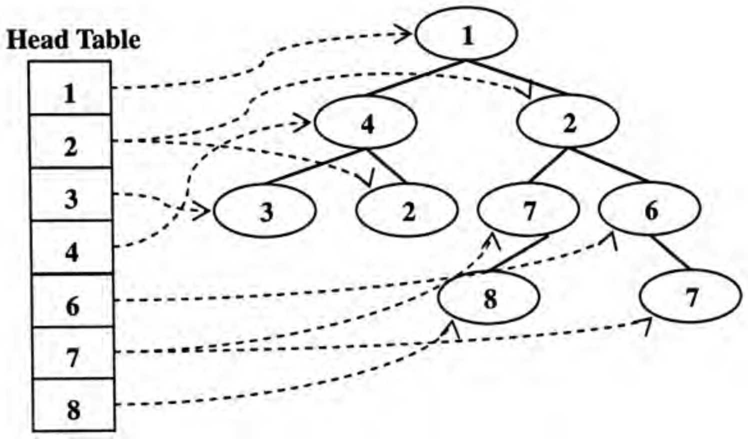
Step4:
Add ball: {1, 2, 4}



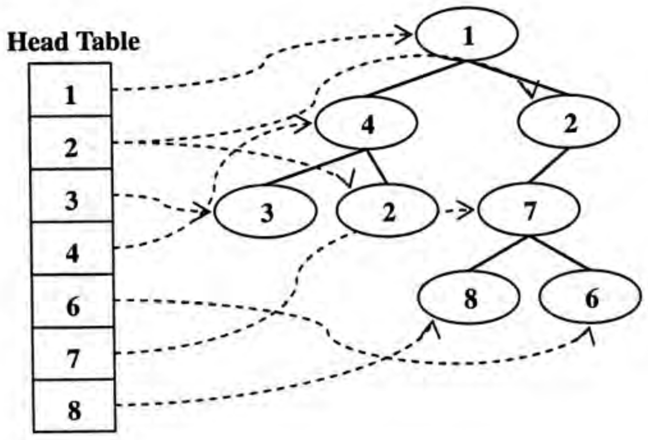
Step5:
Adjust Tree to Keep CB-tree Compact.



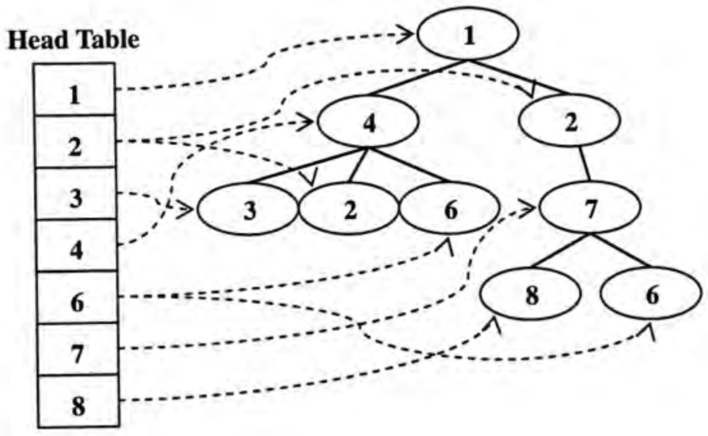
Step6:
Add ball: {1, 2, 7, 8}



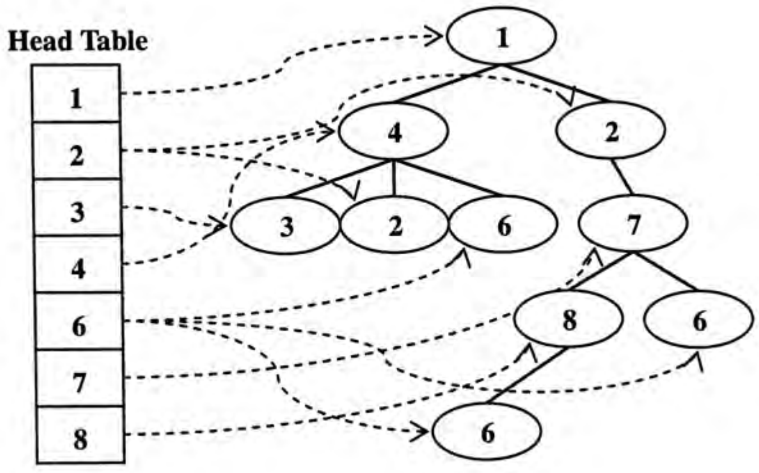
Step7:
Add ball: {1, 2, 6, 7}



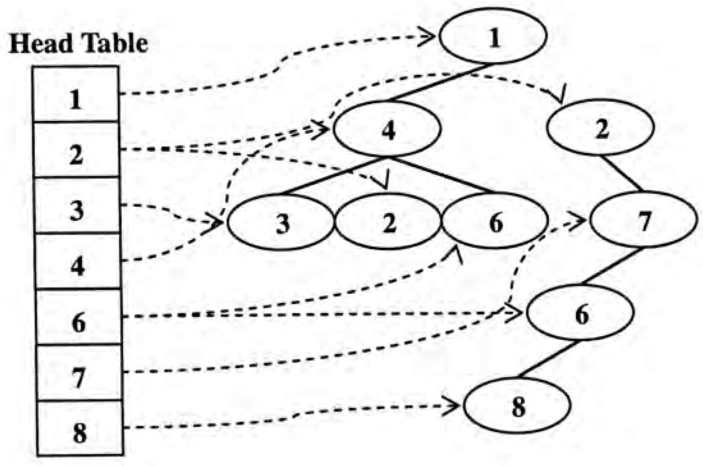
Step8:
Adjust Tree to Keep CB-tree Compact



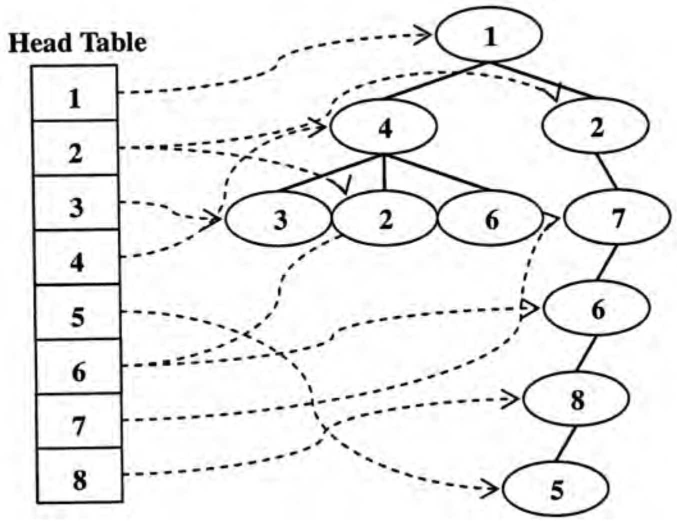
Step9:
Add ball: {1, 4, 6}



Step10:
Add ball: {1, 2, 6, 7, 8}



Step11:
Adjust Tree to Keep CB-tree Compact.



Step12:
Add ball: {1, 2, 5, 6, 7, 8}

Figure 3.6: The Construction Procedure of a CB-tree

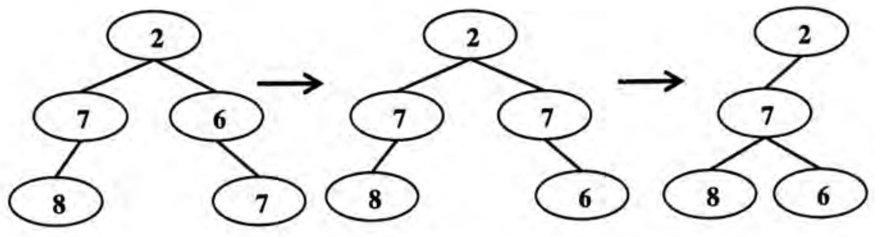


Figure 3.7: *Adjust* CB-tree

Adjust is an important operation to keep the CB-tree compact and complete. The procedure of *adjust* is shown in figure 3.7. Take step 7 in Figure 3.6 as an example. First, we trace towards the *root* of the CB-tree from the first point of the newly linked branch to find the first node s_i, the number of whose children is not 1. Here, s_i is the node with label 2[2]. We then find the first common point on the other children branches of s_i depth-first strategy. We then lift up the common point of both branches as a child of s_i and unite the children of the common point in both branches. This procedure is repeated from the right most branch of the common point until there is no common point shared by the two branches.

The algorithm is shown below. Here the initial input parameter in this example is s_i. The root is assumed to have n children.

In the above adjust algorithm, we choose the first common point using a depth-first strategy. One may claim that the CB-tree could be redundant. We discusses this using the example in Figure 3.8.

The current state of the CB-tree is shown at the top graph in the figure. Suppose the next coming ball is $\{1,2,4,7\}$, the ball will then be added to the position shown in the middle graph based on our algorithm. But notice that the right most branches of node 1 is $B_1 = \{1,2,7\}$ which is a subset of the path point set $\{1,4,2,7\}$. Thus B_1 should be removed. But the *adjust* algorithm cannot do this. We import a new operation called *closeset* to check if there are some branches which are the subset of other branches and remove

[2]We simply use the point contained by the node as the label.

Algorithm 3 *Adjust-CB-tree*$(root)$

```
 1: for (each child $s_k$ of root) and ($k! = n$) do
 2:     $pos = root \rightarrow s_k$;
 3:     if Branch $pos$ and $root \rightarrow s_n$ have common point $p$ then
 4:         $found1$ = pointer of $p$ on branch $pos$;
 5:         $found2$ = pointer of $p$ on branch $root \rightarrow s_k$;
 6:         break;
 7:     end if
 8: end for
 9: Heighten $found1$ to be a child of $root$;
10: Heighten $found2$ to be a child of $root$;
11: Link the children of $found2$ to $found1$, if any;
12: delete $found2$;
```

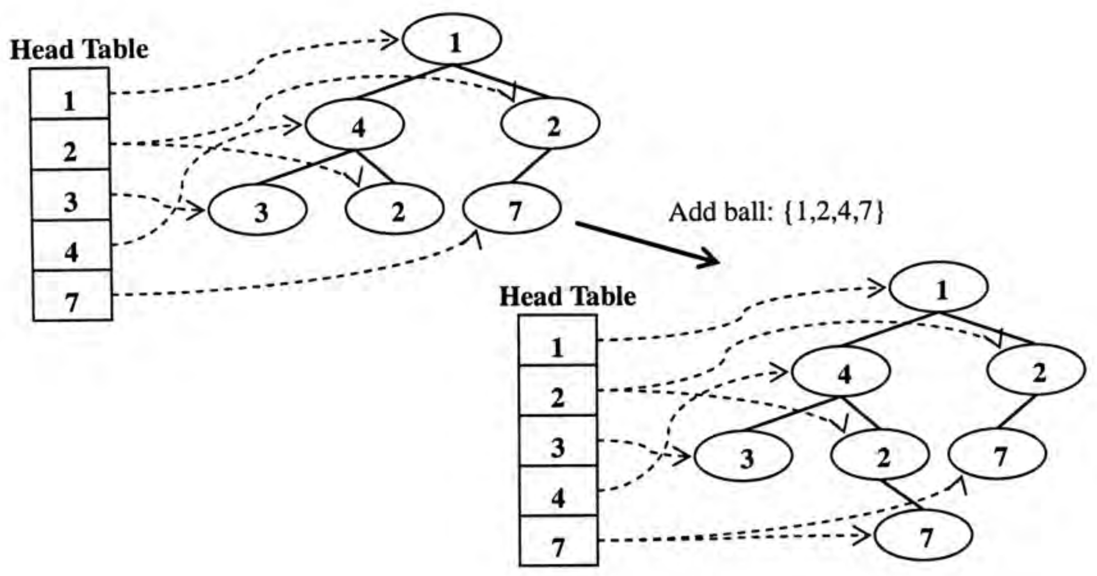


Figure 3.8: An example of the redundancy of a CB-tree

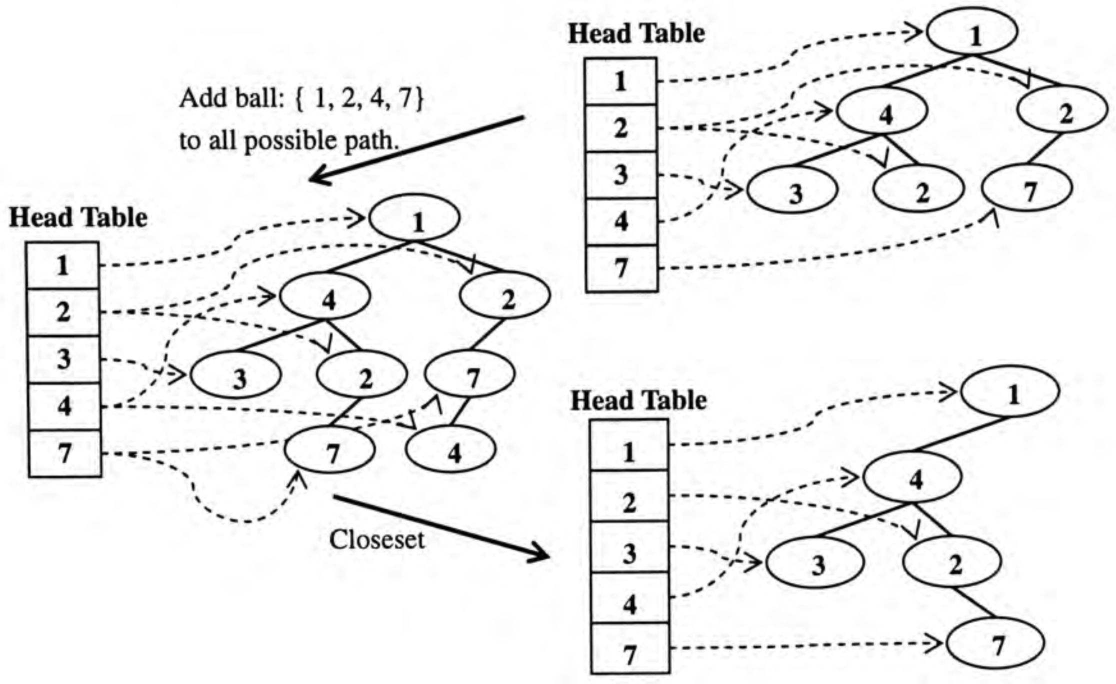


Figure 3.9: How to do *Closeset* to remove redundancy

them in time.

