

國立交通大學

資訊科學與工程研究所

碩士論文

單一直線地圖標記點數最佳化

Maximization of Points Labeling Problem on a Single Line

研究生：余昆霖

指導教授：李德財 教授

中華民國九十五年六月

單一直線地圖標記點數最佳化
Maximization of Points Labeling Problem on a Single Line

研究生：余昆霖

Student : Kuen-Lin Yu

指導教授：李德財

Advisor : D.T. Lee

國立交通大學
資訊科學與工程研究所
碩士論文



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

單一直線地圖標記點數最佳化

學生：余昆霖

指導教授：李德財

交通大學資訊工程學研究所

摘 要

在這篇論文裡，我們主要探討的是地圖標記的問題，而且所有需要被標記的點都限制在單一直線上。

已經眾所皆知的是，在提供了已排序的點集的前提下，1d4P矩形標籤、1d4S方形標籤以及Slope4P固定高度(寬度)的標籤放置問題都可以在線性的時間內解決。

我們在這篇論文中將原本「判斷版本」的結果擴展到「最佳化版本」：也就是Max-1d4P的標籤放置問題，它將會從給定在水平線上的所有點中找出在不違反規則的前提下最多可標記的點數。

我們將會提供一個 $O(n \log n)$ 的演算法，這裡的 n 是所輸入的點數。

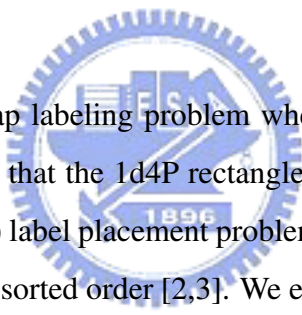
Maximization of Points Labeling Problem on a Single Line

Student: Kuen-Lin Yu

Advisor: Prof. Der-Tsai Lee

Institute of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University

ABSTRACT



In this paper, we consider a map labeling problem where the anchors to be labeled are restricted on a line. It is known that the 1d4P rectangle label, the 1d4S square label and the Slope4P fixed height (width) label placement problems can all be solved in linear time provided that the anchors are in sorted order [2,3]. We extend the decision version results to the maximization version: Max-1d4P label placement problem, which is to maximize the number of labels that can be placed on a given set of anchors on a horizontal line. We present an $O(n \log n)$ time algorithm solution, where n is the input size.

Acknowledgements

這篇論文能夠完成，首先要感謝的就是兩年來李德財老師給我的指導，在我的研究過程中不僅給予我良好的研究環境，並時常提出一些創新的想法，使我能夠在這個研究領域中盡情的發揮，甚為感激，特此銘謝。

也要感謝我的論文口試教授趙坤茂老師與荊宇泰老師，感謝兩位老師給予許多寶貴的意見與想法，讓我的論文能夠更趨完善。

再來要感謝的是實驗室廖崇碩學長，在這兩年的研究生生涯中給了我許多的建議與幫助，並陪同我一起解決研究過程中所遭遇到的種種難題，讓我能夠隨時隨地都找得到人可以討論問題，在此一定要特地感謝學長。另外也非常感謝其他學長與同學這兩年來的陪伴，讓我這兩年可以過得多采多姿。

最後要感謝我的家人給我無條件的支持，使我可以毫無後顧之憂的完成我的學業。也感謝元元、小摸、豬頭、小妞、寶萱、佳盈、山姆、丫後、小金、小艾、葛瑞絲以及我沒有寫到的大家這一路的相挺，讓我知道我不是一個人的。

僅以本論文獻給所有關心過我的人，感謝你們大家的支持。

Contents

Title	i
Chinese Abstract	ii
English Abstract	iii
Acknowledgements	iv
Contents	v
List of Figures	vi
1 Introduction	1
2 Preliminaries	4
3 Maximum Independent Set of Label Placements for Different Models	9
3.1 1d2PH, 1d2PV	9
3.2 1d4P	10
3.3 1d2kP	19
4 Conclusion Remarks	20
Bibliography	21
Appendix I : Example of R-map	24
Appendix II : Experimental Codes	30



List of Figures

1.1	Illustration of 2PV, 2PH, 4P, and 4S models.	2
2.1	Example of maximizing the number of labeled points.	4
2.2	Example of the R-map.	7
3.1	(a) Example of incomparability between realizations resulting from placing labels 1 and 3. (b) Example of undecidable situation as to whether we should choose to include label 1 or not.	11
3.2	Illustration of Lemma 3.2.4.	13
3.3	Proof for Lemma 3.2.8 (b).	18
I.1	Illustration of the growing incomparable points number (1)	24
I.2	Illustration of the growing incomparable points number (2)	25
I.3	Illustration of the growing incomparable points number (3)	25
I.4	Illustration of the growing incomparable points number (4)	26
I.5	Illustration of the growing incomparable points number (5)	26
I.6	Illustration of the growing incomparable points number (6)	27
I.7	Illustration of the growing incomparable points number (7)	27
I.8	Illustration of the growing incomparable points number (8)	28
I.9	Illustration of the growing incomparable points number (9)	28
I.10	Illustration of the growing incomparable points number (10)	29
I.11	Illustration of the growing incomparable points number (11)	29

Chapter 1 Introduction

In cartographic literature, the main approach to letting people know what is on the map is attaching texts or labels to geographic features on the map. Automated label placement subject to the constraint that the labels are pairwise disjoint is a well-known important problem in geographic information systems (GIS). In the ACM Computational Geometry Impact Task Force report [1] the map label placement is listed as an important research area. Since this problem in general is known to be NP-complete, many heuristics or special cases for which polynomial time algorithms are given have been presented. For instance, there are many algorithms that have been developed for labeling points that are on lines [2–6] or in a region [7–16].

Let \mathcal{A} denote a set of points $\{A_1, A_2, \dots, A_n\}$ in the plane, called *anchors*. Associated with each anchor there is an axis-parallel rectangle, called *label*. The *point-feature label placement problem* or simply *point labeling problem*, is to determine a placement of these labels such that the anchors coincide with one of the corners of their associated labels and no two labels overlap. The point labeling problem for labeling an arbitrary set of points has been shown to be NP-complete [8,10,11,14] and some heuristic algorithms were presented in [8,16,17].

There are many variations of the point labeling problem, including shapes of the labels, locations of the anchors to be labeled and where the labels are placed. Consider the case that the placement of the labels are restricted. For instance, one is *fixed-position model*, denoted 4P model, in which a label must be placed so that the anchor coincides with one of its four corners; and another is *slider model*, denoted 4S model, in which a label can be placed so that the anchor lies on one of the four boundary edges of the label. Figure 1.1 shows these two point labeling models. The positions $\{1, 2, 3, 4\}$ in 4P model shown in Figure 1.1 denote the corner positions of labels coincident with the anchor, and the arrows in 4S model indicate the directions along which the label can slide, maintaining contact with the anchor. Moreover, if we restrict the number of fixed positions to be 2, and horizontal or vertical direction in which fixed-position labels are arranged, the problem will be denoted 2PH or 2PV model respectively.

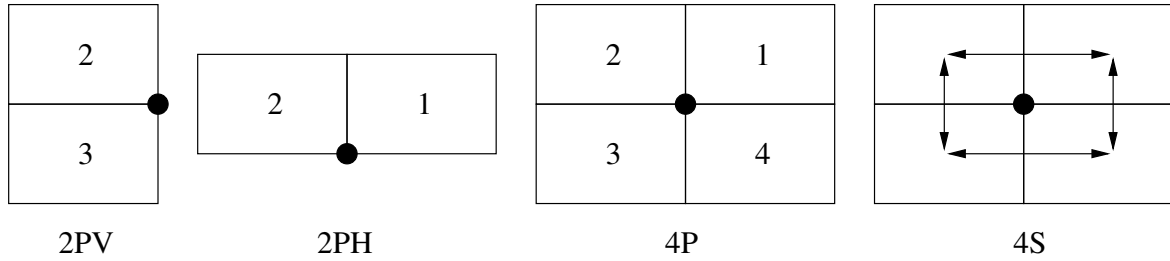


Figure 1.1: Illustration of 2PV, 2PH, 4P, and 4S models.

In this paper we consider the case when the anchors lie on a line and are to be labeled with rectangular labels. This problem has been studied previously [2,3,6,12]. The prefix 1d or Slope refers to the problem in which the anchors lie on a horizontal or a sloping line, respectively. Garrido et al. [3] gave linear time algorithms for 1d4P rectangle label, 1d4S square label, and Slope4P square label models, and a quadratic time algorithm for Slope4S square label model as well. They also showed 1d4S rectangle label is NP-complete and consider the maximization version to maximize the size of labels. Chen et al. [2] further provided linear time algorithms for Slope4P fixed height(or width) rectangle label and elastic rectangular label (of a given area) models. They also presented a lower bound $\Omega(n \log n)$ time and a different method to maximize the label size for 1d4S square label model. In addition, in order to produce map labeling of maximum legibility in automatic label placement, certain optimization versions of the problem were considered. The two most common optimization criteria are the number of labels and the size of labels, for which feasible solutions can be obtained. In 1998, Agarwal et al. [18] provided a PTAS $(1 + 1/k)$ algorithm in $O(n \log n + n^{2k-1})$ time, for any integer $k \geq 1$, for 4P fixed-height rectangle label model and an $O(\log n)$ factor approximation algorithm in $O(n \log n)$ time for 4P arbitrary rectangle labels. Poon et al. [6] further considered the weighted case in which each label is associated with a given weight and provided the same approximation result for 4P fixed-height weighted rectangle model. They also gave a $(2 + \epsilon)$ factor approximation algorithm in $O(n^2/\epsilon)$ time for 1d4S weighted rectangle label. As for 4P arbitrary rectangle label, Berman et al. [19] presented a $\lceil O(\log_k n) \rceil$ factor approximation algorithm in $O(n^{k+1})$ time, for any integer $k \geq 2$. In 2004, Chan [20] improved the previous results to give a PTAS $(1 + 1/k)$ algorithm in $O(n \log n + n\Delta^{k-1})$ time, where $\Delta \leq n$ denotes the maximum number of rectangles whose intersection is nonempty, for 4P fixed-height rectangle label model and a $\lceil O(\log_k n) \rceil$ factor approximation algorithm

in $n^{O(k/\log k)}$ time for 4P arbitrary rectangle label.

