Space-efficient Algorithms for Geometric Optimization Problems

by

Minati De

for the award of the degree of

Doctor of Philosophy

in

Computer Science

at the

Indian Statistical Institute, Kolkata

Advisor:

Professor Subhas C. Nandy
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata, INDIA 700 108
Dedicated to

The Almighty
Undertaking this PhD has been an unforgettable experience for me and it would not have been possible to do without the support and guidance that I received from many people. I take this opportunity to acknowledge and extend my sincere gratitude to all of them.

First of all, I am deeply grateful to my advisor Prof. Subhas C. Nandy. To work with him has been a real pleasure to me, with heaps of fun and excitement. He has been a steady influence throughout my Ph.D. career; he has oriented and supported me with promptness and care, and has always been patient and encouraging in times of new ideas and difficulties. His ability to select and to approach compelling research problems, and his hard work set an example. Without his encouragement and guidance this thesis would not have materialized. Above all, he made me feel a friend, which I appreciate from my heart.

In the early days of my PhD, I attended a course on Research Methodology offered by Prof. Palash Sarkar which really helps me a lot. I am greatly indebted to him for the inspiring lectures.

I have been very privileged to get to know and to collaborate with many other great people. Over the last few years, Dr. Sasanka Roy has been a faithful friend and co-author. I thank him for teaching me so much about life, research, how to tackle new problems etc.. I enjoyed a lot to work with Dr. Gautam K. Das. My heart-felt thanks to him. Prof. Anil Maheshwari’s balance research interests and personal pursuits while I was visiting him at Carleton University was a valuable source of motivation. I have greatly enjoyed the opportunity to work with Prof. Michiel Smid, whose scientific work has inspired me a lot. I thank Dr. P. Carmi for being my co-author.

I owe a lot of gratitude to Indu-kakima. She has always been there for me with her motherly hand whenever I needed it the most and I feel privileged to be associated with a person like her during my life.

I would like to acknowledge all the teachers who help me to learn since my childhood, I would not have been here without their guidance, blessing and support.
I must mention the encouragement, advice and support I received from Dr. Ansuman Banerjee, Prof. Bhargav B. Bhattacharya, Dr. Arijit Bishnu, Prof. Nabanita Das, Prof. Sandip Das, Prof. Jayasree Dattagupta, Dr. Sasthi C. Ghosh, Prof. Krishnendu Mukhopadhaya, Prof. Sandeep Sen, Prof. Bhabani P. Sinha, Prof. Susmita Sur-Kolay. My heart-felt thanks to all of them. I would like to thank all other faculty members of Indian Statistical Institute for their valuable advice, encouragement and support.

Continuing a fine tradition, I want to thank all the lab-mates: Subhasis-da, Debasish-da, Arindam-da, Aritra-da, Anshuman-da, Ayan-da, Sruti-di, Shyamsoor-di, Soumen-da, Subhash-da, Dibakar, Sandip, Soumyottam, and Sukanta, for all the fun we have had in the last few years. I cherish my stay in the institute largely owing to my friends. I thank all of them, who made it enjoyable to be at ISI. I refrain from putting down names, since it is so easy to miss some name or the other. If you think your name should be here, then you know you are one of them.

I would like to thank Indian Statistical Institute, Kolkata for providing me a good research environment, the opportunity to get acquainted with distinguished researchers and financial support during my tenure as a research scholar. My thanks go to all the members of the faculty and office staff of this institute and especially to the members of Advanced Computing and Microelectronics Unit for making this unit and the institute a cordial and wonderful place to work.

My most important acknowledgement goes to my family and friends who have filled my life with happiness; most significantly, to my parents for having always given me the freedom to choose what I wanted to pursue; to my brother, Debasis who have filled my heart with joy.