The reason why such kind of redundancy occur is that the ball $\{1, 2, 4, 7\}$ can be partial mapped to the path $(1, 4, 2)$ or $(1, 2, 7)$. But our algorithm simply chooses the first one encountered from left to right and ignore the other. Figure 3.9 shows the solution to this problem.

When adding the ball to the tree, instead of just adding the ball to the first mapping position, we add the ball to all possible positions and if the paths from root to leaves of the newly added point branches are the same, then we just keep one and remove the others. The procedure of building a CB-tree with the operation *closeset* is given in Algorithm 4. The parameter *root* is the root of the CB-tree and B is the current ball.

Algorithm 4 *Build-CB-tree-Closeset*$(root, B)$

```
1: $i = 0$;
2: for each possible mapping of part of prefix of a path do
3:     $i++$;
4:     Link the remaining part of $B$ to the last point of the prefix;
5:     $p_i$ = the bottom point of the newly added branch;
6: end for
7: for $k = 2$ to $i$ do
8:     if (path set from $p_k$ to $root$) == (path set from $p_1$ to $root$) then
9:         Delete the newly added branch ends at $p_k$;
10:    end if
11: end for
12: $Adjust$ the remain of the newly added branches
13: Repair head table;
```

3.2.3 Algorithm of Finding Balls Using RP-tree and CB-tree

Following, we present the whole algorithm for finding balls using RP-tree and CB-tree.

The function *search-CB-tree* is used to see whether the given data set is a ball. It is similar to the function *Mapping-RP-tree*. We skip it here for brevity.

3.3 Ball Discovery in Large Point Sets

We have developed the ball discovery algorithm for small point sets in the last section. We transfer our attention to large point sets. The radius of a large point set is much large than the previously defined range r. For large point set, the algorithm 5 in the section 3.2.3 will not work well due to the vary large search space.

3.3.1 Candidate Sets of Balls

In order to find all top-k frequent balls in large point sets, a better way is to generate some small candidate sets which are larger than the final balls then find the frequent balls in them. Such candidate

Algorithm 5 *Find-All-Ball-Small*$(\mathcal{S}, r)$

```
1: if search-CB-tree($\mathcal{S}$) then
2:     return;
3: end if
4: $miniball(\mathcal{S})$;
5: $r' = radius(\mathcal{S})$;
6: let $P = boundary(\mathcal{S})$ be a set of points on the spherical surface of the mini-
   mum enclosed ball with a radius $r'$;
7: if $r' \leq r$ then
8:     Link $boundary(\mathcal{S})$ to RP-tree;
9:     Build-CB-tree-Closeset($CBtree \rightarrow root, \mathcal{S}$);
10: else
11:     for $p \in P$ do
12:         if Search-RP-tree($P \setminus \{p\}, root$) then
13:             return;
14:         else
15:             Link point $p$ to the RP-tree;
16:             $Find - All - Ball - Small(P \setminus \{p\}, r)$;
17:         end if
18:     end for
19: end if
```

must be sufficiently small so that the ball discovery algorithm on small data set is applicable. In this subsection, we propose a way to identify the candidate sets. Once the candidate sets are generated, one can apply Algorithm 5 to find all frequent balls.

By the definition of a *ball*, if s_i and s_j are two points in the same ball of range r, they must satisfy the condition that $d(s_i, s_j) \leq 2r$. Based on that, we define candidate sets as follows:

Definition 3. Candidate Set *Given a set of m-dimensional data points $\mathcal{S}$ ($\subset R^m$) and a range r of real number. A candidate set, D_i, is a subset of the given set of m-dimensional data points $\mathcal{S}$ such that for s_j and s_k ($j \neq k$) in D, $d(s_j, s_k) \leq 2r$, where $|D| > 1$.*

Definition 4. Maximal Candidate Set *A candidate set D_i is maximal if there does not exist another candidate set D_j which contain D_i, i.e. $D_i \subset D_j$.*

We use the symbol D_i to refer to both a candidate set and a maximal candidate set, and from now on D_i is refer to a maximal candidate expect for special indication. We will find balls only in such maximal candidate sets, because if we find all maximal balls of a maximal candidate set, then maximal balls of any candidate subsets of the maximal candidate set will also be found. Recall that a *maximal ball* is a ball whose radius r' cannot be smaller than or equal to r if any data point $s_i \in \mathcal{S}$ is added to the ball.

There is a set of maximal candidate sets $\mathcal{D} = \{D_1, D_2, \dots\}$ in some $\mathcal{S}$ ($\subset R^m$) . Note that $D_i \cap D_j \neq \emptyset$ is possible. The top-k frequent balls can only be determined from these candidate sets, because there does not exist any of top-k balls that is not in a subset of certain maximal candidate set D_i.

Proof. $Ball(c) = \{s_1, ...s_n\} \Rightarrow Ball(c)$ satisfies the condition that for any $s_j, s_k \in Ball(c), j \neq k, d(s_j, s_k) \leq 2r \Rightarrow Ball(c) \subseteq D_i \subseteq D_j$, here D_i is a candidate set and D_j is a maximal candidate set. $\square$

But, it does not necessarily mean that a ball with a radius r can enclose all data points in D_i which has a diameter of $2r$. Figure 3.10 shows an example of 3 points, s_1, s_2 and s_3 in the 2-dimensional plane. The distance between any two of the three points in the figure is $2r$. Since the distance of any s_i to the center o of the smallest ball enclosing all three points is $(\sqrt{3} - \frac{1}{2})r$ which is greater than r, there is no ball with radius of r which can enclose all the 3 points.

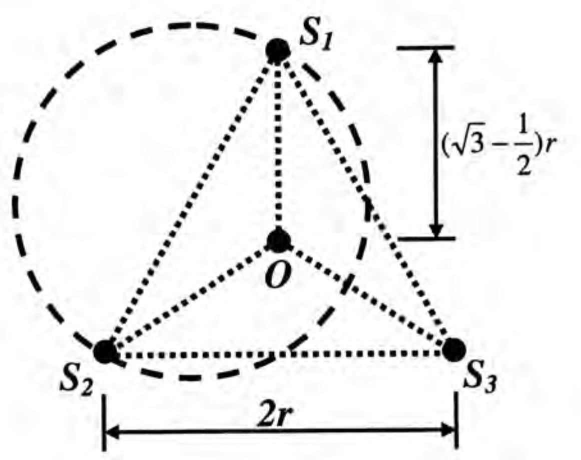


Figure 3.10: Relationship between a ball and a candidate set

The problem of finding candidate sets can be solved using the A-priori property. If there is a candidate subset D' in which there exist $d(s_i, s_j) > 2r$ for any $s_i, s_j \in D_i$, then any of its superset D" of D' (i.e. $\supset D'$) cannot be a candidate set. The problem here is to find all maximal candidate sets that are not included in any other candidate sets. In a similar way, maximum frequent itemsets in data mining can be found as follows.

First, let's build a transaction file. Each line of the file is a transaction T_i including a set of points, $\{s_j \mid d(s_j, s_i) \leq 2r\}$, for a given data point s_i. Suppose that there are n data points in S. And n such transactions in the file. Apparently, if the point s_i appears at the line of s_j, then s_j must appear at the line of s_i. Second, conduct frequent itemsets mining on a condition such that the absolute support of a frequent itemset is equal to the size (i.e. number of points) of the itemset, in order to ensure that all data points in the set are within $2r$ distance with any others. Such frequent items are just the maximal candidate set we want.

A major drawback of this solution is efficiency, especially when the set of $\mathcal{S}$ is dense. This is equivalent to a transaction containing too many items (points). All these items could not be pruned as much as expected in frequent itemsets mining where the support is large to prune most items.

In this paper, we propose a graph based solution. First, we construct an undirected node-label connected graph, $G(V, E)$, based on the given set of data points, $\mathcal{S}$. Here, $V = \mathcal{S}$, and $E \subseteq V \times V$ such that $(v_i, v_j) \in E$ if $d(v_i, v_j) \leq 2r$. The distance, $d(v_i, v_j)$, is the edge label of (v_i, v_j). Second, the candidate sets can be found by finding all cliques. Recall a *clique* in a graph is a maximal complete subgraph. We use the algorithm in Appendix A.1.2 to find all cliques of an undirected graph.

3.3.2 A Divide-and-Conquer Algorithm

Usually, we need not generate candidate sets from the complete point set because the algorithm of finding all cliques requires to construct a connected matrix of all vertices. If the size of the complete point set is large, such matrix would also be very large. Another reason is that the size of the candidate sets or the balls to be found are small though the size of the given point set S in R^m is large,

For any point $s_i \in \mathcal{S}$, the possible candidate sets or balls that include s_i must be a subset of the set $\{s_j | d(s_i, s_j) \leq 2r\}$. Based on this observation, we proposed the divide-and-conquer strategy (see Algorithm 6). Algorithm 6 presents the entire ball discovery algorithm.

In each loop from line 2, we attempt to find all balls of the candidate set S_i smaller than S, where $S_i \subset S$ and $S_i = \{s_j | d(s_i, s_j) \leq 2r\}$. Since r is a small number, the size of S_i' is much smaller than

Algorithm 6 *Finding-All-Ball*$(\mathcal{S}, r)$

```
1: $F = \emptyset$;
2: for $s_i \in \mathcal{S}$ do
3:     $S_i = \{s_i\}$;
4:     $F = F + \{s_i\}$;
5:     for $s_j \in (S \setminus F)$ do
6:         if $d(s_i, s_j) \leq 2R$ then
7:             $S_i = S_i + \{s_j\}$;
8:         end if
9:     end for
10:    Construct connected graph $G$ of $S_i$;
11:    Find all cliques $C$ of graph $G$;
12:    for $c_i \in C$ do
13:        Find-all-ball-small($c_i$);     // Algorithm 5
14:    end for
15: end for
```

the original set S. While each clique will be even smaller than S_i', we can run our finding algorithm of small set to find the balls.

Note that all cliques c_i in C contain the point s_i. That's why in the output list of balls in Section 3.2.2, each ball contains point 1. We add the constrain that the balls found must contain s_i. This constrain is to ensure that Algorithm 6 does not miss any $Ball(c)$ in the input point set. Below is the proof.

Proof. Suppose we can order the input points in $\mathcal{S}$ as $s_1, ..., s_i, ...s_n$, where n is the size of $\mathcal{S}$. In each loop from line 2 of Algorithm 6 select point s_i, which is the first point in the remaining point set. If $Ball(c)$ is a ball of $\mathcal{S}$ and s_k is the first point in $Ball(c)$, then $Ball(c)$ will output in the loop which selects s_k. □

One improvement over Algorithm 6 is to construct the RP-tree and CB-tree for each s_i before line 12 instead of constructing them in *find-all-ball-small()*. This is because the balls around s_i may share many common points. Another improvement is to search not only the ball tree of s_i in *find-all-ball-small()*, but the ball trees of s_k, $s_k \in S_i$ in each loop also. In this way the search space can be better pruned.

3.4 Heuristic Greedy Algorithms for Ball Discovery

Algorithm 6 is an algorithm, which provide an exact solution set to our ball discovery problem. It works well when the point set is large but not huge and the range r is small. We propose two heuristic greedy algorithms to improve the efficiency of ball finding. Our goal is to replace *find-all-ball-small()* in Algorithm 6 in order to cope with large point sets.

3.4.1 A Heuristic Greedy Algorithm

Algorithm 7 shows the heuristic greedy algorithm. The algorithm repeatedly removes one point or some points from the boundary set of the smallest enclosed ball of the current point set until the radius of the smallest enclosed ball is smaller than or equal to r.