We investigate the maximization version of the feasible number of labels when the anchors lie on a horizontal line. That is, we want to maximize the number of labels for which a *feasible* placement exists that no two labels overlap. As a contrast to the previous results in which they all employed the *line stabbing* technique, we describe a different approach to solving this model, referred to as *Max-1d4P rectangle label model*, or *Max-1d4P* for short, in $O(n \log n)$ time. This paper is organized as follows. In Section 2, we introduce some definitions and our **R-map** method. Then we present in Section 3 an intuitive method to solve Max-1d2PH, Max-1d2PV models and point out the difference between Max-1d2P and Max-1d4P models. We also consider the intricate Max-1d4P model and provide an $O(n \log n)$ time algorithm. Then we extend it to an $O(kn \log kn)$ solution for Max-1d2kP model, which is a generalization of Max-1d4P model. Finally we conclude in Section 4 with some discussions of future work.



Chapter 2 Preliminaries

Consider a set of anchors $\mathcal{A} = \{A_1, A_2, \dots, A_n\}$ on a horizontal line, and each anchor A_k is associated with its position (in x-coordinate) x_k and label size l_k . The aim is to maximize the number of *feasible* labels so that they do not overlap with each other.

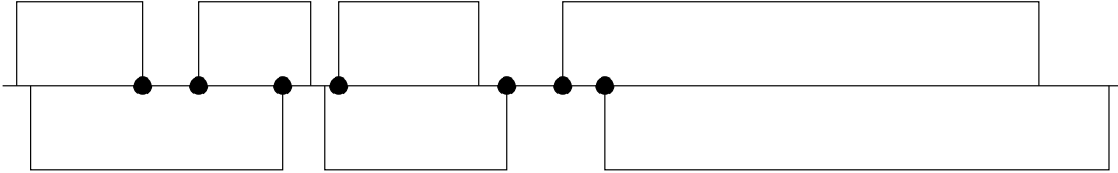


Figure 2.1: Example of maximizing the number of labeled points.

Since we consider the problem on a horizontal line and put the label either *above* or *below* the line, we can simply associate a 2-tuple, namely (a, b) , to represent the current labeling state of a realization R , with $R.a = a$ and $R.b = b$. The first variable, a , shows the coordinate of right edge of the rightmost label *above* the line; the second variable, b , shows the rightmost one *below* the line. A realization R will also contain a specification of the placement of feasible labels associated with a subset of anchors. To be more precise, we can use $A_i.\ell \in R$, where $\ell \in \{0, 1, 2, 3, 4\}$ indicates the label position for anchor A_i included in R , with $\ell = 0$ representing anchor A_i is not labeled. If R contains k -feasible labels, i.e., it contains k non-zero ℓ 's associated with $A_i.\ell$, then R is called a k -realization, and we use $R.c$ to denote the cardinality of the subset of feasible labels represented by the realization R . We shall use the notation R to represent not only a realization R of \mathcal{A} , which is a subset of feasible anchors, i.e., $R \subseteq \mathcal{A}$, but also use $R.a$, $R.b$ and $R.c$ to represent the state of its configuration and its size, respectively. Let us assume that the set of anchors has been ordered so that their x -coordinates are in strictly increasing order. That is, $x_1 < x_2 < \dots < x_n$. Let \mathcal{A}_i denote the subset of anchors $\{A_1, A_2, \dots, A_i\}$, for $i = 1, 2, \dots, n$, and R^i denote a realization of \mathcal{A}_i for some i . An optimal solution is a realization R^n such that $R^n.c$ is maximum among all possible realizations of \mathcal{A}_n .

We shall process the anchors, and their associated labels, in ascending order of their x -coordinates, i.e., in the order of A_1, A_2, \dots, A_n . Given a realization R^{i-1} of \mathcal{A}_{i-1} , and the next anchor, A_i , $i > 1$, the placements of the label of A_i that do not overlap the last

label both above and below the line in R^{i-1} are called *feasible label placements*. Before proceeding we define the notion of *equivalence* of two realizations:

Definition 2.0.1 Given two realizations R_1^i and R_2^i of \mathcal{A}_i such that $R_1^i.c$ and $R_2^i.c$ are equal, if

$$\{R_1^i.a, R_1^i.b\} = \{R_2^i.a, R_2^i.b\}$$

we say that the two realizations are **equivalent in size**, or simply **equivalent** to each other.

Based on the above definition, for a realization R^i with $R^i.a < R^i.b$, we always swap the upper and lower sides of the realization. That is, a realization will be represented in a *normal form* in which the coordinate *above* the line is no less than the coordinate *below* the line, i.e., $R^i.a \geq R^i.b$ without loss of generality. Here we define the *comparability* of two realizations.



Definition 2.0.2 For any two realizations R_i^k and R_j^k , $k = 1, 2, \dots, n$, if the following statements hold,

1. $R_i^k.c = R_j^k.c$, and
2. $R_i^k.a \leq R_j^k.a$, and $R_i^k.b \leq R_j^k.b$

then we say that the two realizations are **comparable** and R_i^k is **better** than R_j^k . Otherwise, they are **incomparable**.

Lemma 2.0.3 Let \mathcal{R} be an optimal realization of \mathcal{A} . Suppose R_i^k and R_j^k are two comparable realizations for \mathcal{A}_k and R_i^k is better than R_j^k for some $n \geq k \geq 1$. If \mathcal{R} contains R_j^k as a subset, then there exists another optimal solution that contains R_i^k .

Proof: Since $R_i^k.a \leq R_j^k.a$ and $R_i^k.b \leq R_j^k.b$, we can obtain \mathcal{R}' by replacing R_j^k in \mathcal{R} with R_i^k , so that $\mathcal{R}'.c = \mathcal{R}.c$. □

By using a 2-tuple to represent the labeling state of a realization, we can transform it into a point in the 2-dimensional plane, which we call an **R-map**. To be more precise, given a realization R represented by a 2-tuple $(R.a, R.b)$, we transform it into a point (x,y) , where $x = R.a$ and $y = R.b$, in the plane. The two equivalent realizations will then be transformed into two points that are symmetric with respect to the line $x = y$. From now on, we use $P.x$ and $P.y$ to represent the x and y coordinates of a point P on the R-map, and $P.c$ to represent its associated cardinality.

We assume the point labeling on a line starts at the origin without loss of generality, which means the transformed points on the R-map are all in the first quadrant. Using the normal form representation of a realization, all realizations will be mapped to points that are all located in the first quadrant below the line $x = y$. More specifically, it is the area generated by the following equations: $x \geq y$, and $y \geq 0$.

Based on the *comparability* definition between two realizations, if one is better than the other, then the transformed points will carry the relationship of *domination*. That is, if realization R_A is *better* than realization R_B , then point P_B *dominates*¹ point P_A on the R-map. On the other hand, if they are *incomparable* realizations, the transformed points on the R-map do not dominate each other.

We shall also transform each anchor with its given label from the original representation into another term on the R-map. We use a **cross** on the R-map to stand for an anchor with its label, and the length of each arrow shows its given label size. The center of every cross is located on line $x = y$. The arrows are labeled from 1 to 4 for left, bottom, top and right, corresponding to the numbering of original labels respectively. The label overlapping situation is then transformed into arrows intersecting either the vertical line passing through the x -coordinate or the horizontal line passing through y -coordinate of a point on the **R-map**. We define the operations on the R-map as follows.

Definition 2.0.4 Given a point $p(x, y)$ representing a realization and a cross C representing a new label i on the R-map, whose center is $(C.x, C.y)$ and arrow length is $C.l$, we have the following operations depending on how we select the placement of label i of

¹If $P_B.x \geq P_A.x$, and $P_B.y \geq P_A.y$, then P_B is said to *dominate* P_A .

the anchor c for the realization p (i.e. apply arrow i)

1. $i = 1$, then $p(x, y)$ generates $p'(C.x + C.l, y)$

2. $i = 2$, then $p(x, y)$ generates $p'(C.x, y)$

3. $i = 3$, then $p(x, y)$ generates $p'(x, C.y)$

4. $i = 4$, then $p(x, y)$ generates $p'(x, C.y + C.l)$

The cardinality associated with point p' will be one more than that with point p 's cardinality. If the y -coordinate of p' is greater than the x -coordinate, we do the swapping operation to exchange the x - and y - coordinates. We call point p the **parent point** of p' and the generated point p' the **child point** of p .

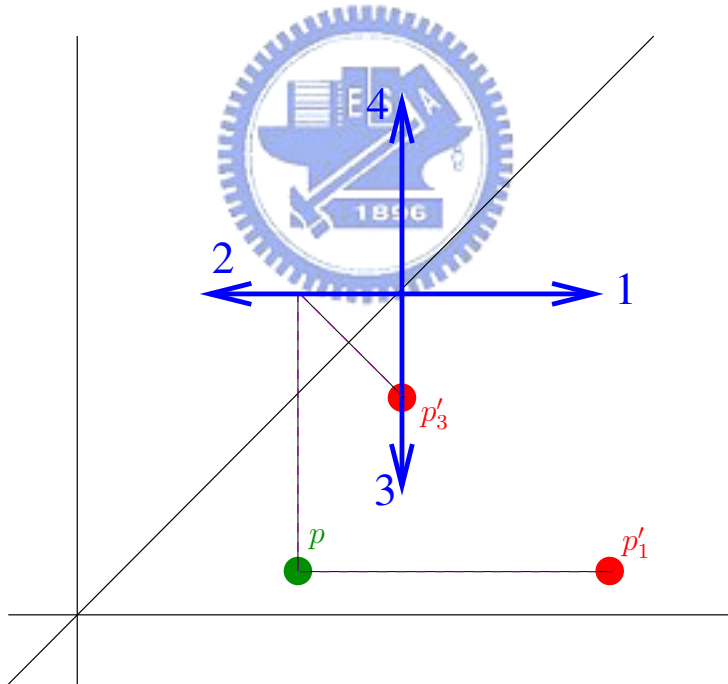


Figure 2.2: Example of the R-map.