(Minati De)
Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
203 B T Road, Kolkata
Pin 700 108, India
Abstract

The efficiency of an algorithm is mainly measured using two types of resources: time and space. As the trend of using and handling large pool of data is going to impact new age of computing, the space-efficiency of an algorithm gets spacious attention. In this thesis, our main focus is on space-efficient aspects of some fundamental geometric optimization problems while not deteriorating the time complexity too much. We describe an in-place maintenance of priority search tree, a widely used data structure for storing points in $\mathbb{R}^2$. Maximum matching in a bipartite graph is an important problem which has large theoretical as well as practical implications. We provide an in-place algorithm for this problem which is the first attempt in the literature. Prune-and-search is an important paradigm which is largely used to solve different optimization problems. We describe general schemes to implement prune-and-search algorithms in both in-place and read-only model. We explain these by computing minimum enclosing circle, low dimensional linear programming which accepts prune-and-search solutions. We show how to find the maximum clique of different geometric intersection graphs in an in-place manner using constant amount of extra-space. While most of the approximation algorithms are very difficult to implement, we provide in-place way to implement some constant factor approximation algorithms for finding the minimum discrete piercing set for unit disks.
Contents

1 Introduction 1

1.1 The state of the art ............................. 2
1.2 Scope of the thesis ............................... 4
1.3 Organization of the thesis ....................... 6

2 Review and Related Works 7

2.1 Fundamental problems ........................... 8
2.2 Geometric problems .............................. 10
   2.2.1 Range searching ............................. 10
   2.2.2 Convex hull ................................. 12
   2.2.3 Nearest neighbor problems ................. 13
   2.2.4 Klee's measure problem .................... 14
   2.2.5 Line segment intersection .................. 14
   2.2.6 Slope selection ............................. 14
   2.2.7 Visibility and shortest path problems in simple polygon 15
   2.2.8 Minimum enclosing circle ................... 16
   2.2.9 Linear programming ......................... 17
2.3 Graph theoretic problems ...................... 18
   2.3.1 Maximum clique and independent set of geometric intersection graphs .................. 20
   2.3.2 Discrete piercing set for unit disk graph .......... 22