Algorithm 7 *HG-Ball-1*$(\mathcal{D}, r)$

```
1: $B = \emptyset$;
2: for each $D_i \in \mathcal{D}$ do
3:     $miniball(D_i)$;
4:     $r' = radius(D_i)$;
5:     $P = boundary(D_i)$;
6:     while $r' > r$ do
7:         remove one point or some points in $P$;
8:     end while
9:     $B = B \cup D_i$;
10: end for
11: return $B$;
```

There are several ways to remove points on the spherical surface of the smallest enclosed ball:

1. Remove all points on the spherical surface of the ball. This strategy is fast and will stop after only a few loops. This strat-

egy is fast but shrinks the radius too much. Many points will be missed while they should be enclosed in the ball.

2. Remove the point on the spherical surface which is the furthest to the mean vector of D_i. Removing the furthest point in P to the mean vector of D_i makes sense because since the points in a ball will cover most points of the small set, the mean vector of the current set tend to be near to the ball. Thus we could remove the furthest point to the mean to make the mean vector nearer to the ball until the $radius(D_i) \leq r$.

3. Remove the point on the spherical surface which is furthest to s_i when *HG-Ball-1()* is embedded in Algorithm 6 to substitute *find-all-ball-small()*. This strategy works with large point sets and we will not miss interesting balls that contain s_i. Here s_i is the point in algorithm 6 at line 2. Since in each candidate set, we could only find one ball using Heuristic greedy algorithms. The strategy guarantee that for any point s_i in the large point set, if there exist a $ball(c), s_i \in ball(c)$, the greedy algorithm will output a $ball(c')$ and $s_i \in ball(c')$ and $ball(c')$ is similar to $ball(c)$.

Algorithm 7 always tries to remove points on the spherical surface of the ball of a radius r' until $r' \leq r$. It may not find the exact answer as shown below in Figure 3.11. We take strategy 1 as an example. Suppose that there is a D_i with all data points in Figure 3.11 with a radius r' ($> r$) which is the largest circle. Suppose that we remove the three points, s_1, s_2 and s_3, on the circumference. The ball to be found with a $r' \leq r$ can be the smaller dot circle. But, the true ball should be the solid circle including s_1 on the circumference.

3.4.2 Another Heuristic Greedy Algorithm

Another greedy algorithm is shown in Algorithm 8. The idea of the heuristic greedy algorithm comes from the k-mean clustering

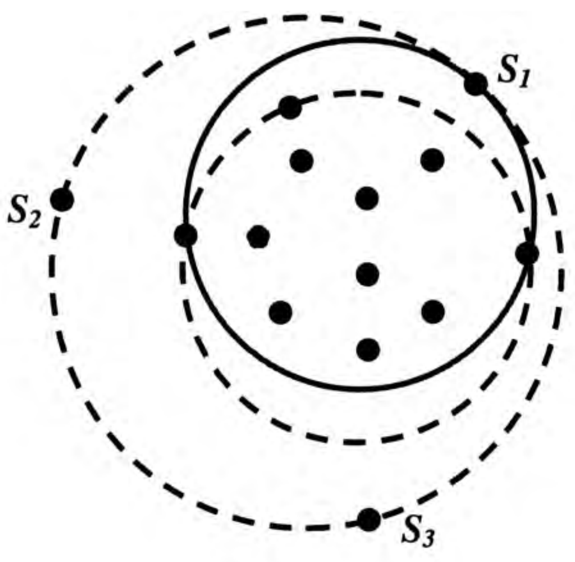


Figure 3.11: Removing points

method. At each loop, we find balls by choosing all points whose distances to the mean of the current point set are smaller than or equal to r. We then calculate the mean of the new ball and repeat such procedure until the points in the ball are not changed again.

Algorithm 8 *HG-Ball-2*$(\mathcal{D}, r)$

```
1: calculate the mean $m$ of $\mathcal{D}$;
2: $D = \{d_k | d(d_k, m) \leq r, d_k \in \mathcal{D}\}$;
3: repeat
4:     $D' = D$;
5:     calculate the mean $m$ of $D$;
6:     $D = \{d_k | d(d_k, m) \leq r, d_k \in \mathcal{D}\}$;
7: until $D = D'$;
8: return $D$;
```

□ **End of chapter.**

Chapter 4

Evaluations

In this chapter, we study the performance of the proposed algorithms and evaluate the quality of the balls (patterns) discovered.

All the experiments are performed on a Pentium 2.8GHz PC with 1GB ram, running on Microsoft Windows XP. All the programs are written in standard C++ and compiled by Microsoft Visual C++.net.

We report our experimental results on time series data sets of different length, which are all stock price series. As mentioned before, in the stock price series, people are interested in the patterns of the prices of a stock that is often selected from the frequent occurring subsequences of the time series. We will see in the following that the algorithms are trying to find such time series patterns.

We use real price series of three stocks. (a) One small time series from the daily closed prices of HSBC in the Hong Kong Stock Exchange from 2001 to 2002. It comprises of about 500 points. (b) Two large data sets consisting of about 12500 points. They are the tick by tick prices of IBM and Microsoft in the New York Stock Exchange also from 2001 to 2002. All the time series are normalized before evaluation.

We transform a time series of length n into $n - m + 1$ subsequences of length m using sliding window. Each subsequence of length m can be projected to a point in a m-dimensional Euclidean space. We use these time series subsequences to evaluate the proposed ball discovery algorithms. In the experiments, the subsequence length m and radius r of the balls are pre-defined. For data set (a), we conduct experiments when m is set to 16, 32 and 48. For data set (b), m is set to 32, 64 and 128. The pre-defined r is different for different data set and different m. We ran our algorithms in a range of r incrementally and chose the representatives as shown in the figures. In real applications, the value of r could be decided by the following strategy. Choose ten sets of similar points from the given data set S artificially, and use the average radius of the smallest enclosed ball of these point sets as the proper r for S.

We investigated the following six algorithms: the exact ball discovery algorithm introduced in Section 3.3.2, *HG-Mean*, *HG-Current*, *HG-All*, *HG-Direct* and *r-center*. *HG-Mean*, *HG-Current* and *HG-All* are Algorithm 6 embedded with Algorithm 7 by substituting *find-all-ball-small()* with different point removal strategies (See page37 for detail.). *HG-Mean* removes the point on the spherical surface which is the furthest to the mean vector. *HG-Current* removes the point on the spherical surface that is furthest to s_i which is the selected point at line 2 of Algorithm 6. *HG-All* removes all points on the spherical surface of the ball. *HG-Direct* is Algorithm 6 using Algorithm 8 as a substitute of *find-all-ball-small()*.

r-center is the algorithm that computes the number of points within a range of radius r of every existing data point and choose the one with the largest number as the first ball. Here the center point of the balls found by *r-center* is in the given data set. Algorithm 9 is *r-center* algorithm for finding the first ball of a given data set S with a pre-defined radius r.

For each stock price series and every possible value of m, we measure the following values.

1. The count of the first ball found by the above six algorithms. Here the count of a ball is the number of points in it. Recall that balls are previously unknown frequent occurring patterns. If the count of a ball is large, that means the pattern are more frequent.

Algorithm 9 *r-center*$(\mathcal{S}, r)$

```
1: $ball = \emptyset$;
2: $best\_ball\_count = 0$;
3: for each $s_i \in S$ do
4:     $count = 0$;
5:     $ball\_so\_far = \emptyset$;
6:     for $s_j \in$S$, s_j \neq s_i$ do
7:         if $d(s_i, s_j) < r$ then
8:             $count = count + 1$;
9:             $ball\_so\_far = s_j$;
10:        end if
11:    end for
12:    if $count > best_ball_count$ then
13:        $best\_ball\_count = count$;
14:        $ball = ball\_so\_far$;
15:    end if
16: end for
```

2. The mean vector of a ball is the mean of all point vector in the ball. Suppose M represents the mean vector of the first ball, O represents the center point of the first ball and C represents the count of the first ball. We use the value of the following equation to measure whether the ball is meaningful. We expect the count of a ball is as large as possible and the density of points in a ball is uniform at the mean while. $d(M, O)$ is to measure whether the density of the points in a ball is uniform. The distance between M and O would be small, if the density of a ball is uniform. If most points in a ball are gathered in a local area of the ball, the distance would be large, which means the ball is not a real pattern in fact. So a ball of smaller value of $meaningfulness$ is more meaningful than one of a larger value.

$$meaningfulness = d(M, O)/(C - 1) \quad C > 1.$$

Here $d(M, O)$ is the Euclidean distance between M and O.

3. The CPU time of finding all balls of the exact algorithm and the five heuristic greedy algorithms with different radius r.

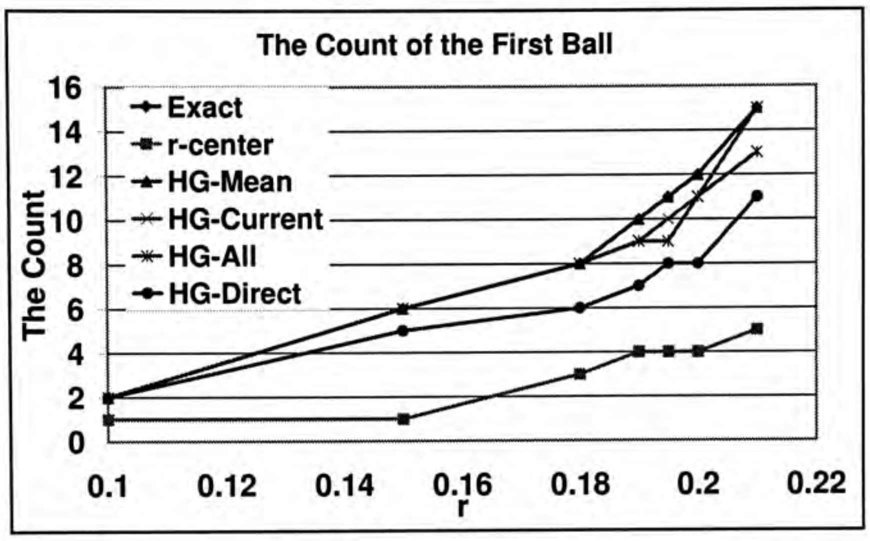
Figure 4.1 to 4.9 shows the experiment results.

Figure 4.1 shows the experiment results of HSBC stock price series when the length of subsequences is 16 with different radius r incrementally.

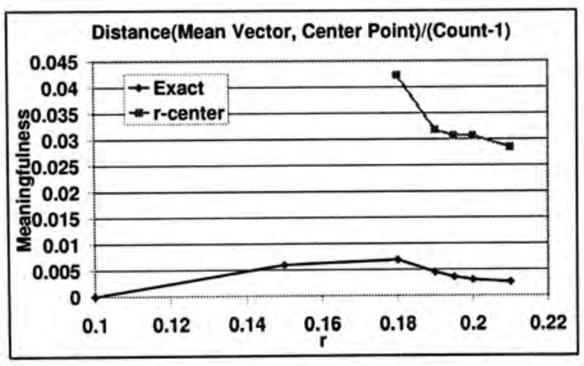
Figure 4.1(a) shows the count of the first ball found by the above six algorithms. The Y-axis is the count and the X-axis is radius r. We expect the count to be as large as possible within a given r. As we can see that the tradition method *r-center* is not good for the count of its first ball is much smaller than other algorithms. That means *r-center* fails to find some frequent patterns in the data set. *HG-Mean* is a good heuristic algorithm for the count of its first ball is always equal to the count of the exact algorithm. The performance of other heuristic algorithms are only a little worse than *HG-Man* some times. The performance of *HG-Direct* is worst in heuristic greedy algorithms, but still better then *r-center*.

Figure 4.1(b) shows the value of $meaningfulness$ of the first ball found by the exact algorithm and *r-center* respectively. The Y-axis is the value and the X-axis is radius r. In Figure 4.1(b), as we can see that the value of the exact algorithm is much smaller than *r-center*. That means the first ball found by the exact algorithm is more meaningful than the one found by *r-center*. There is no points of *r-center* when $r < 0.18$ is because that the algorithm of *r-center* cannot find a ball under this condition (The count of its first ball is equal to 1).

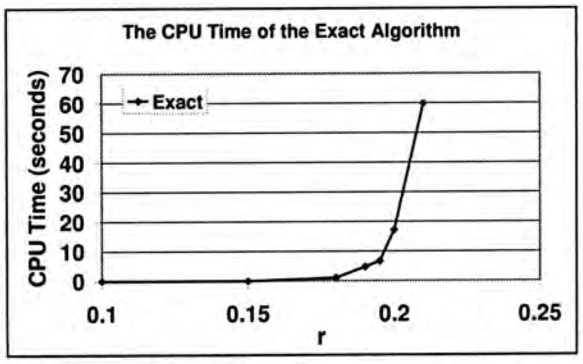
Figure 4.1(c) shows the CPU time of the exact algorithm. The Y-axis is the CPU time and the X-axis is radius r. The CPU time increases fast after certain value of radius r. This is because that the radius r is larger than the radius of real patterns. In such situation, there are too many balls in the data set, so the computational time is long.