Take Figure 2.2 as an example. Given a parent point $p(x, y)$ and a cross located at $(C.x, C.y)$ with arrow length $C.l$, arrow 2 of the cross C represents an infeasible placement with respect to point p . Applying arrows 1 and 3, point p generates two child points $p'_1(C.x + C.l, y)$ and $p'_3(C.x, x)$ respectively. The following property is immediate.

Property 2.0.5 Given a parent point $p(x, y)$ and a cross C located at $(C.x, C.y)$ with arrow length $C.l$,

1. If $y > C.y$, the point can apply no arrow.
2. If $C.y \geq y > C.y - C.l$, the point can apply arrow 4.
3. If $C.y - C.l \geq y$, the point can apply arrows 3 and 4.
4. If $x > C.x$, the point can apply neither arrow 1 nor arrow 2.
5. If $C.x \geq x > C.x - C.l$, the point can apply arrow 1.
6. If $C.x - C.l \geq x$, the point can apply arrows 1 and 2.

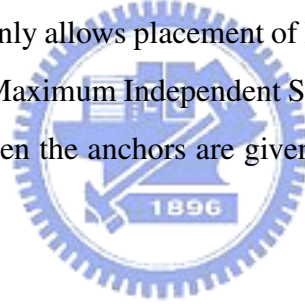


Chapter 3 Maximum Independent Set of Label Placements for Different Models

We adopt a greedy method to solve the problem for each model, namely, we process the anchors in sequential manner, and will maintain a set of realizations that reflect the best possible labeling, ignoring those that are known to be no better than the present set of realizations, after each anchor is processed.

3.1 1d2PH, 1d2PV

For the *1d2PH* model, which only allows placement of labels 1 and 2, the problem can be transformed to the problem of Maximum Independent Set (MIS) of Interval Graph, which can be solved in $O(n)$ time when the anchors are given sorted. The following lemma is obvious.



Lemma 3.1.1 *Given two realizations R_A^i and R_B^i , for $i = 1, 2, \dots, n$, we will select R_A^i over R_B^i , either if $R_A^i.c > R_B^i.c$, or if $R_A^i.c = R_B^i.c$, and R_A^i is better than R_B^i .*

Lemma 3.1.2 *Given a realization R^{i-1} , when both placements of labels 2 and 3 of next anchor A_i are feasible, the selection of label 2, above the line will yield a better realization R^i .*

Proof: Given a current realization (a, b) with the normal form representation (i.e., $a > b$) and the anchor position x , placing label at x either above or below the line changes the labeling state into (x, b) or (a, x) respectively. Since $a > b$, placing the label above the line generates a better realization. \square

Here we give a greedy algorithm for the problem *1d2PV*, which only allows labels 2 and 3 without loss of generality.

Algorithm 3.1.1 M2PV

Input: A set of anchors sorted by their x-coordinate and their associated labels.

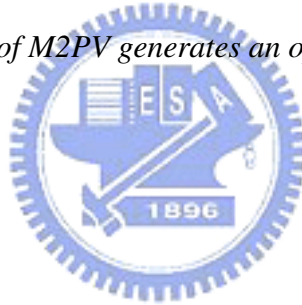
Output: The maximum cardinality of map labeling for model 1d2PV.

- 1: **while** pick up the next anchor from the sorted list **do**
 - 2: **if** the placement of label 2 is feasible **then**
 - 3: place label 2 and put the anchor into MIS solution, cardinality++
 - 4: **else if** the placement of label 3 is feasible **then**
 - 5: place label 3 and put the anchor into MIS solution, cardinality++
 - 6: **else**
 - 7: discard the anchor.
 - 8: **return** cardinality
-

The time complexity of M2PV is $O(n)$ given the sorted anchors.

Theorem 3.1.3 *The algorithm of M2PV generates an optimal solution.*

Proof: Immediate. □



3.2 1d4P

Now let us consider model **1d4P**. Given an anchor whose feasible labels are label 1, 3 and 4, placing either label 1 or label 3 may generate an optimal solution. We cannot tell which placement is better at the time when we process the anchor. Figure 3.1(a) shows that the choice between the placements of labels 1 and 3 is best decided by future anchors. The top realization shown in Figure 3.1(a) shows the placement of label 1 of the anchor leads to an optimal solution. However, the bottom realization shown in Figure 3.1(a) shows the placement of label 3 of the anchor leads to an optimal solution. Furthermore, given an anchor whose feasible labels are labels 1 and 4, we cannot immediately decide whether placing either one can generate an optimal solution or not. Figure 3.1(b) shows the placement of label 1 may block the future labels. That is, we may need to postpone the decision as to whether the current anchor is to be included in the realization until

later. The condition also occurs for label 4. However, if we must choose one placement between label 1 and label 4 of an anchor, then following Lemma 3.1.2, the choice of the label above the line (label 1) gives a better realization. We first observe some properties of model $1d4P$.

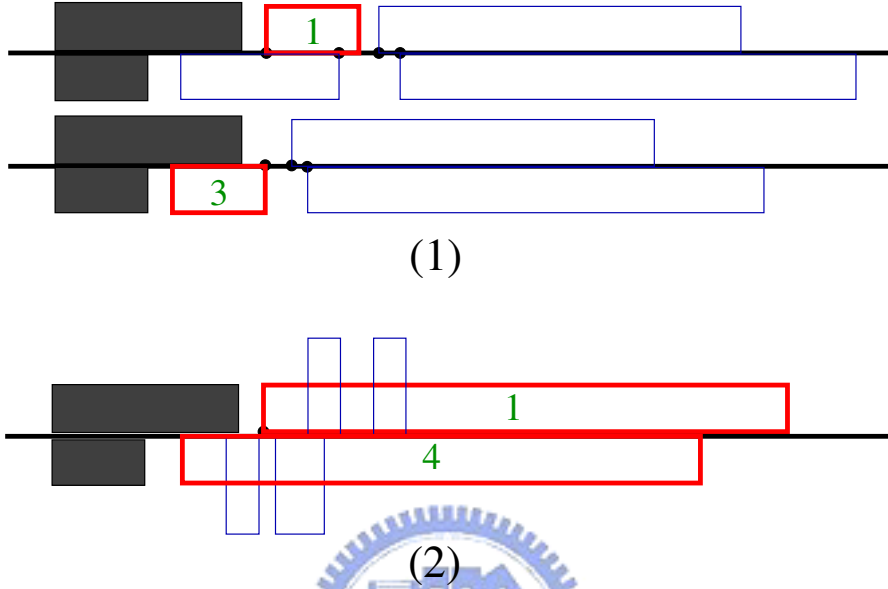


Figure 3.1: (a) Example of incomparability between realizations resulting from placing labels 1 and 3. (b) Example of undecidable situation as to whether we should choose to include label 1 or not.

Proposition 3.2.1 *Given a realization R^{i-1} , when both placements of labels 1 and 2 (respectively, labels 3 and 4) of next anchor A_i are feasible, the selection of label 2 (respectively, label 3) will yield a better realization R^i .*

Lemma 3.2.2 *Given a realization R^{i-1} , when the placement of label 3 of next anchor A_i is feasible, the anchor is included in the MIS solution.*

Proof: Suppose the current realization is (a, b) with $a > b$ and the next anchor A_i whose placement at label 3 is feasible. Assume that there is an optimal solution such that it does not contain A_i . It is trivial that if placing label 1 or label 3 of A_i into the solution overlaps no labels, then we can place one more label into the solution to get a larger cardinality, which leads to a contradiction. Otherwise, there is an anchor A_j , $j > i$. i.e, A_j is to the

right of A_i , whose label 3 overlaps the label 3 of A_i . Since A_j is to the right of A_i , we can replace label 3 of A_j by the placement of label 3 of A_i to get a better realization. \square

Corollary 3.2.3 *Given a realization R^{i-1} , when the placement of label 2 of next anchor p_i is feasible, the anchor is in the MIS solution.*

We introduce our main idea as follows. Let $S[i, j]$ denote a set of incomparable realizations R^j of cardinality i , for $1 \leq i \leq j \leq n$. We shall apply a dynamic programming method to process the anchors and record the 'better' realizations of each possible cardinality. To find an optimal realization R^n , we may need to maintain intermediate realizations $S[i, j]$ for $1 \leq i \leq j \leq n$, that have the potential leading to an optimal realization. As we shall show later, for each $j \leq n$ we only need to maintain at most *five* subsets $S[k, j]$, $S[k + 1, j]$, $S[k + 2, j]$, $S[k + 3, j]$, and $S[k + 4, j]$ for some k . We shall process the table from $j = 1$ till n and fill each entry $S[i, j]$ with a set of incomparable realizations, and there are probably quite a few incomparable realizations to be maintained at each step.

The realizations in an incomparable set form a "point chain" on the R-map without having any point in the chain dominate another. When we encounter a new *cross*, some of the point in this chain of cardinality k , for some k , will generate new child points, thus getting *upgraded* to a realization of cardinality $k + 1$, some will remain as *non-upgraded* with cardinality k , and are kept as potential candidates leading possibly to an optimal solution, and some get eliminated due to some new child points upgraded from points of cardinality $k - 1$. At the end after anchor A_n is processed, the realizations in the non-empty entry $S[i, n]$ with the largest i are optimal solutions.

To sum up, some points in the set $S[i, j]$ may simply move to $S[i, j + 1]$ without increasing cardinality, following what we call a *non-upgrading process*. Other points in the set may generate points which are included in $S[i + 1, j + 1]$, whose cardinality is incremented, following what we call an *upgrading process*. When a point moves from one entry to another, it should be compared with other points in the target entry, and only *better* ones are kept. We repeat such operations until we have processed all anchors. The following is the algorithm of model 1d4P.

Algorithm 3.2.1 M4P

Input: A set of anchors sorted by their x-coordinates and their associated labels.

Output: A maximum cardinality of map labeling for model 1d4P.