3 An In-Place Min-Max Priority Search Tree 27

3.1 Introduction .................................. 27
3.2 Constructing the in-place MINMAXPST ............ 31
   3.2.1 Improving the construction time .......... 35
3.3 Queries on the in-place MINMAXPST ............. 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>\textsc{LeftMostNE}(x_0,y_0)</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2</td>
<td>\textsc{Highest3SidedUp}(x_0,x_1,y_0)</td>
<td>42</td>
</tr>
<tr>
<td>3.3.3</td>
<td>\textsc{Enumerate3SidedUp}(x_0,x_1,y_0)</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Enumerating all axis-parallel maximal empty rectangles</td>
<td>54</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Correctness and complexity</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusion</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>Space-efficient Prune-and-Search</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>General scheme for prune-and-search</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>For in-place model</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>For read-only model</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Selection in the pairing scheme</td>
<td>68</td>
</tr>
<tr>
<td>4.5.1</td>
<td>ALGO-MR</td>
<td>69</td>
</tr>
<tr>
<td>4.5.2</td>
<td>ALGO-RR</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusion</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>Minimum Enclosing Circle with Limited Work-space</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of Megiddo's algorithm</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>In-place implementation of MEC</td>
<td>77</td>
</tr>
<tr>
<td>5.3.1</td>
<td>In-place implementation of constrained-MEC</td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>When the memory is read-only</td>
<td>83</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Constrained-MEC in read-only setup</td>
<td>84</td>
</tr>
<tr>
<td>5.4.2</td>
<td>(Unconstrained) MEC in the read-only setup</td>
<td>88</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion</td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td>Convex Hull in Read-only Memory</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>91</td>
</tr>
</tbody>
</table>
### Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Overview of Kirkpatrick and Seidel’s algorithm</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Reporting the hull-edges in-order</td>
<td>94</td>
</tr>
<tr>
<td>6.4</td>
<td><strong>COMPUTE-BRIDGE</strong> with limited work-space</td>
<td>94</td>
</tr>
<tr>
<td>6.5</td>
<td>Correctness and complexity</td>
<td>98</td>
</tr>
<tr>
<td>6.6</td>
<td>Conclusion</td>
<td>98</td>
</tr>
<tr>
<td>7</td>
<td>Low Dimensional Linear Programming</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>101</td>
</tr>
<tr>
<td>7.2</td>
<td>Linear programming in $\mathbb{R}^2$</td>
<td>102</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Read-only linear programming algorithm in $\mathbb{R}^2$</td>
<td>102</td>
</tr>
<tr>
<td>7.3</td>
<td>Linear programming in $\mathbb{R}^3$</td>
<td>104</td>
</tr>
<tr>
<td>7.3.1</td>
<td><strong>TESTING-LINE</strong></td>
<td>106</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Read-only linear programming in $\mathbb{R}^3$</td>
<td>108</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusion</td>
<td>108</td>
</tr>
<tr>
<td>8</td>
<td>Maximum Clique of Geometric Intersection Graphs</td>
<td>111</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>111</td>
</tr>
<tr>
<td>8.2</td>
<td>Maximum clique of an interval graph</td>
<td>113</td>
</tr>
<tr>
<td>8.3</td>
<td>Maximum clique for rectangles of arbitrary size</td>
<td>115</td>
</tr>
<tr>
<td>8.4</td>
<td>Maximum clique for fixed height rectangles</td>
<td>116</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Processing of a strip $S$</td>
<td>117</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Processing of Case 3</td>
<td>119</td>
</tr>
<tr>
<td>8.5</td>
<td>Geometric clique for disks of arbitrary radii</td>
<td>120</td>
</tr>
<tr>
<td>8.6</td>
<td>Graphical clique for unit disk graph</td>
<td>122</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Maximum matching in the bipartite graph $G_{B_i}$</td>
<td>124</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Complexity analysis</td>
<td>134</td>
</tr>
<tr>
<td>8.7</td>
<td>Maximum independent set</td>
<td>135</td>
</tr>
<tr>
<td>8.8</td>
<td>Conclusion</td>
<td>136</td>
</tr>
<tr>
<td>9</td>
<td>Minimum Discrete Piercing Set for Unit Disks</td>
<td>137</td>
</tr>
</tbody>
</table>
9.1 Introduction ........................................... 137
9.2 Approximation algorithms ............................ 138
  9.2.1 Simple $t$-approximation algorithms ............ 139
  9.2.2 Further improving the approximation factor ....... 144
9.3 Polynomial time approximation scheme ............... 153
  9.3.1 Improvement in time complexity .................. 156
9.4 Conclusion ............................................. 157

10 Concluding Remarks ..................................... 159
  10.1 Space-efficient tools .............................. 159
    10.1.1 In-place priority search tree .................. 159
    10.1.2 Space-efficient prune-and-search .............. 160
    10.1.3 Maximum matching for bipartite graph ......... 161
  10.2 Space-efficient algorithms for geometric intersection graphs .... 161
    10.2.1 Maximum clique of geometric intersection graphs ... 162
    10.2.2 Minimum discrete piercing set for unit disks .... 162
  10.3 Conclusion ........................................... 163
## List of Figures

3.1 Example of a MINMAXPST ........................................... 30
3.2 Demonstration of the invariants of MINMAXPST ................. 32
3.3 Construction of \(i\)-th level of MINMAXPST ....................... 33
3.4 Improving the construction time of MINMAXPST - the \(i\)-th iteration ................................................................. 36
3.5 Different cases for the procedure LEFTMOSTNE\((x_0, y_0)\) and the corresponding assignments of \(p\) and \(q\) ................ 41
3.6 Different cases for the procedure CHECKLEFT\((p)\) ............... 47
3.7 Different cases for the procedure CHECKLEFTIN\((p')\) .......... 48
3.8 Enumerating Type-\(C\) axis-parallel MER keeping the point \(p^*\) at the top boundary. ...................................................... 58