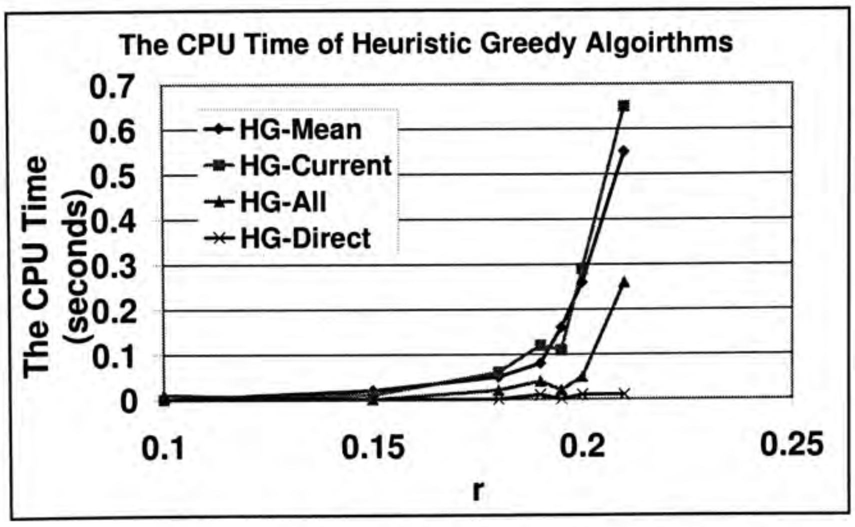
(a) The Count of the First Balls



(b) $meaningfulness = d(M, O)/(C - 1)$ $C > 1$



(c) The CPU Time of The Exact Algorithm



(d) The CPU Time of Heuristic Greedy Algorithms

Figure 4.1: Experiment Results of HSBC stock price series (Length = 16)

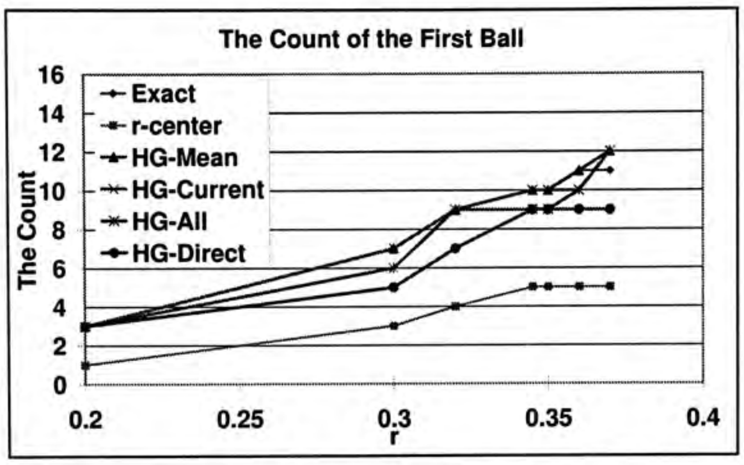
Figure 4.1(d) shows the CPU time of the four heuristic greedy algorithms which is dramatically faster than the exact algorithm. The Y-axis is the CPU time and the X-axis is radius r. But one thing need to be noticed is that the heuristic greedy algorithm only finds one ball of each candidate set while the exact algorithm finds all possible balls. The CPU time of *HG-Direct* is almost linear but the count of balls it found is less than other algorithms as can be seen from Figure 4.1(a).

Figure 4.2 shows the experiment results of HK0005 stock price series when the length of subsequences is 32 with different radius r incrementally.

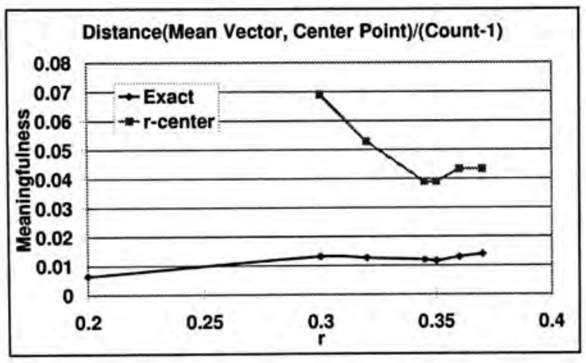
As we can see, *r-center* is still not good for the count of its first ball is much smaller than ones of other algorithms and the value of $meaningfulness$ of its first ball is larger than the exact algorithm.

Figure 4.3 shows the experiment results of HK0005 stock price series when the length of subsequences is 48.

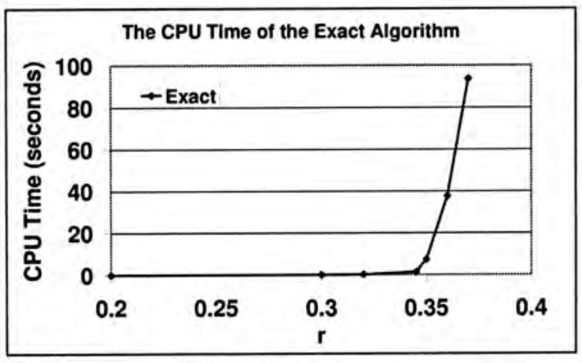
In Figure 4.3(a), the count of first is not changed when radius r larger than 0.45. We can treat 0.45 as a good choice of r in this data set when length is 48.



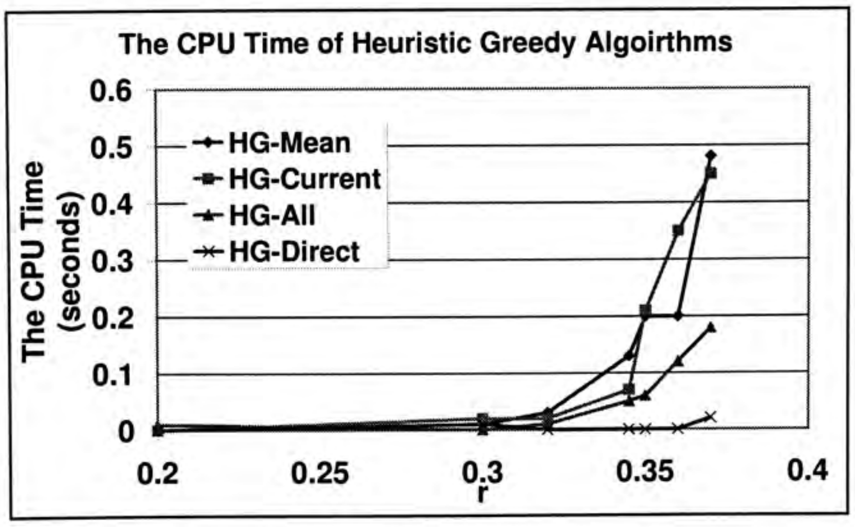
(a) The Count of the First Balls



(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$

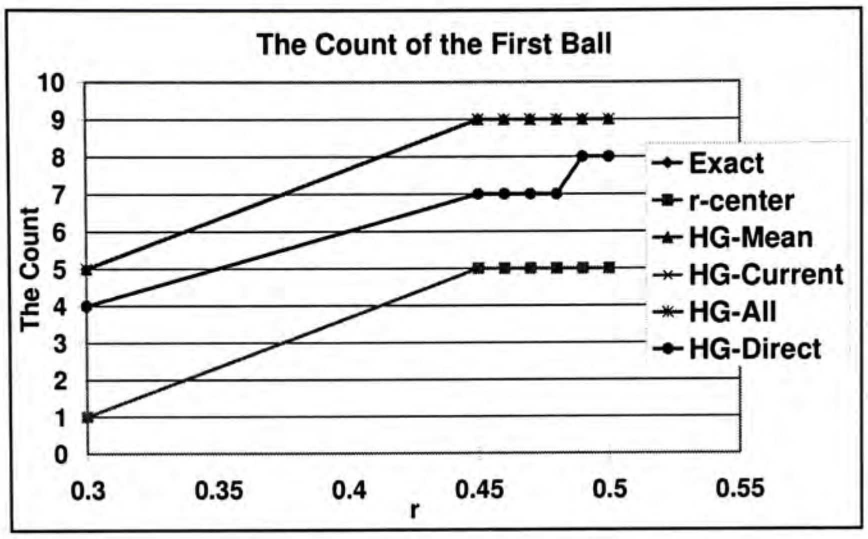


(c) The CPU Time of The Exact Algorithm

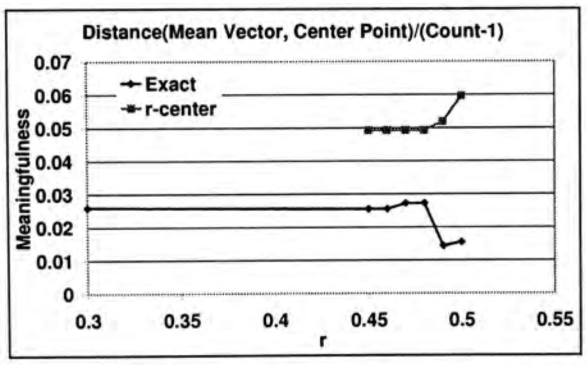


(d) The CPU Time of Heuristic Greedy Algorithms

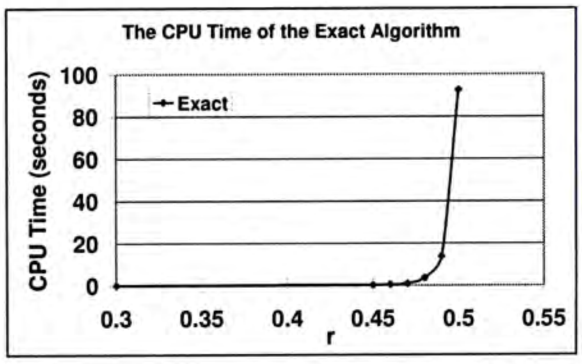
Figure 4.2: Experiment Results of HSBC stock price series (Length = 32)



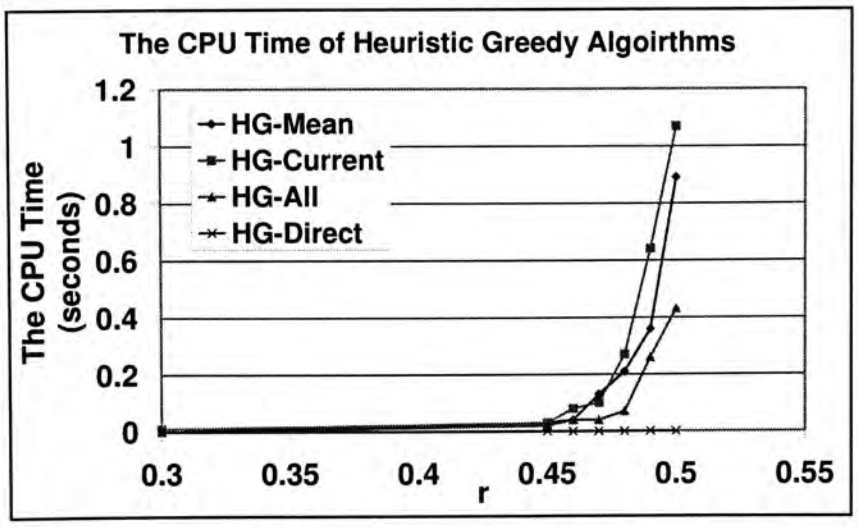
(a) The Count of the First Balls



(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$



(c) The CPU Time of The Exact Algorithm



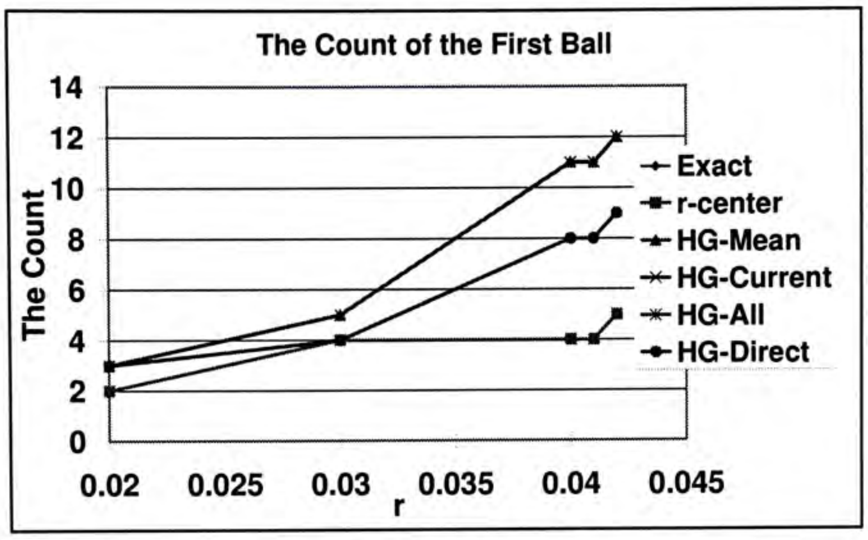
(d) The CPU Time of Heuristic Greedy Algorithms

Figure 4.3: Experiment Results of HSBC stock price series (Length = 48)

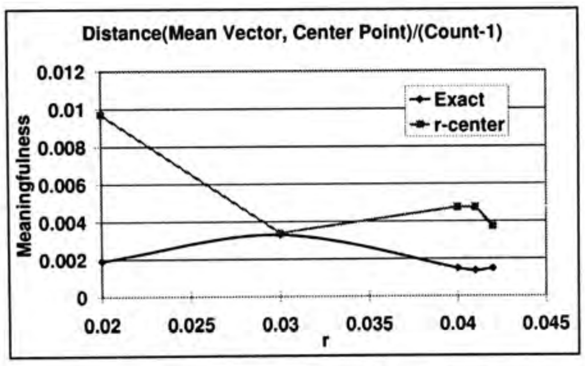
Figure 4.4 to Figure 4.6 shows the experiment results of Microsoft stock price series when the length of subsequences is 32, 64 and 128 respectively with different radius r incrementally. The results are similar to ones of the HK0005 stock price series.