- 1: Use dynamic programming method on two parameters $S[i, j]$: the anchor ordering in column and the cardinality of possible solutions in row. Initialize the first entry by the placement of label 2 of the first anchor.
 - 2: **for** $j = 2$ **to** n **do**
 - 3: Let the largest cardinality of non-empty entries in column $j - 1$ be k .
 - 4: **for** $i = k$ **to** $\max\{k - 4, 0\}$ **do**
 - 5: 2-1. Classify the points in $S[i, j - 1]$ into upgrading and non-upgrading classes.
 - 6: 2-2. Move the non-upgraded points into $S[i, j]$.
 - 7: 2-3. Move the upgraded points into $S[i + 1, j]$.
 - 8: 2-4. Compare the points in $S[i + 1, j]$ and keep the better ones.
 - 9: **return** The maximum cardinality of non-empty entries.
-

In what follows we will prove a few results that help establish the correctness of our algorithm. Let P_A and P_B be two points on the R-map, and $(C.x, C.y)$ be the coordinates of the next cross.

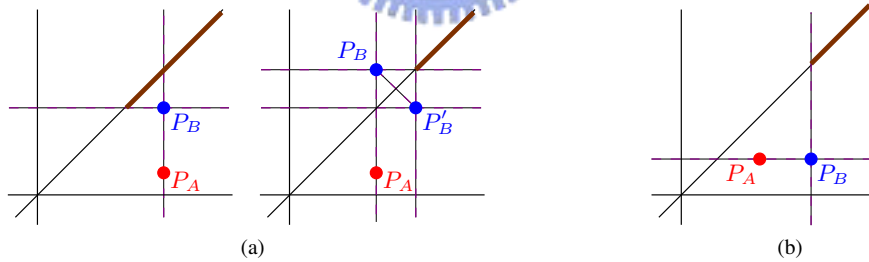


Figure 3.2: Illustration of Lemma 3.2.4.

Lemma 3.2.4 For the following two cases

$$(a) P_A.x \geq P_B.x, P_A.y < P_B.y, P_A.c < P_B.c \text{ and } C.y \geq P_B.y$$

$$(b) P_A.y \geq P_B.y, P_A.x < P_B.x, P_A.c < P_B.c \text{ and } C.x \geq P_B.x$$

P_B is better than P_A .

Proof:

(a) Since $C.y \geq P_B.y > P_A.y$, applying the cross, P_A will generate a point P_{new} where $P_{new}.x \geq P_A.x \geq P_B.x$ and $P_{new}.y \geq C.y \geq P_B.y$ and $P_{new}.c = P_A.c + 1 \leq P_B.c$. That is, P_A cannot generate any child point which is better than P_B , so P_A need not be kept.

(b) Since $C.x \geq P_B.x > P_A.x$, applying the cross, P_A will generate a point P_{new} where $P_{new}.x \geq C.x \geq P_B.x$ and $P_{new}.y \geq P_A.y \geq P_B.y$ and $P_{new}.c = P_A.c + 1 \leq P_B.c$. That is, P_A cannot generate any child point which is better than P_B , so P_A need not be kept. \square

Lemma 3.2.5 *The points in an incomparable set going through an upgrading process generates at most two incomparable child points.*

Proof: By the definition of operations on the R-map, we know that after applying a cross C of the center $(C.x, C.y)$, the child points will be located on one of the following four lines:

1. $x = C.x$
2. $x = C.x + C.l$
3. $y = C.y$
4. $y = C.y + C.l$



Moreover, since we represent a realization in a normal form, such four lines can be expressed more specifically as:

1. $x = C.x, y \leq C.y$
2. $x = C.x + C.l, y \leq C.y + C.l$
3. $y = C.y, x > C.x$
4. $y = C.y + C.l, x > C.x + C.l$

It is then easy to see that at most two incomparable points can be found on these four lines. \square

Theorem 3.2.6 *The number of points in an incomparable set is bounded by $O(n)$.*

Proof: By Lemma 3.2.5, there are at most two points added to an entry while processing every cross. After processing n crosses, the number of points in an entry is bounded by

$O(n)$.

□

Theorem 3.2.7 *After processing a cross C , if there is a point with cardinality k which is the ancestor of a point with cardinality $k + 5$, then no point with cardinality k will lead to an optimal solution. That is, the difference in cardinality of incomparable points is at most four.*

Proof: Let point $P(x, y)$ whose associated cardinality is k be the ancestor of a point with cardinality $k + 5$. For any point $Q(x', y')$ whose associated cardinality is k , we have the following two cases to consider, $y' \leq y$ and $y' > y$.

Case 1: $y' \leq y$. Consider the three crosses that were applied to upgrade point P to a point P' of cardinality $k + 3$. Among these three crosses, at least two labels, say C_i and $C_j, j > i$, must be placed on the same side of the line. It is clear that the anchor corresponding to the next cross C lies totally to the right of the label associated with C_i . Since $Q(x', y')$ is an arbitrary point of cardinality k with $y' \leq y$, we know that cross C_i can be applied to point Q . Then point Q' upgraded by C_i is always better than point Q , by Lemma 3.2.4(a). Thus point Q need not be kept, which is to say, any point with cardinality k whose y -coordinate is no more than y need not be retained. That is, when a point $P(x, y)$ of cardinality k got upgraded by three to a point P' of cardinality $k + 3$, those points $Q(x', y')$ of the same cardinality $k, y' \leq y$ would be upgraded by at least one to Q' of cardinality $k + 1$ and Q' always better than Q . And therefore the difference in cardinality of these points of cardinality k , when one of them got upgraded by three to P' of cardinality $k + 3$, is at most two.

Case 2: $y' > y$. Without loss of generality let us consider an arbitrary point Q with $x' < x$ and $y' > y$. Among the five crosses that have been applied to point P , at least three labels, say those associated with crosses C_i, C_j , and $C_k, k > j > i$, must be placed on the same side of the line, and the next cross C lies totally to the right of labels associated with C_i and C_j . If C_i is placed above the line, it is obvious that cross C_i can also be applied to upgrade Q , since $x' < x$. By Lemma 3.2.4(b), point Q' upgraded by C_i is always better than point Q , and hence Q of cardinality k need

not be kept. Suppose all these three labels are placed below the line. Without loss of generality, let us take the leftmost three crosses, C_i, C_j , and $C_k, k > j > i$. Let P' denote the child point upgraded from point P after cross C_j is applied, and let $P'.y$ denote the right side of the label associated with C_j . If $P'.y \geq y'$, then cross C_k and the other two crosses, will play the role of the three crosses, P' will play the role of P , as in **Case 1** above, then as shown before, we no longer need to keep Q (of cardinality k). Otherwise, ($P'.y < y'$) consider the the cross C_h corresponding to the last label below the line associated with point Q . We know that C_h (or anchor A_h) is to the left of C_i (or anchor A_i). Since the label associated with C_j is placed totally to the right of the label associated with C_i , we can replace the last label associated with C_h in Q by the label associated with C_j to obtain a new point Q' , so that $Q'.c = Q.c = k$, and $Q'.y = P'.y < y'$. This implies that Q' is better than Q after the replacement, and Q would have been eliminated after C_i and C_j were considered. Then when C_k is applied to P' , it can also be applied to point Q' . Since cross C is totally to the right of the label associated with C_k , by Lemma 3.2.4(a), point Q' (which is better than Q) upgraded by C_k is always better than point Q . Thus, we no longer need to keep point Q . That is, when a point $P(x, y)$ of cardinality k got upgraded by five to a point of cardinality $k + 5$, those points $Q(x', y')$ of the same cardinality $k, y' > y$ would be upgraded by at least one to a point of cardinality $k + 1$ and always better than Q which remain non-upgraded, and therefore the difference in cardinality of these points of cardinality k , when one of them got upgraded by five to a point of cardinality $k + 5$, is at most four.

To sum up, any point with cardinality k which gets upgraded to a point with cardinality $k + 5$, will force all the points which remain non-upgraded to have cardinality k to be subsumed by other upgraded points. Therefore no point with cardinality k will lead to an optimal solution and the difference in cardinality of these incomparable points is at most four. \square

By Theorem 3.2.7, in computing $S[i, j]$ for $1 \leq i \leq j \leq n$, it is sufficient to maintain at most five consecutive sets of incomparable realizations $S[k, *], S[k + 1, *], S[k + 2, *], S[k + 3, *]$ and $S[k + 4, *]$.

Lemma 3.2.8 *The following operations all take $O(\log n)$ time.*

- (1) *Classifying points into the upgrading and non-upgrading classes.*
- (2) *Finding two incomparable points among all upgraded points.*
- (3) *The comparison between one point and n incomparable points.*

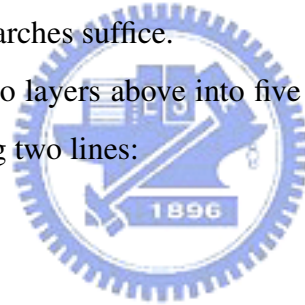
Proof: (1) When we process a cross C centered at $(C.x, C.y)$, we can simply divide the set of points of our R-map into three *layers* by two lines below:

1. $y = C.y$
2. $y = C.y - C.l$

The points P in the top layer with $P.y \geq C.y$ belong to non-upgrading class. The points P in the middle layer with $C.y \geq P.y \geq C.y - C.l$ belong to both upgrading and non-upgrading classes. The points P in the bottom layer with $C.y - C.l \geq P.y$ belong to the upgrading class. Since the set of incomparable points can be maintained in sorted order in y -coordinates, two binary searches suffice.

(2) We can divide the lower two layers above into five regions (R_1, R_2, R_3, R_4 and R_5 , see Figure 3.3) by the following two lines:

1. $x = C.x - C.l$
2. $x = C.x$



By the operations on the R-map, we know that the following points in the chain of upgraded points are better than all the other points (in the chain) in each of their regions after the upgrading process:

1. the rightmost point of the chain in R_1
2. the leftmost point of the chain in R_2
3. the leftmost point of the chain in R_3
4. the rightmost point of the chain in R_4
5. the leftmost point of the chain in R_5

We can also use binary search four times to obtain the five points (at most three points in a point chain actually). Finding the two incomparable points out of these points takes constant time.