5.1 Block partition of the array \(P\) ........................................ 80

8.1 (a) Graphical clique and (b) geometric clique of disk graph ... 112
8.2 (a) Arrangement of \(R_t \cup R_b\) in the array \(R\), and (b) processing of Case 3 .......................................................... 118
8.3 (a) Closed and (b) non-closed arc, and (c) the circular arcs along \(\Delta(C_i)\) after considering all the disks intersecting \(\Delta(C_i)\) . 121
8.4 Demonstration of \(R_{ij}\) .................................................... 123
8.5 Different steps of finding an augmenting path ...................... 127
8.6 Significance of \(\beta\) ......................................................... 128
8.7 Demonstration for the proof of Lemma 8.3 ......................... 129
8.8 Handling useless matched vertex ................................. 131
8.9 Demonstration for the proof of Lemma 8.4 ......................... 131
8.10 (a) Augmenting path, and (b) updating the matching ......... 133

9.1 Discrete piercing set for unit disks .............................. 140
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Using hexagonal partitioning</td>
<td>143</td>
</tr>
<tr>
<td>9.3</td>
<td>Implementing hexagonal partitioning</td>
<td>144</td>
</tr>
<tr>
<td>9.4</td>
<td>Proof of Lemma 9.1</td>
<td>146</td>
</tr>
<tr>
<td>9.5</td>
<td>Coloring the cells for 3-factor approximation algorithm</td>
<td>150</td>
</tr>
<tr>
<td>9.6</td>
<td>Demonstration of packing 15 unit disks in a rectangle of size 10 × 6 without any overlapping pair</td>
<td>153</td>
</tr>
</tbody>
</table>
We are living in the world of computation which needs to tackle enormous amount of data. Starting from astronomy to statistics, everywhere massive data is now a phenomenon. Because of rapid increase of human interaction with internet, different business models target to give personalized online services which needs to handle huge data. On the other hand, memory is also becoming cheap day by day. External memory like flash devices are extensively used. This gives programmers the freedom of using huge amount of virtual memory. But this is not always a good solution for handling huge amount of data as it may lead to a huge number of disk access which is too much time and energy consuming task.

On the other hand, increasing interest of using tiny portable devices with multipurpose activity shift the scenario. Now, the manufacturers relish to make devices as tiny as possible. As a result of that those devices have small amount memory.

Such environment leads researchers to think how to compute with minimal extra-space apart from the space required for storing the input, as well as how to represent data succinctly. Thus, designing space-efficient algorithms get spacious attention.

In this thesis, our focus is mainly on space-efficient aspects of some fundamental geometric optimization problems.
1.1 The state of the art

To cope up with these settings, the researchers are using the following paradigms: in-place model, read-only model and succinct data structure. The details of the paradigms are described below. Unless otherwise mentioned, throughout the thesis we use the term space/extra-space to mean a word of $O(\log n)$ bits.

**In-place model:** An *in-place* algorithm satisfies all of the following constraints:

(i) the input objects are given as array elements,

(ii) during the execution of the algorithm, swapping of the elements in the input array is permissible,

(iii) after the execution, all the input elements should be present in the array (may be in some different permutation),

(iv) only $O(1)$ extra-space usage during execution.

Among all these constraints, if constraint (iii) is violated only then the algorithm is called *destructive in-place* algorithm because the input is destroyed in this type of algorithm. If all the first three constraints are maintained but it uses $O(k)$ extra-space where $k \geq n$ and $n$ is the size of the input, then that algorithm does not make sense. If $k < n$, then we refer that algorithm as an *in-place algorithm with $O(k)$ extra-space*. If $k = \log n$, then that algorithm is referred to as *in-situ*. When we refer to an algorithm as *in-place* without specifying the amount of extra-space, then it means that it is an in-place algorithm with $O(1)$ extra-space.