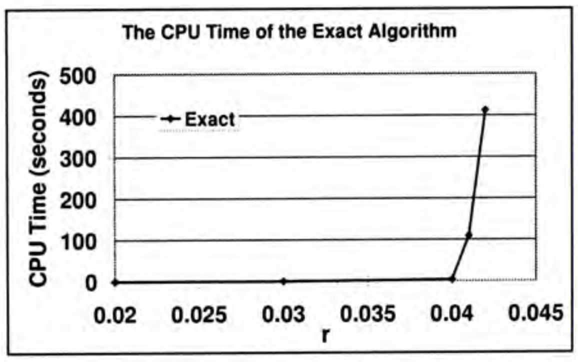
As we can see in Figure 4.4(b), at $r = 0.03$, the value of *meaningfulness* of *r-center* is only a little smaller than the one of the exact algorithm. It means in few circumstance that the quality of the first ball of *r-center* could be as good as one of the exact algorithm.

Figure 4.7 to Figure 4.9 shows the experiment results of IBM stock price series when the length of subsequences is 32, 64 and 128 respectively with different radius r incrementally. Once again we see the similar results to the above two stock price series.



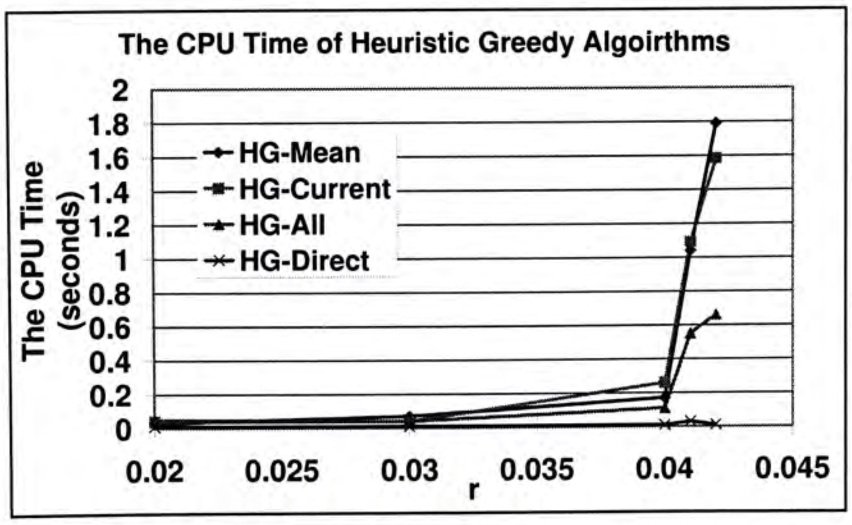
(a) The Count of the First Balls





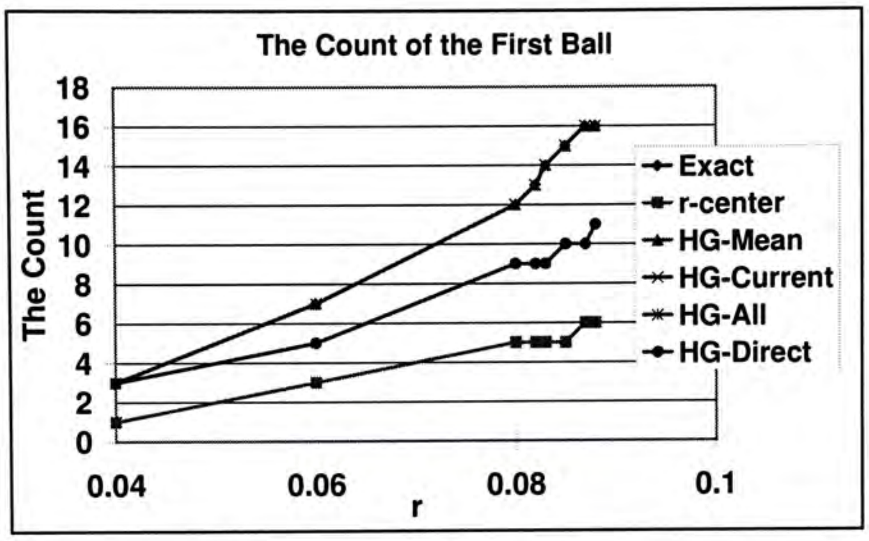
(b) $meaningfulness = d(M, O)/(C - 1)$ $C > 1$

(c) The CPU Time of The Exact Algorithm

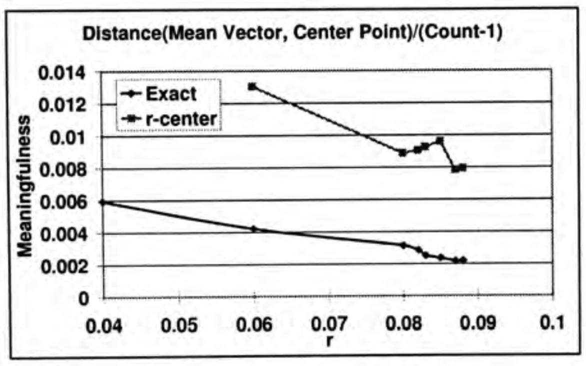


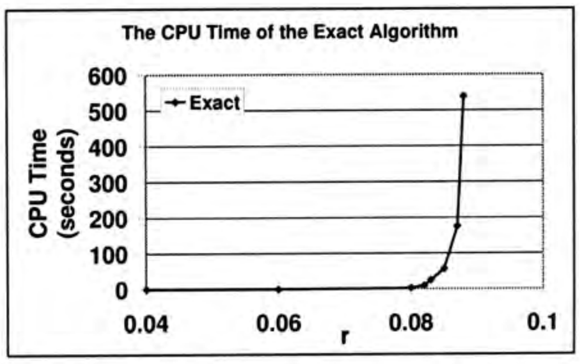
(d) The CPU Time of Heuristic Greedy Algorithms

Figure 4.4: Experiment Results of Microsoft stock price series (Length = 32)

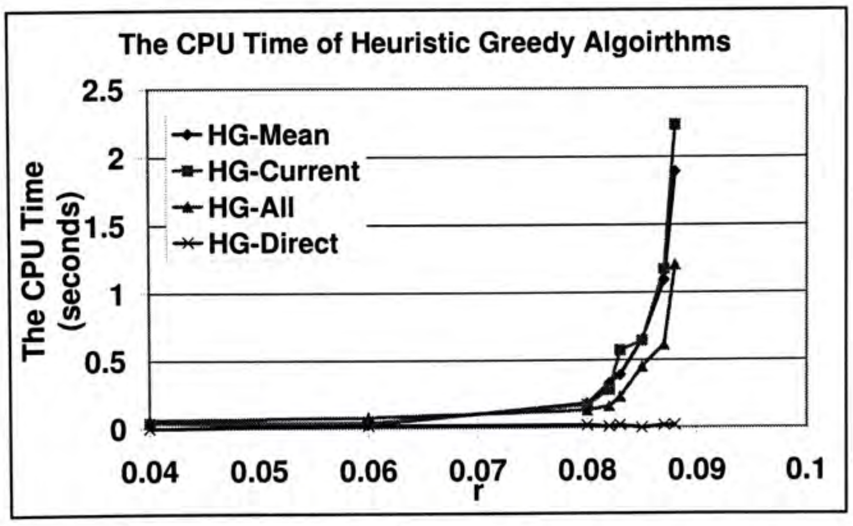


(a) The Count of the First Balls



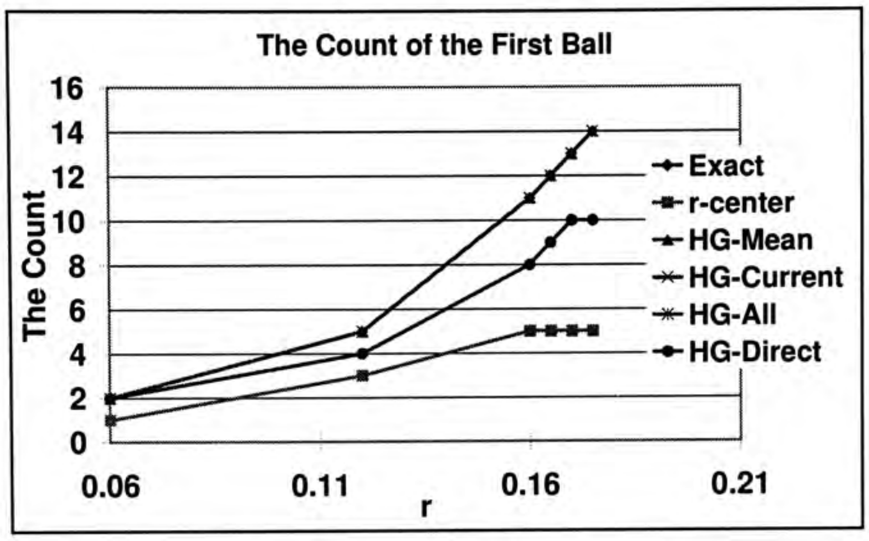


(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$ (c) The CPU Time of The Exact Algorithm

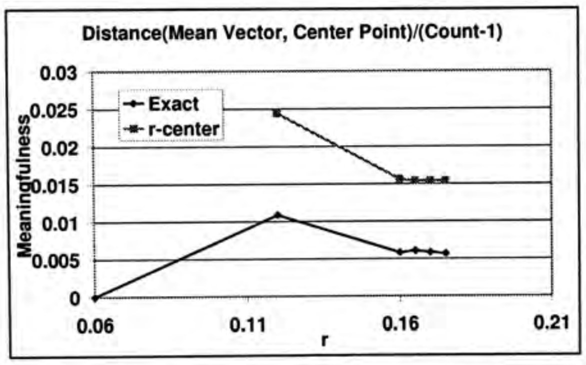


(d) The CPU Time of Heuristic Greedy Algorithms

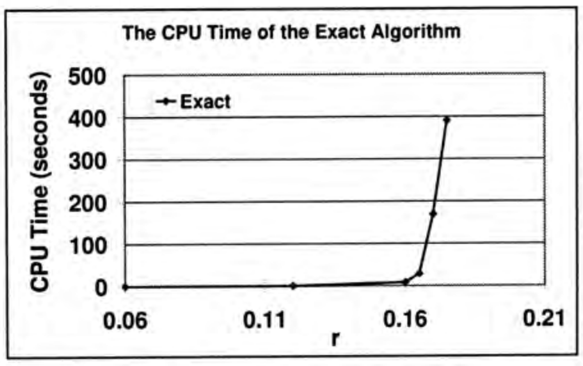
Figure 4.5: Experiment Results of Microsoft stock price series (Length = 64)