(3) Since an incomparable set is a point chain without any point dominating another, we need only to find out all points in the chain that dominate p when we have to compare a point p with such a point chain. We can simply do binary search in the chain two times

to divide the chain into three sub-chains of consecutive points and the middle sub-chain which dominates p can then be eliminated and replaced by p . \square

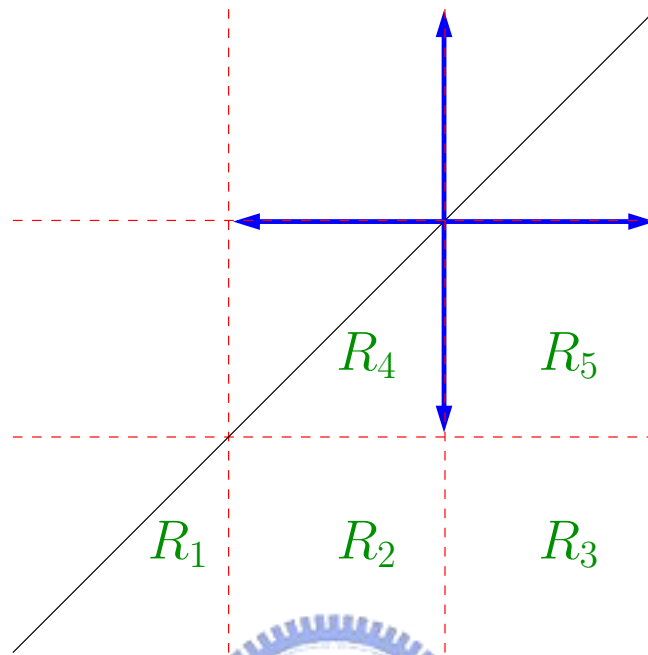


Figure 3.3: Proof for Lemma 3.2.8 (b).

Theorem 3.2.9 *The time complexity of Algorithm M4P is $O(n \log n)$.*

Proof: By Theorem 3.2.7, the inner for loop repeats at most five times. By Lemma 3.2.8, the tasks in the inner for loop can be completed in $O(\log n)$ time. The outer for loop repeats n times to handle n anchors. Then the total time complexity for solving model 1d4P is $O(n \log n)$. \square

When the algorithm terminates, any point in $S[k, n]$ with the largest k is an optimal solution (of maximum cardinality k). The actual placement of labels can be obtained if we record the processing history when a point is upgraded.

3.3 1d2kP

The 1d2kP model is a generalization of 1d4P model. It allows the label of each anchor to be placed at k positions, $k \geq 2$, both above and below the line. Poon et al., Strijk and van Kreveld, Kim et al. studied the decision k -position models [13,21,22]. We consider the maximization of 1d2kP problem. Due to the definition of k -position model, the incomparable points in an entry could be up to $O(kn)$, therefore the time complexity of classifying points in Lemma 3.2.8 (1) is $O(\log kn)$. The number of lines in Lemma 3.2.5 is also modified by k , and thus the number of incomparable points of upgrading process in Lemma 3.2.5 would be k . Therefore the time complexity of Lemma 3.2.8 (2) is $O(k \log kn)$, and the comparing time between k points and kn points would also be $O(k \log kn)$ in Lemma 3.2.8 (3). The overall time complexity of solving model 1d2kP would be $O(kn \log kn)$.



Chapter 4 Conclusion Remarks

We have extended the decision version of the map labeling problem on a horizontal line to an optimization version where the number of feasible labels is to be maximized. It is a variation of maximum independent set problem on interval graphs. It can also be regarded as a simplified version of job-machine scheduling problem (two machines, each job has k time intervals to be selected in its given time range) in which the number of schedulable jobs is to be maximized. We present an $O(n \log n)$ time algorithm for Max-1d4P model by dynamic programming on two parameters: the anchor ordering and the cardinality of possible solutions.

We have proposed the R-map idea which can be useful for quite a few problems, including many versions of job scheduling problems. Moreover, we feel that it may be used to improve the approximation ratio for general Max-4P problem when combining the line stabbing technique and our solution. Whether there exist solution for the Max-Slope4P fixed height(or width) rectangle label model remains to be seen.



Bibliography

- [1] B. Chazelle and . co-authors. The computational geometry impact task force report (407-463), *Advances in Discrete and Computational Geometry*, vol. 223. American Mathematical Society, Providence, 1999.
- [2] D. T. L. Yu-Shin Chen and C.-S. Liao, “Labeling points on a single line,” *International Journal of Computational Geometry and Applications (IJCGA)*, vol. 15, no. 3, pp. 261–277, 2005.
- [3] A. M. J. R. P. P. R. M. Á. Garrido, C. Iturriaga and A. Wolff, “Labeling subway lines,” in *Proc. 12th Annual International Symposium on Algorithms and Computation (ISAAC’01)*, vol. volume 2223 of Lecture Notes in Computer Science, pp. 649–659, 2001.
- [4] C. Iturriaga and A. Lubiw, “Elastic labels: The two-axis case,” in *Proceedings of the Symposium on Graph Drawing (GD’97)*, vol. volume 1353 of Lecture Notes in Computer Science, pp. 181–192, 1997.
- [5] C. Iturriaga and A. Lubiw, “Elastic labels around the perimeter of a map,” *Journal of Algorithms*, vol. 47, no. 1, pp. 14–39, 2003.
- [6] T. S. T. U. S.-H. Poon, C.-S. Shin and A. Wolff, “Labeling points with weights,” *Algorithmica*, vol. 38, no. 2, pp. 341–362, 2003.
- [7] A. V. R. Duncan, J. Qian and B. Zhu, “Polynomial time algorithms for three-label point labeling,” *Theoretical Computer Science*, vol. 296, no. 1, pp. 75–87, 2003.
- [8] M. Formann and F. Wagner, “A packing problem with applications in lettering of maps,” in *Proceedings of the 7th ACM Symposium on Computational Geometry*, pp. 281–288, 1991.
- [9] Z. Q. B. Z. M. Jiang, J. Qian and R. Cimikowski, “A simple factor-3 approximation for labeling points with circles,” in *Inform. Process. Letters*, vol. 87(2), pp. 101–105, 2003.

- [10] T. Kato and H. Imai, “The np-completeness of the character placement problem of 2 or 3 degrees of freedom,” in *In Record of Joint Conference of Electrical and Electronic engineers in Kyushu.*, p. 1138, 1988.
- [11] D. Knuth and A. Raghunathan, “The problem of compatible representatives,” in *SIAM Disc. Math.*, vol. 5(3), pp. 422–427, 1992.
- [12] T. S. M. van Kreveld and A. Wolff, “Point labeling with sliding labels,” *Computational Geometry: Theory and Applications*, vol. 13, pp. 21–47, 1999.
- [13] C.-S. S. S. K. Kim and T.-C. Yang, “Labeling a rectilinear map with sliding labels,” *International Journal of Computational Geometry and Applications*, vol. 11, no. 2, pp. 167–179, 2001.
- [14] J. Marks and S. Shieber, *The computational complexity of cartographic label placement*, Technical Report TR-05-91. Harvard University CS, 1991.
- [15] T. Strijk and A. Wolff, “Labeling points with circles,” *International Journal of Computational Geometry and Applications*, vol. 11, pp. 181–195, April 2001.
- [16] F. Wagner and A. Wolff, “A practical map labeling algorithm,” *Computational Geometry: Theory and Applications*, vol. 7, pp. 387–404, 1997.
- [17] J. M. J. Christensen and S. Shieber, “An empirical study of algorithms for point feature label placement,” in *ACM Transactions on Graphics*, vol. 14(3), pp. 203–232, 1995.
- [18] M. v. K. Pankaj K. Agarwal and S. Suri, “Label placement by maximum independent set in rectangles,” in *Computational Geometry: Theory and Applications 11*, pp. 209–218, 1998.
- [19] S. M. S. R. P. Berman, B. DasGupta, “Efficient approximation algorithms for tiling and packing problems with rectangles,” *Journal of Algorithms*, vol. 41, pp. 443–470, 2001.
- [20] T. M. Chan, “A note on maximum independent sets in rectangle intersection graphs,” in *Inform. Process. Letters*, vol. 89, pp. 19–23, 2004.

- [21] B. Z. C.K. Poon and F. Chin, “A polynomial time solution for labeling a rectilinear map,” in *Information Processing Letters*, vol. 65(4), pp. 201–207, 1998.
- [22] T. Strijk and M. van Kreveld, “Labeling a rectilinear map more efficiently,” in *Information Processing Letters*, vol. 69(1), pp. 25–30, 1999.



Appendix I : Example of R-map

The following are the illustrations of the growing incomparable points number.