An in-place data structure (sometimes referred to as *implicit* data structure) for a set of $n$ input objects given in an array $D$, is one which needs no additional space other than the array $D$ and $O(1)$ extra-space may be used during the operations to be performed on it. So, the resultant implicit data structure is basically some permutation of the input objects in the same input array. During the construction of the implicit data structure, only $O(1)$ extra-space is used. A basic example of in-place data structure for one dimensional points is an array in sorted order where searching can be done in $O(\log n)$ time.
This paradigm has several merits compared to the traditional algorithms. As the in-place algorithm uses small additional amount of memory, it shows better locality of reference. During execution, it can store large amount of data in faster memory. As a result of which the number of slower memory access is almost nil. Thus it practically performs faster than the traditional algorithms. In mission critical application, the traditional algorithm may fail because of out of memory during execution; whereas the in-place algorithm is less prone to failure [Brönnimann 2002].

**Read-only model:** In this paradigm, the input to the algorithm is considered to be in read-only array and during the computation only sub-linear amount of extra-space is allowed. This is a more restricted version of in-place model. Here, swapping of the elements in the input array is not allowed. Such a situation arises in the concurrent programming environment where many processes access the same data, and hence modifying the data by a process during the execution is not permissible. For security or technical reason also, write access to data may be restricted. There are many more reasons to consider the input memory to be read-only.

It is really challenging to design algorithm to work in read-only model without degrading the computation time too much.

A problem that is solvable with input data in a read-only memory and with $O(1)$ extra-space is widely known to be in log-space ($L$) in complexity theory literature [Arora 2009], i.e, the problem can be solved by a deterministic Turing machine using logarithmic amount of boolean flags or constant number of pointers.

A restricted variation of the read-only model is the multi-pass model where random access of the elements of the input array is not permitted. Here in each pass, the input array is read from the beginning to its end. Thus, objective is to minimize the number of passes.

**Succinct data-structures:** In this area of research, the objective is to construct a data structure which can be used to solve a given problem using space close to information theoretic lower bound of that problem. Guy Ja-
cobson [Jacobson 1989] introduced this idea in the year 1989. Since then this field is very active. The important studies include succinct representation of (i) dynamic binary tree [Munro 2001], (ii) triangulation of a point set inside a bounded region [Aleardi 2005], to name a few. For a detailed review see [Munro 2005].

1.2 Scope of the thesis

We study some fundamental geometric optimization problems for space-efficient algorithms. Apart from theoretical interest, all of these problems have vast practical applications as well. The specific problems which are focused in this thesis are listed below.

**In-place min-max priority search tree:** One of the classic data structures for storing point sets in $\mathbb{R}^2$ is the priority search tree, introduced by McCreight in 1985. We show that this data structure can be made in-place, i.e., it can be stored in an array such that each entry stores only one point of the point set and no input point is stored in more than one location of that array. It combines a binary search tree with a heap. We show that all the standard query operations can be performed within the same time bounds as for the original priority search tree [McCreight 1985], while using only $O(1)$ extra-space.

We introduce the min-max priority search tree which is a combination of a binary search tree and a min-max heap. We show that min-max priority search tree can also be constructed and stored in an in-place manner, and all the standard queries which can be done in two separate versions of a priority search tree can be done with a single min-max priority search tree with $O(1)$ extra-space.

As an application, we present an in-place algorithm to enumerate all maximal empty axis-parallel rectangles amongst points in a rectangular region $R$ in $\mathbb{R}^2$ in $O(m \log n)$ time with $O(1)$ extra-space, where $m$ is the total number of maximal empty rectangles. The details of this work is described in Chapter 3.
1.2. Scope of the thesis

**Space-efficient prune-and-search:** Prune-and-search is an excellent algorithmic paradigm for solving various optimization problems. In Chapter 4, we provide a general scheme for prune-and-search technique and show how to implement it in space-efficient manner. We consider both the in-place and read-only model. Our techniques can be applied to a large number of problems that are amenable to a prune-and-search base solution. For examples, we study the following problems each of which has tremendous practical usage apart from theoretical implication:

- computing the minimum enclosing circle (MEC) of a set of points in $\mathbb{R}^2$,
- the convex hull of a set of points in $\mathbb{R}^2$ when these points are sorted with respect to their $x$-coordinates, and
- linear programming problems with two and three variables.