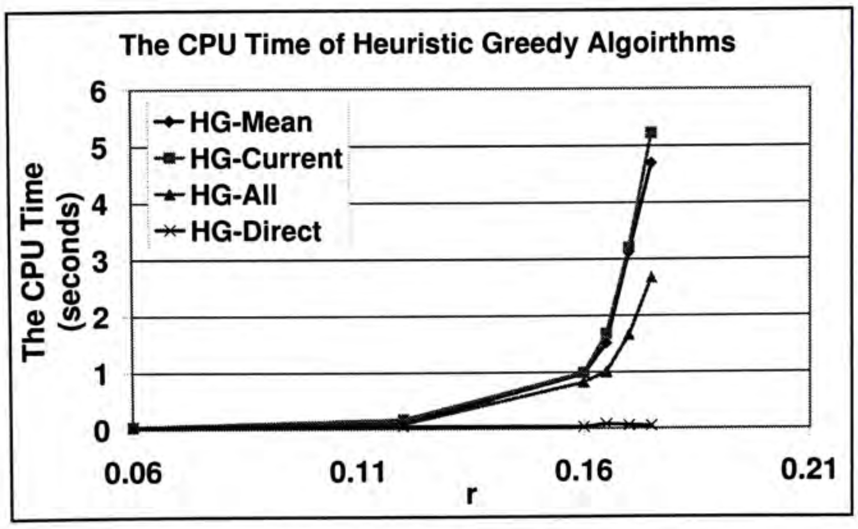
(a) The Count of the First Balls



(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$

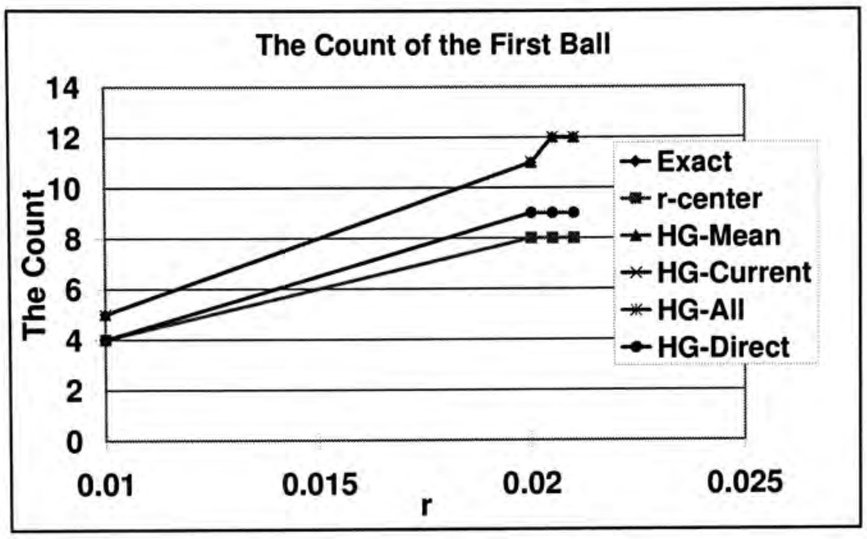


(c) The CPU Time of The Exact Algorithm

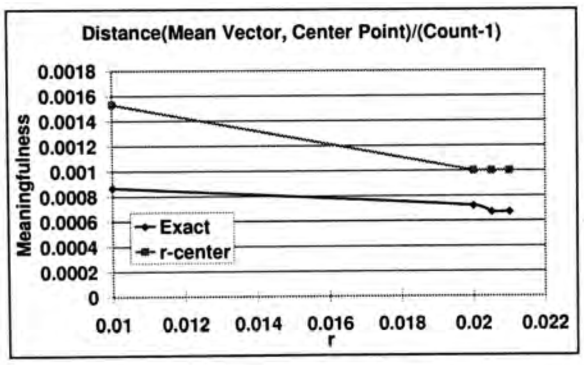


(d) The CPU Time of Heuristic Greedy Algorithms

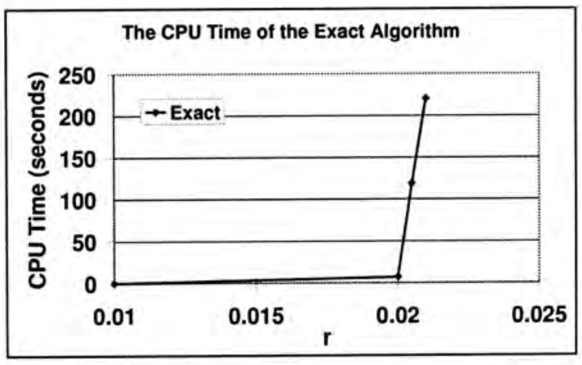
Figure 4.6: Experiment Results of Microsoft stock price series (Length = 128)



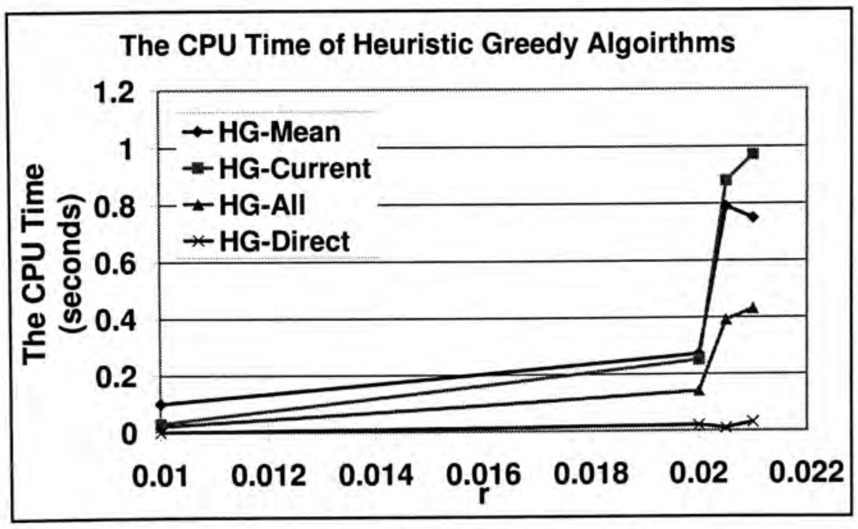
(a) The Count of the First Balls



(b) $meaningfulness = d(M, O)/(C - 1)$ $C > 1$

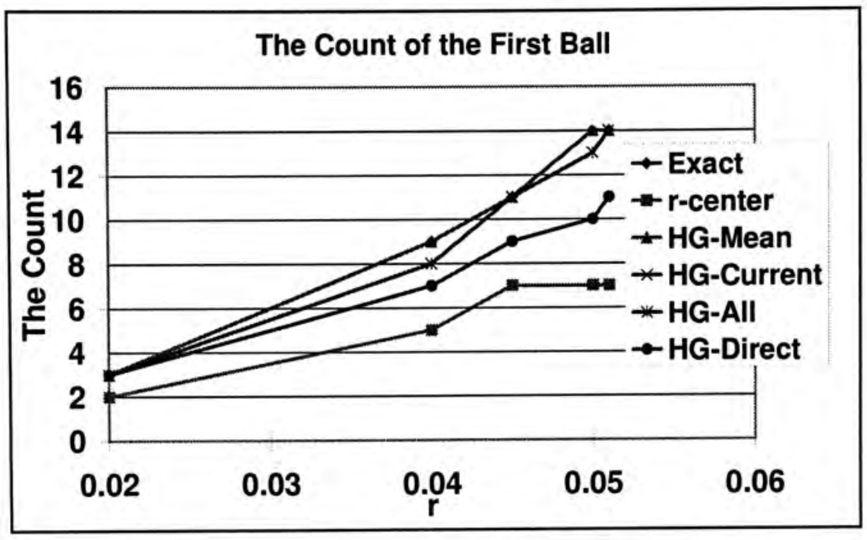


(c) The CPU Time of The Exact Algorithm

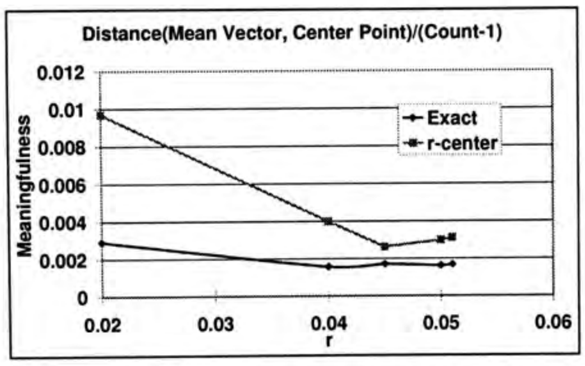


(d) The CPU Time of Heuristic Greedy Algorithms

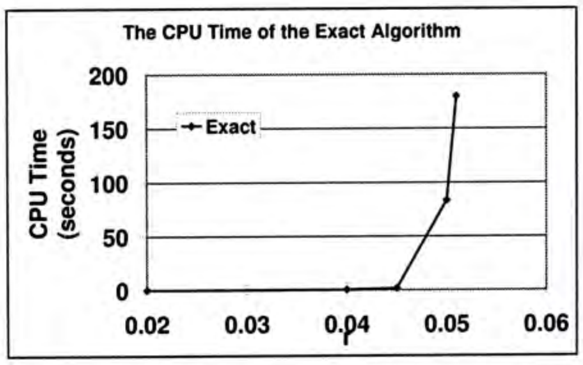
Figure 4.7: Experiment Results of IBM stock price series (Length = 32)



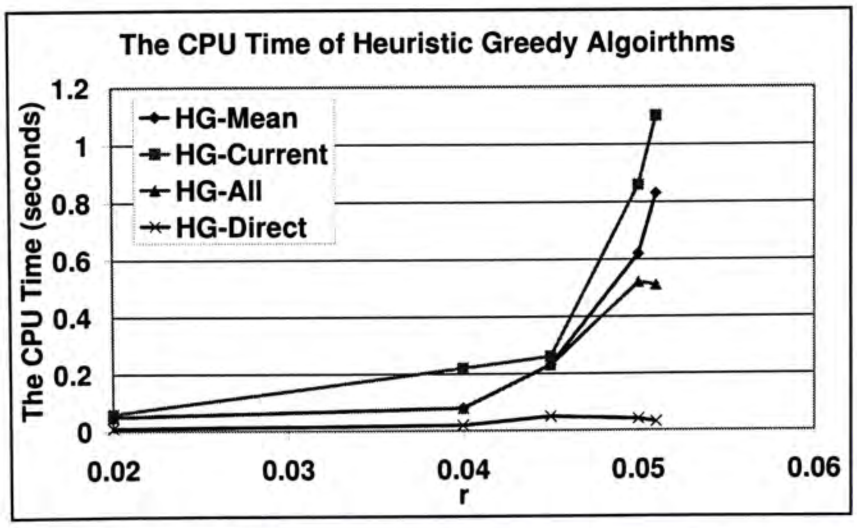
(a) The Count of the First Balls



(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$

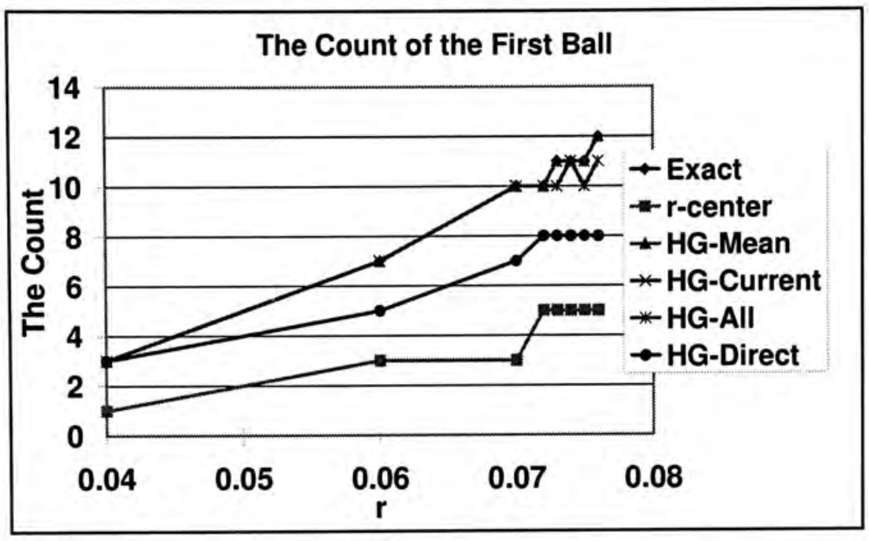


(c) The CPU Time of The Exact Algorithm

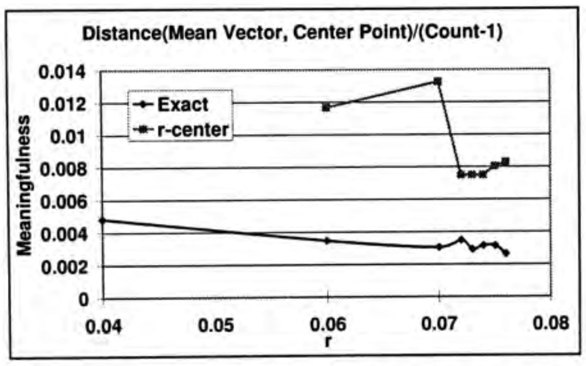


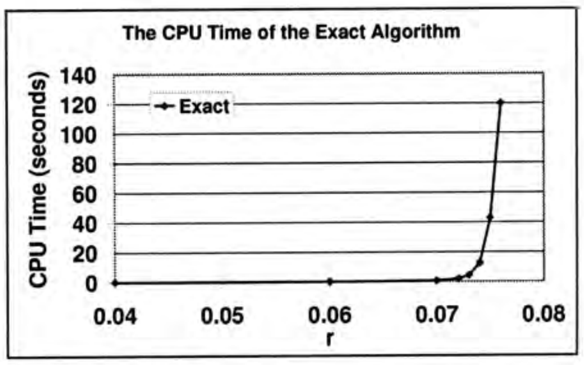
(d) The CPU Time of Heuristic Greedy Algorithms

Figure 4.8: Experiment Results of IBM stock price series (Length = 64)