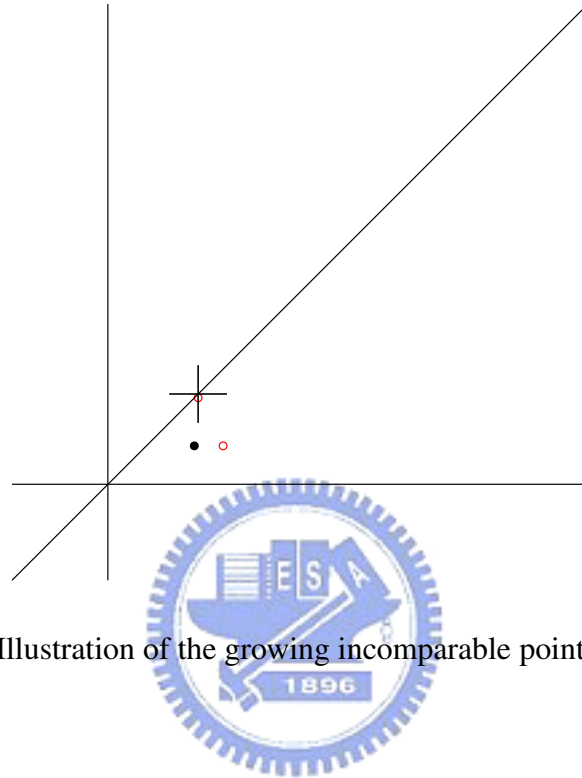


Figure I.1: Illustration of the growing incomparable points number (1)

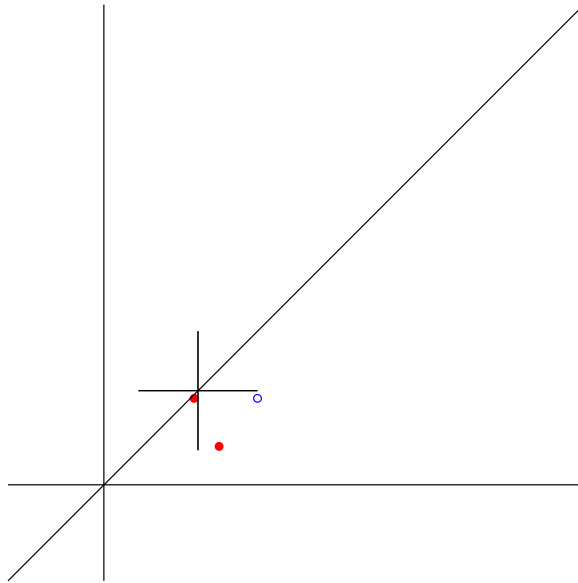


Figure I.2: Illustration of the growing incomparable points number (2)

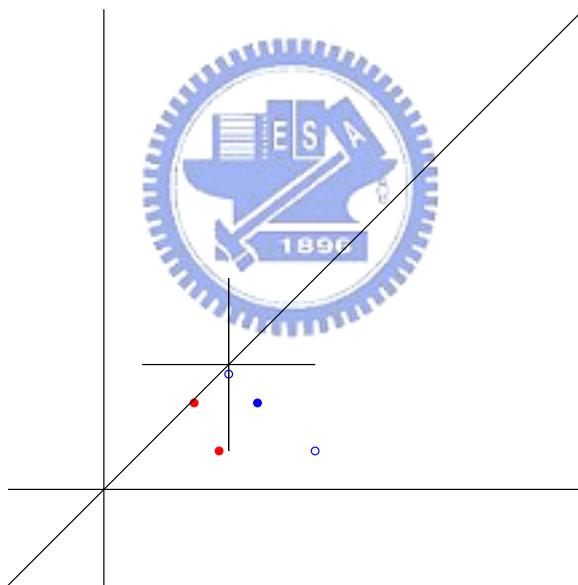


Figure I.3: Illustration of the growing incomparable points number (3)

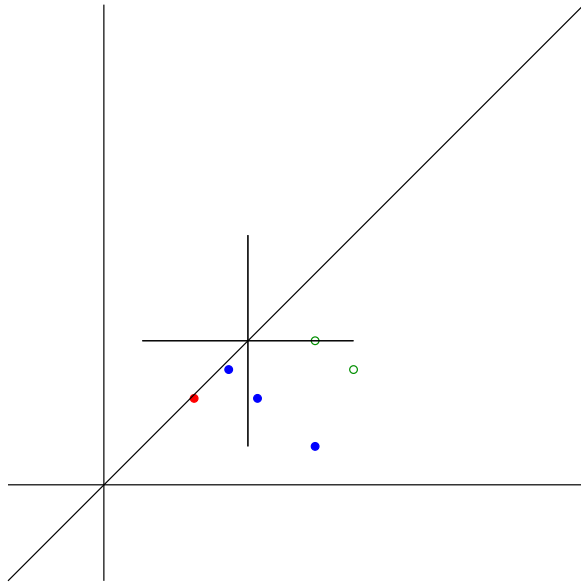


Figure I.4: Illustration of the growing incomparable points number (4)

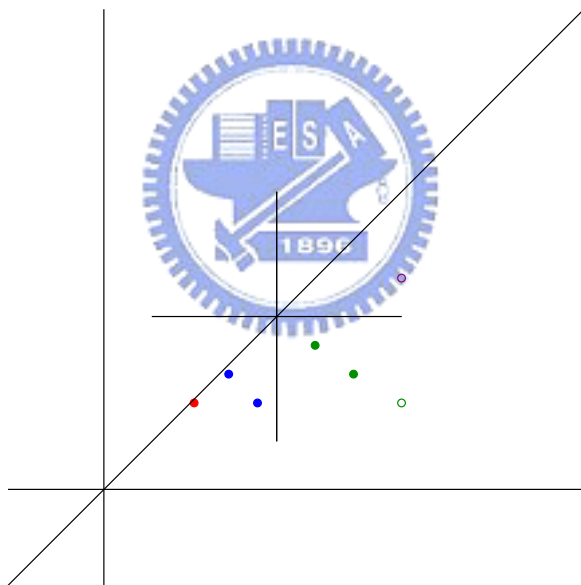


Figure I.5: Illustration of the growing incomparable points number (5)

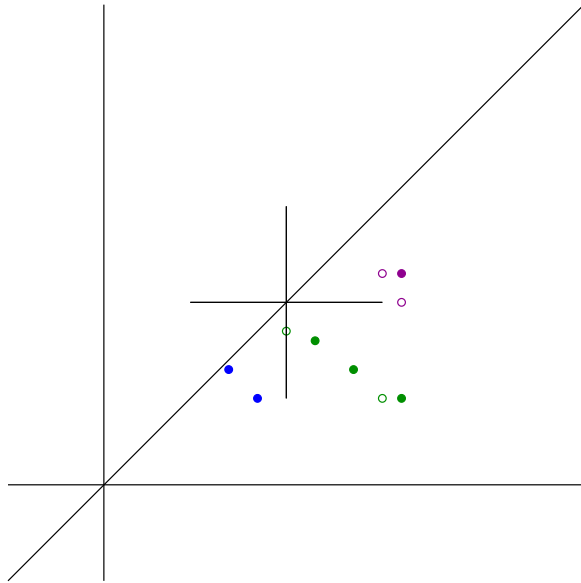


Figure I.6: Illustration of the growing incomparable points number (6)

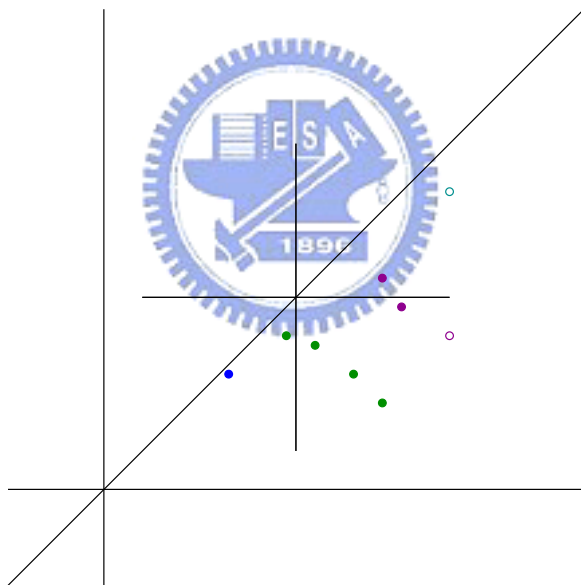


Figure I.7: Illustration of the growing incomparable points number (7)

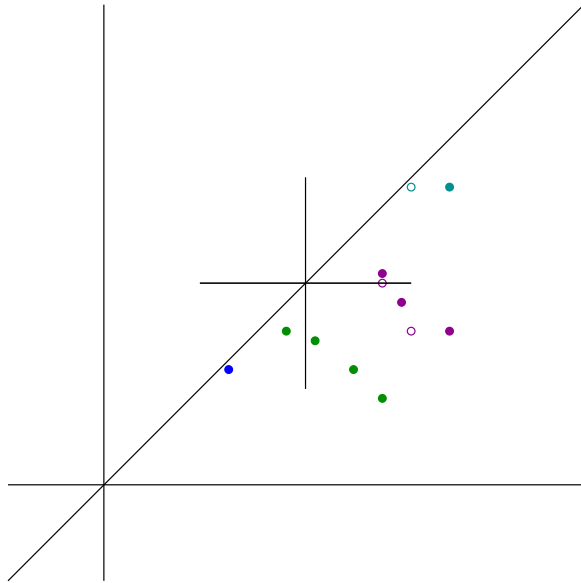


Figure I.8: Illustration of the growing incomparable points number (8)

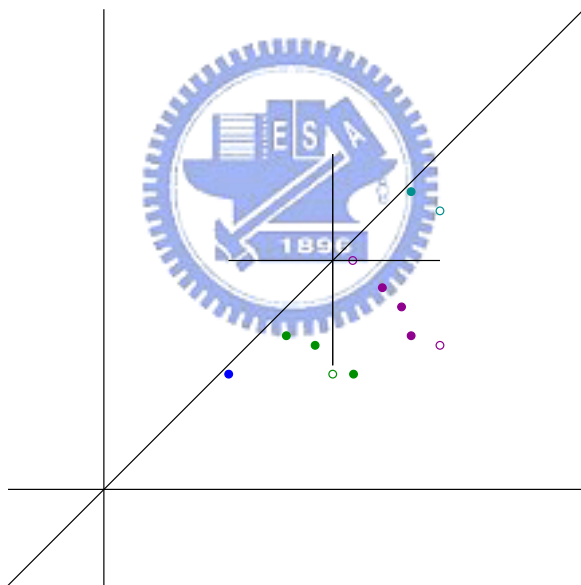


Figure I.9: Illustration of the growing incomparable points number (9)

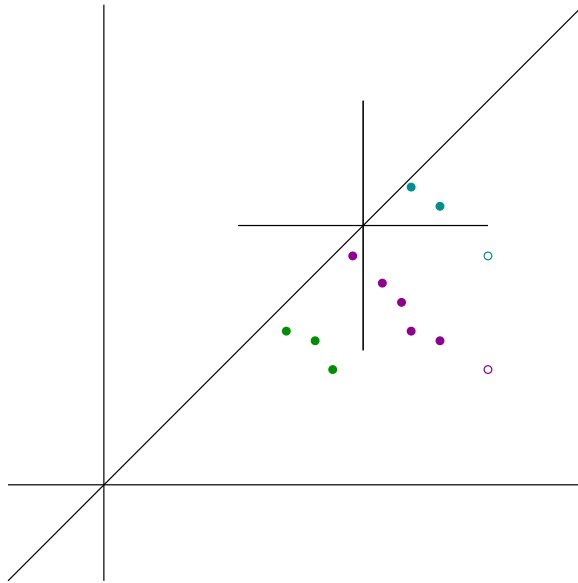


Figure I.10: Illustration of the growing incomparable points number (10)