The time and extra-space complexities of the proposed algorithms for all these problems are $O(n \text{ polylog}(n))$ and $O(\text{polylog}(n))$ respectively. The details of these are described in Chapter 5, 6 and 7, respectively.

**Maximum clique of geometric intersection graphs:** The intersection graph of axis-parallel rectangles and the disk graph play an important role in formulating different problems in sophisticated database query, VLSI physical design, mobile ad hoc network etc.. In Chapter 8, we concentrate on designing in-place algorithms for finding the maximum clique on several variations of the intersection graphs of axis-parallel rectangles and disks in 2D. We first propose $O(n^2 \log n)$ time in-place algorithms for finding the maximum clique of the intersection graphs of a set of axis-parallel rectangles of arbitrary sizes. For the rectangle intersection graph of fixed height rectangles, the time complexity can be slightly improved to $O(n \log n + nK)$, where $K$ is the size of the maximum clique which can be $O(n)$ in the worst case. For disk graphs, we consider two variations of the maximum clique problem, namely geometric clique and graphical clique. The time complexity of our algorithm for finding the largest geometric clique is $O(n^2 \log n)$, and it works for disks of arbitrary radii. For graphical clique, our proposed algorithm works for unit disks (i.e., of same radii) and the worst case time complexity is $O(n^2 + mK^3)$, where $m$ is the
number of edges in the unit disk intersection graph, and $K$ is the size of the largest clique in that graph. It uses an $O(n^3)$ time in-place algorithm for the computation of maximum matching in a bipartite graph, which is of independent interest. All these algorithms need $O(1)$ extra-space in addition to the input array.

**Discrete piercing set for unit disks:** In Chapter 9, we consider constant factor approximation algorithms for a variation of the discrete piercing set problem for unit disks. Here a set of points $P$ is given; the objective is to choose minimum number of points in $P$ to pierce all the disks of unit radius and centered at all the points in $P$. This problem is also known as *minimum dominating set problem*. We propose a very simple algorithm that produces 12-approximation result in $O(n \log n)$ time. Next, we improve the approximation factor to 4 and then to 3. The worst case running time of these algorithms are $O(n^8 \log n)$ and $O(n^{15} \log n)$ respectively. All these algorithms can be made in-place without any degradation in their asymptotic running time. Finally, we propose a PTAS for our problem with worst case time complexity $n^{O(\frac{1}{\epsilon})}$ which improves the best known result $n^{O(\frac{1}{\epsilon^2})}$ on the time complexity of the problem [Gibson 2010, Mustafa 2010]. Note that we could not suggest an in-place implementation of the proposed PTAS for the problem.

### 1.3 Organization of the thesis

The thesis is organized in ten chapters. In Chapter 2, we give a detailed review of related works. In each of the following seven chapters (i.e., Chapter 3 to Chapter 9), we discuss one specific problem within the scope of the thesis. Finally, in Chapter 10, concluding remarks on our studies and possible future directions of research are illuminated.
Space-efficient algorithms are studied for a very a long time in the context of both theoretical interest and practical applications. In this chapter, we survey the existing space-efficient algorithms for fundamental problems like sorting, searching, some geometric problems and graph theoretic problems.
2.1 Fundamental problems

Fundamental algorithms like sorting, selection, partitioning, merging are exhaustively studied in the space-efficient model from a long time, some of them even dating from the year 1969 [Kronrod 1969], where an in-place algorithm for merging a pair of sorted sub-arrays was proposed.