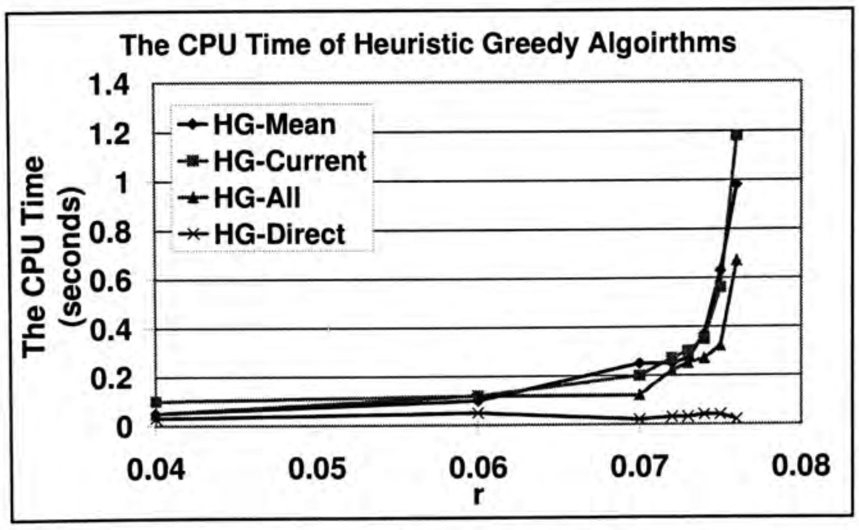


(a) The Count of the First Balls





(b) $meaningfulness = d(M,O)/(C-1)$ $C > 1$ (c) The CPU Time of The Exact Algorithm



(d) The CPU Time of Heuristic Greedy Algorithms

Figure 4.9: Experiment Results of IBM stock price series (Length = 128)

Below is an analysis of the top-k balls found by the following four algorithms: Exact algorithm, *HG-Mean*, *HG-Current* and *HG-All*. Since it's obvious that the count of balls found by *HG-Direct* is much smaller than other algorithms, we skipped it here. We choose the stock of HSBC in Hong Kong Stock Market as an example. Here $length = 16$ and $r = 0.2$. In the experiment, we found that the count of the top seven balls found by the exact algorithm is 12 and from the eighth ball, the count is smaller than 12, so we choose $k = 7$ for the analysis.

Each table from Table 4.1 to Table 4.4 contains six columns: Content of balls, C, r, $Min.$, $Max.$ and $Avg.$. The content of a ball is the sequence number of all points in the ball where the sequence number of a point is the start position of the subsequence in the original time series. C is the total number of points in a ball, in other words, the count of a ball. r is the actual radius of the ball. $Min.$ is the minimum distance between a point in a ball and the center point. $Max.$ is the maximum distance between them. It's obvious that r is equal to $Max.$. $Avg.$ is the average distance between a point in a ball and the center point. Smaller average distance means that the points in a ball are closer to the center point.

Table 4.2 to Table 4.4 have two extra columns: $recall$ and $precision$. Each row in Table 4.2 to Table 4.4 is aligned to a row in Table 4.1 according their first column: Content of Balls. Suppose $CBExact$ is the column Content of Balls of a row A in Table 4.2 to Table 4.4 and $CBHeuristic$ is the one of a row B in Table 4.1. A and B is aligned if $|CBExact \cap CBHeuristic|$ is maximum. Then

$$recall = \frac{|CBExact \cap CBHeuristic|}{|CBExact|} \tag{4.1}$$

$$precision = \frac{|CBExact \cap CBHeuristic|}{|CBHeuristic|} \tag{4.2}$$

As we can see from the table 4.1, the count of the top seven balls is 12 and they share part of their points.

Content of Balls	C	r	$Min.$	$Max.$	$Avg.$
147 148 149 150 320 321 322 374 375 470 471 472	12	0.199521	0.173187	0.199521	0.192137
147 148 149 319 320 321 322 323 373 374 470 471	12	0.199115	0.169821	0.199115	0.190499
148 149 150 151 320 321 322 323 374 375 471 472	12	0.199863	0.177048	0.199863	0.193953
148 149 150 320 321 322 323 374 375 470 471 472	12	0.199722	0.157453	0.199722	0.189658
148 149 150 320 321 322 323 373 374 375 470 471	12	0.198925	0.162921	0.198925	0.192065
148 149 150 319 320 321 322 323 373 374 470 471	12	0.198454	0.17453	0.198454	0.194198
148 149 150 151 321 322 323 375 376 471 472 473	12	0.19701	0.173611	0.19701	0.194915

Table 4.1: Top 7 Balls of the Exact Algorithm

Content of Balls	C	r	$Min.$	$Max.$	$Avg.$	$recall$	$precision$
147 148 149 319 320 321 322 323 373 374 470 471	12	0.199115	0.169821	0.199115	0.190499	1	0
148 149 150 320 321 322 323 373 374 375 470 471	12	0.198925	0.162921	0.198925	0.19206	1	0
147 148 149 150 320 321 322 323 374 470 471	11	0.196627	0.155546	0.196913	0.188528	0.83	0.91
147 148 319 320 321 322 373 374 375 470 471	11	0.196913	0.155546	0.196913	0.188528	0.83	0.91
148 149 150 151 321 322 323 324 374 471 472	11	0.198894	0.163601	0.198894	0.187942	0.83	0.91
148 149 319 320 321 322 323 373 374 470 471	11	0.19373	0.17065	0.19373	0.189872	0.92	1
148 319 320 321 322 372 373 374 375 470 471	11	0.198187	0.166973	0.198187	0.191099	0.83	0.91

Table 4.2: Top 7 Balls of Algorithm *HG-Mean*

Table 4.2 is the experiment result of *HG-Mean* algorithm. As we can see that only the first two balls are in the top seven balls found by the exact algorithm. The count of other balls is smaller than 12.

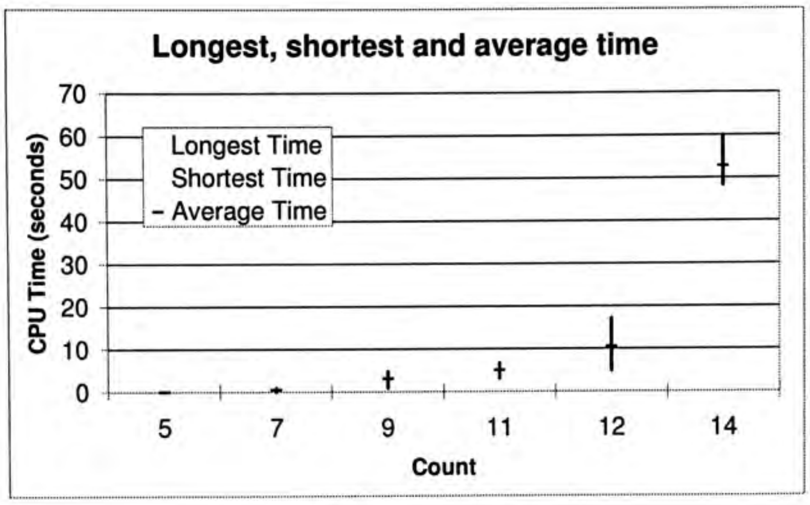
Table 4.3 and Table 4.4 shows the experiment results of *HG-Current* and *HG-All* separately. Both of them fail to find a ball whose count is equal to 12.

Content of Balls	C	r	$Min.$	$Max.$	$Avg.$	$recall$	$precision$
148 149 150 151 321 322 323 324 375 471 472	11	0.199229	0.153394	0.199229	0.18873	0.83	0.91
148 319 320 321 322 372 373 374 375 470 471	11	0.198187	0.166973	0.198187	0.191099	0.83	0.91
149 150 151 152 153 322 323 324 325 473 474	11	0.197976	0.159178	0.197976	0.188273	0.5	0.55
149 150 151 321 322 323 375 376 471 472 473	11	0.196214	0.173435	0.196214	0.193242	0.92	0
150 151 321 322 323 374 375 376 471 472 473	11	0.199121	0.179541	0.199121	0.194328	0.83	0.91
141 142 143 267 268 269 270 315 316 317	10	0.198025	0.177274	0.198025	0.194415	0	0
146 147 148 319 320 321 322 373 470 471	10	0.199938	0.170469	0.199938	0.191933	0.67	0.8

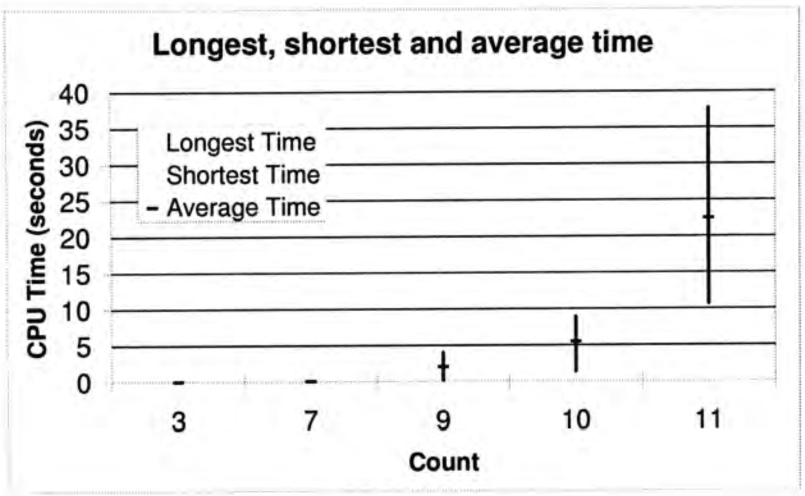
Table 4.3: Top 7 Balls of Algorithm *HG-Current*

Content of Balls	C	r	$Min.$	$Max.$	$Avg.$	$recall$	$precision$
148 149 150 320 321 322 323 373 374 375 471	11	0.197434	0.159057	0.197434	0.189461	0.92	1
148 149 150 151 321 322 323 374 375 471 472	11	0.197398	0.173511	0.197398	0.191148	0.92	1
141 142 143 267 268 269 270 315 316 317	10	0.198025	0.177274	0.198025	0.194415	0	0
147 148 319 320 321 322 373 374 470 471	10	0.190199	0.164218	0.190199	0.185061	0.83	1
147 148 319 320 321 322 372 373 374 471	10	0.197762	0.170676	0.197762	0.19064	0.75	0.9
148 149 150 151 321 322 323 375 471 473	10	0.194101	0.176548	0.194101	0.192092	0.83	1
149 150 151 321 322 323 374 375 471 472	10	0.19453	0.177295	0.19453	0.18955	0.83	1

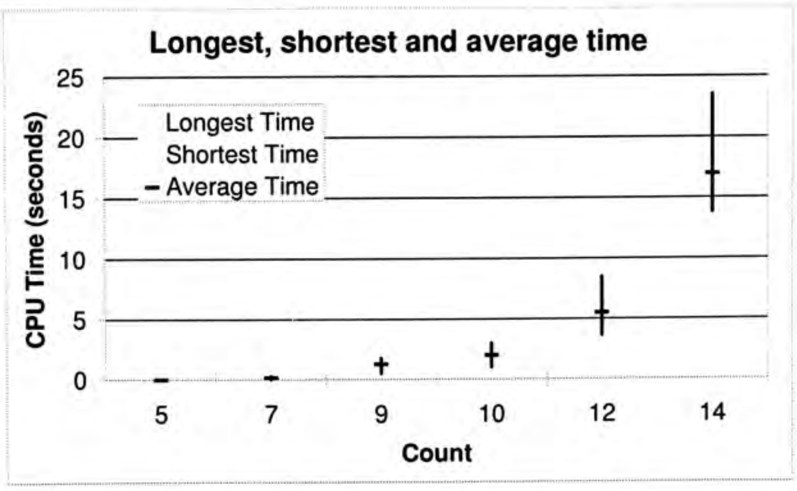
Table 4.4: Top 7 Balls of Algorithm *HG-All*



(a) Length = 16



(b) Length = 32



(c) Length = 48

Figure 4.10: The longest, shortest and average CPU time of the exact algorithm on ten stock price series from Hong Kong Stock Exchange

Now, we choose ten stock price series from Hong Kong Stock Exchange to evaluate the average CPU time of the exact algorithm. Figure 4.10 shows the longest, shortest and average CPU time of the exact algorithm of ten stock price series from Hong Kong Stock Exchange. The Y-axis is the CPU time and the X-axis the count of the first ball. For each value of the count, the figure shows the longest and shortest CPU time of the exact algorithm at the two ends of the vertical line separately. And the short horizontal line between the two ends of the vertical line is the average CPU time of the exact algorithm on the ten stock price series.

In general, we find that the count of balls found by the exact algorithm is always larger than one found by *r-center* at the same length m and radius r. The value of $meaningfulness$ of the first ball found by the exact algorithm is much smaller than the one found by *r-center*, which means balls found by the exact algorithm are more meaningful than ones found by *r-center*. The CPU time of the exact algorithm will increase fast after certain value of radius r and this value is the proper radius of the data set. The CPU time of heuristic greedy algorithms is much smaller than one of the exact algorithm. Among them, *HG-Direct* is almost linear though the count of its balls is not as large as other heuristic greedy algorithms (still better than *r-center*).