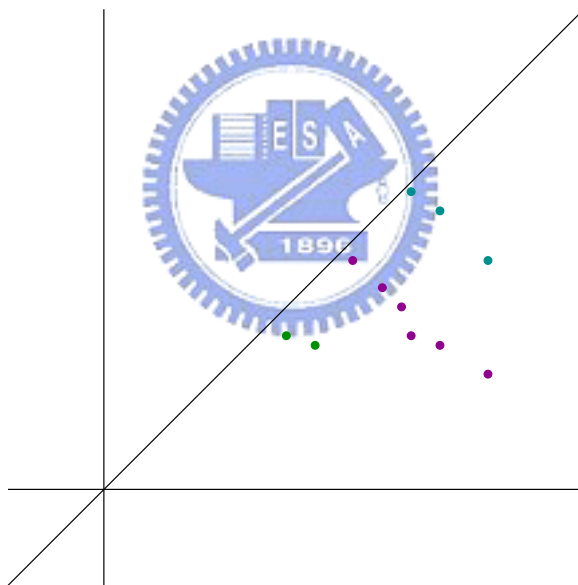


Figure I.11: Illustration of the growing incomparable points number (11)

Appendix II : Experimental Codes

The following is the experimental codes.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
struct data{
    int card;
    int anchor;
    int x;
    int y;
    struct data prev;
    struct data next;
};
```



```
struct fourcard{
    int total;
    int level;
    struct data head;
    struct data tail;
};
```

```
struct data node_list_head, node_list_tail;
struct fourcard level_ptr[4];
```

```
int anchor_array[30000];
```

```
void init_level(void){
    int i;
```

```

for (i=0;i<4;i++){
    level_ptr[i] = malloc(sizeof(struct fourcard));
    level_ptr[i]->total=0;
    level_ptr[i]->level=i+1;
    level_ptr[i]->head=NULL;
    level_ptr[i]->tail=NULL;
}
}

```

```

void special_init(int x, int y){
    int i;
    struct data node = malloc(sizeof(struct data));

```

```

for (i=0;i<4;i++){
    level_ptr[i] = malloc(sizeof(struct fourcard));
    level_ptr[i]->total=0;
    level_ptr[i]->level=i+2;
    level_ptr[i]->head=NULL;
    level_ptr[i]->tail=NULL;
}

```



```

node->card = 2;
node->anchor = 0;
node->x = x;
node->y = y;
node->prev = NULL;
node->next = NULL;
level_ptr[0]->total++;
level_ptr[0]->head = node;
level_ptr[0]->tail = node;
}

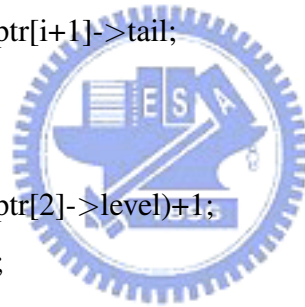
```

```

void incre_level(void){
    int i;
    struct data tmp1, tmp2;

    tmp1 = level_ptr[0]->head;
    while (tmp1){
        tmp2 = tmp1;
        tmp1 = tmp1->next;
    }
    for (i=0;i<3;i++){
        level_ptr[i]->total = level_ptr[i+1]->total;
        level_ptr[i]->level = level_ptr[i+1]->level;
        level_ptr[i]->head = level_ptr[i+1]->head;
        level_ptr[i]->tail = level_ptr[i+1]->tail;
    }
    level_ptr[3]->total = 0;
    level_ptr[3]->level = (level_ptr[2]->level)+1;
    level_ptr[3]->head = NULL;
    level_ptr[3]->tail = NULL;
}

```



```

void incre2_level(void){
    int i;
    for (i=0;i<2;i++){
        level_ptr[i]->total = level_ptr[i+2]->total;
        level_ptr[i]->level = level_ptr[i+2]->level;
        level_ptr[i]->head = level_ptr[i+2]->head;
        level_ptr[i]->tail = level_ptr[i+2]->tail;
    }
    level_ptr[2]->total = 0;
    level_ptr[2]->level = level_ptr[0]->level+2;
    level_ptr[2]->head = NULL;
}

```

```

level_ptr[2]->tail = NULL;

level_ptr[3]->total = 0;
level_ptr[3]->level = level_ptr[1]->level+2;
level_ptr[3]->head = NULL;
level_ptr[3]->tail = NULL;
}

```

```

int find_level(struct data node){
    int i;
    for (i=0;i<4;i++){
        if (level_ptr[i]->level == node->card)
            return i;
    }
    return -1;
}

```



```

int region(struct data node, int anchor){
    if (anchor < node->y)
        return 2;
    else if (anchor >= node->x)
        return 0;
    else
        return 1;
}

```

```

int opera(struct data node, int anchor, int label){
    if (anchor < node->y)
        return 0;
    else if (anchor >= node->x){
        if ((anchor-label < node->x) && (anchor-label >= node->y))
            return 99;
    }
}

```



```

else if (anchor-label >= node->x)
    return 1;
else
    return 11;
}
else{
    if (anchor-label < node->y)
        return 22;
    else
        return 2;
}
}

```

```

struct data operax(struct data node, int anchor, int label){
    struct data tmp = malloc(sizeof(struct data));

    tmp->anchor = anchor;
    tmp->card = node->card+1;
    tmp->prev = NULL;
    tmp->next = NULL;
    tmp->y = node->y;
    if ((anchor-label) >= node->x)
        tmp->x = anchor;
    else
        tmp->x = anchor+label;
    return tmp;
}

```

```

struct data operay(struct data node, int anchor, int label){
    int temp;
    struct data tmp = malloc(sizeof(struct data));

```



```

tmp->anchor = anchor;
tmp->card = node->card+1;
tmp->prev = NULL;
tmp->next = NULL;
tmp->x = node->x;
if ((anchor-label) >= node->y)
    tmp->y = anchor;
else
    tmp->y = anchor+label;
if (tmp->x < tmp->y){
    temp = tmp->x;
    tmp->x = tmp->y;
    tmp->y = temp;
}
return tmp;
}

```



```

void del_node(struct data node){
    int i = find_level(node);
    int j;

    if (i == -1){
        printf("find_level return -1 !!!\n");
        exit(1);
    }
    if (level_ptr[i]->total == 1){
        level_ptr[i]->head = NULL;
        level_ptr[i]->tail = NULL;
    }
    else if (node == level_ptr[i]->head){
        level_ptr[i]->head = level_ptr[i]->head->next;
        level_ptr[i]->head->prev = NULL;
    }
}

```

```

}
else if (node == level_ptr[i]->tail){
    level_ptr[i]->tail = level_ptr[i]->tail->prev;
    level_ptr[i]->tail->next = NULL;
}
else{
    node->prev->next = node->next;
    node->next->prev = node->prev;
}
level_ptr[i]->total--;
}

```

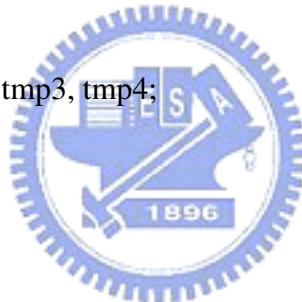
```

int tune_node(int anchor){
    int reg;
    struct data tmp, tmp1, tmp2, tmp3, tmp4;

    tmp3 = level_ptr[2]->head;
    tmp4 = level_ptr[3]->head;
    if (tmp4 && (region(tmp4, anchor) == 0)){
        incre2_level();
        return 1;
    }
    if (tmp3 && (region(tmp3, anchor) == 0)){
        incre_level();
        return 1;
    }

    tmp1 = level_ptr[0]->head;
    tmp2 = level_ptr[1]->head;
    while(tmp2){
        reg = region(tmp2, anchor);
        while (tmp1){

```



```

tmp = tmp1;
tmp1 = tmp1->next;
if (reg == 0){
    if (tmp->x >= tmp2->y)
        del_node(tmp);
}
else if (reg == 1){
    if (tmp->x >= tmp2->x)
        del_node(tmp);
}
}
tmp2 = tmp2->next;
}

```

```

tmp1 = level_ptr[0]->head;
tmp2 = level_ptr[1]->head;
tmp3 = level_ptr[2]->head;
while (tmp3){

```



```

    reg = region(tmp3, anchor);
    while (tmp2){
        tmp = tmp2;
        tmp2 = tmp2->next;
        if (reg == 0){
            printf("increase level twice (in tmp2)!!!\n");
            exit(1);
        }
        else if (reg == 1){
            if (tmp->x >= tmp3->x)
                del_node(tmp);
        }
    }
}
while (tmp1){

```

```

tmp = tmp1;
tmp1 = tmp1->next;
if (reg == 0){
    printf("increase level twice (in tmp1)!!!\n");
    exit(1);
}
else if (reg == 1){
    if (tmp->x >= tmp3->x)
        del_node(tmp);
}
}
tmp3 = tmp3->next;
}

```

```

tmp1 = level_ptr[0]->head;
tmp2 = level_ptr[1]->head;
tmp3 = level_ptr[2]->head;
tmp4 = level_ptr[3]->head;
while (tmp4){

```



```

    reg = region(tmp4, anchor);
    while (tmp3){
        tmp = tmp3;
        tmp3 = tmp3->next;
        if (reg == 0){
            printf("increase 2 levels twice (in tmp3)!!!\n");
            exit(1);
        }
        else if (reg == 1){
            if (tmp->x >= tmp4->x)
                del_node(tmp);
        }
    }
}

```

```

while (tmp2){
    tmp = tmp2;
    tmp2 = tmp2->next;
    if (reg == 0){
        printf("increase 2 levels twice (in tmp2)!!!\n");
        exit(1);
    }
    else if (reg == 1){
        if (tmp->x >= tmp4->x)
            del_node(tmp);
    }
}
while (tmp1){
    tmp = tmp1;
    tmp1 = tmp1->next;
    if (reg == 0){
        printf("increase 2 levels twice (in tmp1)!!!\n");
        exit(1);
    }
    else if (reg == 1){
        if (tmp->x >= tmp4->x)
            del_node(tmp);
    }
}
tmp4 = tmp4->next;
}
return 0;
}

```



```

void replace_node(struct data a, struct data b){
    b->prev = a->prev;
    b->next = a->next;
}

```

```

if (a->prev)
    a->prev->next = b;
if (a->next)
    a->next->prev = b;
}

```

```

void add_node(struct data node){

```

```

    int i;

```

```

    struct data tmp;

```

```

    node->prev = NULL;

```

```

    node->next = NULL;

```

```

    if (node->card > level_ptr[3]->level)

```

```

        incre_level();

```

```

    i = find_level(node);

```



```

    if (level_ptr[i]->total == 0){

```

```

        level_ptr[i]->head = node;

```

```

        level_ptr[i]->tail = node;

```

```

        level_ptr[i]->total++;

```

```

    }

```

```

    else if (node->x < level_ptr[i]->head->x){

```

```

        node->next = level_ptr[i]->head;

```

```

        level_ptr[i]->head->prev = node;

```

```

        level_ptr[i]->head = node;

```

```

        level_ptr[i]->total++;

```

```

        tmp = level_ptr[i]->head;

```

```

        while ((tmp->next) && (tmp->y <= tmp->next->y))

```

```

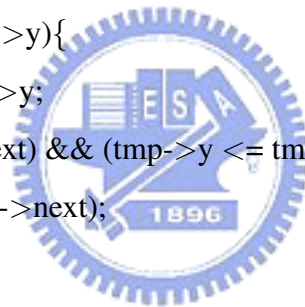
            del_node(tmp->next);

```

```

}
else if (node->x > level_ptr[i]->tail->x){
    if (node->y < level_ptr[i]->tail->y){
        node->prev = level_ptr[i]->tail;
        level_ptr[i]->tail->next = node;
        level_ptr[i]->tail = node;
        level_ptr[i]->total++;
    }
}
else{
    tmp = level_ptr[i]->head;
    while(tmp){
        if (tmp->x == node->x){
            if (tmp->y > node->y){
                tmp->y = node->y;
                while ((tmp->next) && (tmp->y <= tmp->next->y))
                    del_node(tmp->next);
            }
            return;
        }
        else if (node->x < tmp->next->x){
            if (tmp->y > node->y){
                node->prev = tmp;
                node->next = tmp->next;
                tmp->next->prev = node;
                tmp->next = node;
                level_ptr[i]->total++;
                tmp = node;
                while ((tmp->next) && (tmp->y <= tmp->next->y))
                    del_node(tmp->next);
            }
            return;
        }
    }
}

```




```

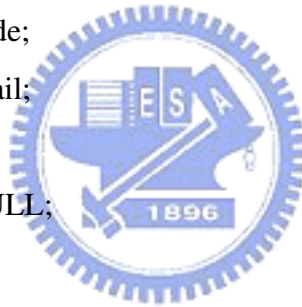
    }
    tmp = tmp->next;
}
}
}

```

```

void add_to_node_list(struct data node){
    if (node_list_tail == NULL){
        node_list_head = node;
        node_list_tail = node;
        node_list_tail->next = NULL;
    }
    else {
        node_list_tail->next = node;
        node->prev = node_list_tail;
        node_list_tail = node;
        node_list_tail->next = NULL;
    }
}

```



```

void gen_node_list(int anchor, int label){
    int i,oper;
    struct data tmp;

    node_list_head = NULL;
    node_list_tail = NULL;

    for (i=0;i<4;i++){
        tmp = level_ptr[i]->head;
        while (tmp){
            oper = opera(tmp, anchor, label);
            if (oper == 99){

```

```

        add_to_node_list(operax(tmp, anchor,label));
        add_to_node_list(operay(tmp, anchor,label));
        del_node(tmp);
    }
    else if (oper == 1){
        add_to_node_list(operax(tmp, anchor,label));
        del_node(tmp);
    }
    else if (oper == 11){
        add_to_node_list(operax(tmp, anchor,label));
    }
    else if (oper == 2){
        add_to_node_list(operay(tmp, anchor,label));
        del_node(tmp);
    }
    else if (oper == 22){
        add_to_node_list(operay(tmp, anchor,label));
    }
    tmp = tmp->next;
}
}
}

```



```

void process_anchor_array(int offset, int anchor, int label, int rate, int sid){
    int i;
    for (i=0;i<5000;i++)
        anchor_array[i] = 0;
    for (i=0;i<(anchorrates);i++){
        srand(time(NULL)+i+sid);
        anchor_array[(rand()%anchor)+offset] = (rand()%label);
    }
    for (i=0;i<anchor;i++){

```

```

    if (anchor_array[i])
        printf("%2d : %d\n",i,anchor_array[i]);
    }
}

```

```

void node_list_to_level(void){

```

```

    int i=0;

```

```

    struct data tmp;

```

```

    while (node_list_head){

```

```

        tmp = node_list_head;

```

```

        node_list_head = node_list_head->next;

```

```

        add_node(tmp);

```

```

    }

```

```

    node_list_head = NULL;

```

```

    node_list_tail = NULL;

```

```

}

```



```

void print_table(void){

```

```

    int i,j;

```

```

    struct data tmp;

```

```

    for (i=0;i<4;i++){

```

```

        if (level_ptr[i]->total == 5)

```

```

            printf("!!!!!!!\n");

```

```

        else if (level_ptr[i]->total == 6)

```

```

            printf("@@@@@@@@\n");

```

```

        else if (level_ptr[i]->total == 7)

```

```

            printf("77777777\n");

```

```

        else if (level_ptr[i]->total == 8)

```

```

            printf("88888888\n");

```

```

        else if (level_ptr[i]->total == 9)

```

```

            printf("99999999\n");

```

```

else if (level_ptr[i]->total > 9)
    printf("orzorzorz\n");
printf("%d(%d) : ",level_ptr[i]->level,level_ptr[i]->total);
if (level_ptr[i]->total){
    j = 0;
    tmp = level_ptr[i]->head;
    while(tmp){
        printf(" (%2d,%2d)", tmp->x, tmp->y);
        tmp = tmp->next;
        j++;
    }
    if (j != level_ptr[i]->total)
        printf("\nXXXXXXXXXXXXXXXXX");
}
printf("\n");
}
}

```



```

void array_to_node_list(int anchor_number){
    int i,j;
    struct data tmp;
    for (i=0;i<anchor_number;i++){
        if (anchor_array[i]){
            if (tune_node(i))
                tune_node(i);
            printf("\n=== %d(%d)\n", i, anchor_array[i]);
            print_table();
            gen_node_list(i,anchor_array[i]);
            tmp = node_list_head;

            printf("—————\n");
            while(tmp){

```

```

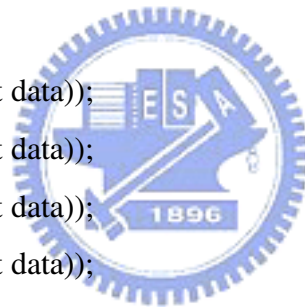
        printf("%d(%d,%d) ",tmp->card,tmp->x,tmp->y);
        tmp = tmp->next;
    }
    printf("\n");
    printf("—————\n");
    node_list_to_level();
    print_table();
}
}
}

```

```

void input_spec(void){
    struct data node1, node2, node3, node4, node5;
    int i;
    node1 = malloc(sizeof(struct data));
    node2 = malloc(sizeof(struct data));
    node3 = malloc(sizeof(struct data));
    node4 = malloc(sizeof(struct data));
    node5 = malloc(sizeof(struct data));
    for (i=0;i<4;i++){
        level_ptr[i] = malloc(sizeof(struct fourcard));
        level_ptr[i]->total = 0;
        level_ptr[i]->level = i+88;
        level_ptr[i]->head = NULL;
        level_ptr[i]->tail = NULL;
    }
    node1->prev = NULL;
    node1->next = NULL;
    node1->card = 89;
    node1->anchor = 12;
    node1->x = 13;
    node1->y = 12;

```



```
add_node(node1);
node2->prev = NULL;
node2->next = NULL;
node2->card = 89;
node2->anchor = 12;
node2->x = 16;
node2->y = 9;
add_node(node2);
node3->prev = NULL;
node3->next = NULL;
node3->card = 89;
node3->anchor = 12;
node3->x = 24;
node3->y = 2;
add_node(node3);
node4->prev = NULL;
node4->next = NULL;
node4->card = 90;
node4->anchor = 12;
node4->x = 24;
node4->y = 16;
add_node(node4);
node5->prev = NULL;
node5->next = NULL;
node5->card = 90;
node5->anchor = 12;
node5->x = 27;
node5->y = 12;
add_node(node5);
}
```

int



```

main(int argn, char argv[]){
    int i,j;
    if (argn!=9){
        printf("bad arguments!!\n");
        return;
    }

    j = atoi(argv[7]);
    for (i=atoi(argv[6]);i>0;i--){
        if (atoi(argv[1]) < atoi(argv[2])){
            process_anchor_array(atoi(argv[1]),atoi(argv[3]),atoi(argv[4])/j,atoi(argv[5]),i);
            if (atoi(argv[8])==0)
                special_init(atoi(argv[2]), atoi(argv[1]));
            else
                input_spec();
        }
        else{
            process_anchor_array(atoi(argv[2]),atoi(argv[3]),atoi(argv[4])/j,atoi(argv[5]),i);
            if (atoi(argv[8])==0)
                special_init(atoi(argv[1]), atoi(argv[2]));
            else
                input_spec();
        }
        j--;
        if (j==0)
            j = atoi(argv[7]);
        array_to_node_list(atoi(argv[3]));
        printf("\n\n\n\n");
    }
    return;
}

```