Top-k ball analysis shows that the first few balls found by *HG-Mean* is in the top-k balls found by the exact algorithm. But *HG-Mean* fails to find all possible top-k balls as the exact algorithm does. Other heuristic algorithms *HG-Current* and *HG-All* fail to find even one top-k ball found by the exact algorithm.

□ **End of chapter.**

Chapter 5

Discussion

In this section, we discuss several issues related to further improvements of the performance of our ball discovery algorithms.

5.1 Order and Index the Points

In Section 3.3.2, for ensuring algorithm 6 not to miss any ball, we assumed that we could order the input points. We now propose a method to order the points based on the density of the points. At line 7 of Algorithm 6, we compute the neighbors of s_i within a range R. We can based on the number of neighbors to order the points from large to small. This idea is based on the observation that balls are more likely to be generated from high density neighborhood.

5.2 Incremental Points Update

Point set may be changed for time to time, and our divide-and-conquer algorithm is stable for the update of data points. If a point s_i is removed from the data set, just remove it from the CB-tree and link the children of s_i to its parent then do the operation $adjust$ and $closeset$ to keep the CB-tree compact and complete. If a point s_j is added to the data set, what we need to do is just recalculate point s_k if $d(s_j, s_k) \leq r$ and update the CB-tree.

5.3 Smallest Enclosed Ball Algorithm

We have mentioned in the dissertation that the algorithm of smallest enclosed ball works well in low dimensions. In fact, there is some other algorithms that can compute the smallest enclosed ball fast almost in the arbitrary dimensions. If one really needs to calculate the top-k balls in a very high dimensional space, he could refer to the high dimensional *miniball* algorithm.

□ **End of chapter.**

Chapter 6

Conclusion and Future Research

The problem we study in this dissertation is to find frequent occurring previously unknown patterns which are sets of high-dimensional data points in a Euclidean space. We attempt to find k distinctive balls that contain the largest numbers of data points in a given data set.

We proposed several solutions for the ball discovery problem including an exact solution and several approximate solutions. Two new pruning techniques implemented in novel tree-structures with help of indexing are also presented to speed up the exact solution. This ensures that our exact algorithm could finish in a reasonable time.

From the experiments, as we can see that the balls found by our exact algorithm is more dense than the one found by the traditional method *r-center*. Sometimes, the traditional method even cannot find a single ball of a given data set with a pre-defined r. Heuristic Greedy algorithms give good approximate answers which are still better then *r-center*. Thus in really large data set, the heuristic greedy algorithms are good choices, among which, *HG-Mean* performs best.

There are several directions in which we intend to extend this work.

- Our ball discovery algorithm is a top-down algorithm because it repeats removing points on the boundary set. Could a bottom-up algorithm solve this problem better?

- We only considered the problem of speeding up main memory search. If the data set are really huge, disk-based techniques are highly desirable.
- Our ball discovery algorithm utilizes the Euclidean metric and may be modified to use the Minkowski metric. However, there are some other similarity measures proposed this year. This encourages us to generalize our work.

□ **End of chapter.**

Appendix A

Appendix

A.1 Fundamental Algorithms

In this section, we will briefly review two basic building tool for our k-ball discovery algorithm in high dimensions: the algorithm of computing smallest enclosed ball of a point set in Euclidean space and the algorithm of finding all cliques in an undirected graph. Our algorithm uses these two algorithms to solve the k-ball problem. The aim to introduce these two algorithms here is to make the k-ball discovery algorithm completed in detail.

A.1.1 Computing Smallest Enclosed Ball of a Point Set in Euclidean Space

The smallest enclosed ball is the ball that contain the give point set m-dimensional Euclidean space when the radius of such ball is smallest. The algorithm to computing smallest enclosed ball is first introduced at [26] which adopt a move-to-front strategy to speed the algorithm. Berne Gätner implement a slightly modified algorithm in C++ based on his paper [12].

Let $S = s_1, ..., s_n$ is a point set of size n in R^m Euclidean space and $MB(S)$ denote the ball of the smallest radius that contain S. $MB(S)$ is unique. For $S, B \subseteq R^m$ and $S \cap B =$, let $MD(S, B)$ be the smallest ball that contains S and has all points of B on its boundary. Then we have $MB(S) = MD(S, \emptyset)$, and if $MD(S, B)$ exists, it is unique, too. Finally, define $CB(B) = MD(\emptyset, B)$ to be the smallest ball wit all points of B on the boundary[1]. Algorithm 10 is procedure of computing a smallest enclosed ball of point set S.

Algorithm 10 $miniball(S_n, B)$

```
compute a random permutation of $S_n = (s_1, ..., s_n)$ of $S$;
$mb = CB(B)$
if $|B| = m + 1$ then
    return;
end if
for $i = 1$ to $n$ do
    if $p_i \notin mb$ then
        $mb = miniball(S_{i-1}, B \cup \{s_i\})$;
        move $s_i$ to the front of $S_n$;
    end if
end for
```

The algorithm *miniball* compute the smallest enclosed ball in following a strategy: if a point s_i is not in the current ball mb, then it must line on the boundary of the point set $\{s_i\} \cup mb$. By iterating such process from an empty set, we finally got the boundary point set B of the input point set S, then $CB(B)$ is the smallest enclosed ball of S.

miniball is a randomized algorithm because at the beginning, we first randomized the point list S_n. At line 7, the worst case is that in p_i is not a boundary point of $MB(S)$, in this case, we will call $CB(B)$ many times while the procedure $CB(B)$ is a linear programming algorithm[2].

One simple but important technique to speed the algorithm is to select the 'important' point to test instead of the straight forward at line 6. We want to end the iteration as soon as possible, so we expect the boundary points of $MB(S_n)$ come as early as possible to reduce the times of calling $CB(B)$. Such important points is the furthest

[1]$CB(B)$ may not exists for some B

[2]The procedure to computer $CB(B)$ is omitted here for brief. One can refer to [26] for details.

point to the center of $MB(S_n)$.

In our experiment part, we slightly modify Gätner's implementation code for the smallest enclosed ball algorithm which is written in C++ to make fit for our algorithm. The author claim that it only works well at dimension $d \leq 20$ and will expect a severe performance dropoff when come to higher dimension. From our experiments, we found it's true for random large point set. But one interesting finding from both theory and experiments is that when the number of input points is not very large, the algorithm is fast enough to calculation the smallest enclosed ball of the small point set in time. In k-ball problem, the previous given range r is a small number compared to the diameter of the whole input point set, so the number points in each of the final top-k ball could not be very large.

A.1.2 Finding All Cliques of an Undirected Graph

A *clique* in a graph is a complete subgraph in which any two of vertices are connected by an edge. A maximal clique is a clique that is not contained in any other clique. In our k-ball discovery algorithm, we adopt the cliques finding algorithm in [3]. Almost all the newer algorithms are a modified version of the origin algorithm in [3] with some optimal techniques.

The algorithm in [3] is mainly dealing three sets of vertices. One is the set *compsub* which is the set to be extended by a new point or shrunk by one point on traveling along a branch of the backtracking tree. Another is the set *candidates* is the set of all points that will in due time serve as an extension to the present configuration of *compsub*. The last is the set *not* is the set of all points that have at an earlier stage already served as an extension of the present configuration of *compsub* and are now explicitly excluded. The the basic idea of the algorithm is show below:

1) Select a candidate;

2) Adding the selected candidate to *compsub*;
3) Creating new sets *candidates* and *not* from the old sets by removing all points not connected to the selected candidate (to retain consistent with the definition), keeping the old sets in tact;
4) Calling the extension operator to operate on the sets just formed;
5) Upon return, removal of the selected candidate from *compsub* and its addition to the old set *not*.

Some technique another fast version algorithm by slightly modifying the above algorithm. At the first step of selecting a candidate, instead of selecting a random vertex, they select the vertex whose count is lowest. Here count is the number of candidates that this vertex is not connected to.

This is an old algorithm but still works well in small data set and the use of it in our algorithm will come clear in Section 3.3.

A.2 An Example of a Small Data Set

The data used in the construction example of CB-tree in section 3.2.3 is a real world data from a time series. Each point is a 32 dimensional vector in the Euclidean space. We list the data point below one by one for references.

Point 1: (0.0566, 0, 0.0283, -0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132)

Point 2: (0, 0.0283, -0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283)

Point 3: (0.0283, -0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -

0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184)

Point 4: (-0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184, 0.1274)

Point 5: (0.0283, -0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184)

Point 6: (-0.1274, -0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184, 0.1274)

Point 7: (-0.1274, -0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184, 0.1274, 0.1557)

Point 8: (-0.0991, -0.2406, -0.3113, -0.2123, -0.2547, -0.1698, -0.2406, -0.1698, -0.1415, -0.1698, -0.2972, -0.1981, -0.184, -0.1981, -0.0566, 0.0142, -0.1274, -0.0566, 0.1981, 0.2406, 0.1132, 0.184, 0.1557, 0.1557, 0.2264, 0.3396, 0.1132, 0.0283, 0.184, 0.1274, 0.1557, 0.1415)

□ **End of chapter.**

Bibliography

[1] P. K. Agarwal and C. M. Procopiuc. Exact and approximation algorithms for clustering (extended abstract). In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 658–667, San Francisco, California, USA, 1998.

[2] J. Bar-Ilan, G. Kortzars, and D. Peleg. How to allocate network centers. *Journal of Algorithms*, 15(3):385–415, 1993.

[3] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[4] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 626–635, El Paso, Texas, 1997.

[5] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining, (KDD'03)*, pages 493–498, Washington DC, USA, 2003.

[6] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: Model and algorithms. In *Proceedings of the 2004 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'04)*, pages 707–718, Paris, France, 2004.

[7] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *Proceeding of Knowledge*

Discovery and Data Mining, pages 16–22, New York City, New York, USA", 1998.

[8] M. Dyer and A. Frieze. A simple heuristic for the p-center problem. *Operations Research Letter*, 3:285–288, 1985.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. Density-connected sets and their application for trend detection in spatial databases. In *Proceedings of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, pages 10–15, Newport Beach, California, USA, 1997.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 419–429, Minneapolis, Minnesota, United States, 1994.

[11] L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *Proceedings of the 2002 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'02)*, pages 370–381, Madison, Wisconsin, USA, 2002.

[12] B. Gärtner. Fast and robust smallest enclosing balls. In *Proceedings of ESA '99*, pages 325–338, Prague, Czech Republic, 1999.

[13] V. Guralnik and J. Srivastava. Event detection from time series data. In *Proceedings of the 5th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 33–42, San Diego, California, United States, 1999.

[14] J. Han and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of the 15th International Conference on Data Engineering*, pages 106–115, Sydney, Australia, 1999.

[15] D. J. Hand. Pattern detection and discovery. In *ESF Exploratory Workshop*, volume 2447 / 2002 of *Lecture Notes in Computer Science*, pages 16–19, London, UK, September 16-19 2002. Springer-Verlag Heidelberg.

[16] D. Hochbaum and D. shmoys. A best possible approxmation algorithm for the k-center problem. *Mathematics of Operation Research*, 10:180–184, 1985.

[17] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 102–111, Edmonton, Alberta, Canada, 2002.

[18] E. Keogh, J. Lin, and W. Truppel. Clustering of time series subsequences is meaningless: Implications for past and future research. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 115–122, Melbourne, FL., 2003.

[19] J. Lin, E. Keogh, S. Lonardi, and P. Patel. Finding motifs in time series. In *Proceedings of the 2nd Workshop on Temporal Data Mining, at the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, 2002.

[20] L. Lovász. On the ratio of optimal integral and factional covers. *Discrete Mathematics*, 13:383–390, 1975.

[21] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666, San Francisco, California, 2002.

[22] P. Patel, E. Keogh, J. Lin, and S. Lonardi. Mining motifs in massive time series database. In *Proceedings of the 2002*

IEEE International Conference on Data Mining, pages 370–377, Maebashi City, Japan, 2002.

[23] P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in dna sequences. In *Proceedings of the International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 269–278, UC San Diego, La Jolla, California, USA, 2000.

[24] Y. Tanaka and K. Uehara. Discover motifs in multi-dimensional time-series using the principal component analysis and the mdl principle. In *Proceedings of the Third International Conference, MLDM 2003*, pages 252–265, Leipzig, Germany, 2003.

[25] A. Udechukwu, K. Barker, and R. Alhajj. Discovering all frequent trends in time series. In *Proceedings of the 2004 Winter International Symposium on Information and Communication Technologies (WISICT 2004)*, pages 1–6, Cancun, Mexico, 2004.

[26] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[27] J. X. Yu, M. K. Ng, and J. Z. Huang. Patterns discovery based on time-series decomposition. In *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 336–347, Hong Kong, China, 2001.