**Sorting:** The classic data structure heap is a beautiful example of in-place data structure. Heapsort which is covered in any standard textbook (for example, see [Cormen 2009]) is also in-place. Most of the common sorting algorithms like bubble sort, selection sort, insertion sort, shell sort can be implemented in an in-place manner easily. On the other hand, quick-sort is not in-place as it requires $O(\log n)$ extra-space for recursion; it is an in-situ algorithm. Katajainen and Pasanen [Katajainen 1992] proposed how 0/1 stable sorting can be computed in linear time in an in-place manner. Huang and Langston [Huang 1992] presented an $O(n \log n)$ time $O(1)$ extra-space stable sorting procedure where each key can appear reasonable number of times.

A practical in-place algorithm for merge-sort was proposed by Katajainen, Pasanen and Teuhola [Katajainen 1996]. It takes $O(n \log n)$ comparisons and $O(n \log n)$ element movements. The first in-place sorting algorithm performing $O(n \log n)$ comparisons and $O(n)$ element moves in the worst case was given by Franceschini and Geffert [Franceschini 2005]. Franceschini, Mukthukrishnan and Pătraşcu [Franceschini 2007] proposed a simple, stable, integer sorting (in-place) algorithm for words of size $O(n \log n)$, which works in $O(n)$ time using only $O(1)$ extra-space.

On the other hand, if the elements are given in a read-only array, then one can report the smallest element in $O(n)$ time by scanning the entire array. The second smallest element can be reported in the next scan. In this way all sorted elements can be reported in $O(n^2)$ time using $O(1)$ extra-space.

If $O(s)$ (where $s < n$) extra-space is provided, then an easy sub-quadratic time sorting algorithm for read-only memory is as follows.

Scan the entire array $O(\frac{n}{s})$ times. After each scan, report $s$ elements. So, at a particular scan it ignores all the elements whose value is less
than the last reported element of the previous scan. In each scan, it maintains a heap of \( s \) elements to identify the smallest \( s \) elements among the set of unreported elements. The time complexity of this algorithm is \( O(n^2 \log \frac{n}{s}) \).

Frederickson [Frederickson 1987] gave an \( O(\frac{n^2}{s} + n \log s) \) time algorithm for the sorting problem which uses \( O(s) \) extra-space apart from the read-only input array. Later Pagter and Rauhe [Pagter 1998] improved this upper bound which achieves the lower bound on the time-space product stated by Borodin et al. [Borodin 1981].

**Selection:** One of the most studied problems in this setting is the selection problem [Frederickson 1987, Munro 1996, Raman 1999]. Carlsson and Sundström [Carlsson 1995] showed that if a set of \( n \) real values are given in an array, then the problem of computing the median in an in-place environment can be solved using at most \( 3n \) comparisons with \( O(1) \) extra-space.

If the array is read-only (i.e., swapping two values in the input array is prohibited), then an easy way to compute the median using \( O(\sqrt{n}) \) extra-space is as follows.

Split the array into \( \lceil \sqrt{n} \rceil \) blocks each containing \( \lfloor \sqrt{n} \rfloor \) consecutive elements. Compute the median of each part in an in-place manner copying it in a separate array of size \( \lfloor \sqrt{n} \rfloor \) and store these block medians in another array of size \( \lceil \sqrt{n} \rceil \). Now calculate the median of the block medians in an in-place manner and disregard at-least \( \frac{1}{4} \) fraction of the elements for the next iteration. This needs \( O(\log n) \) iterations. But in each iteration, we need to scan the entire input array to consider the relevant elements for that iteration. Thus, the overall time complexity is \( O(n \log n) \).

Munro and Paterson [Munro 1980] showed that to find median in two passes, it requires at least \( \Omega(\sqrt{n}) \) and at most \( O(\sqrt{n \log n}) \) extra-space. Munro and Raman [Munro 1996] proved that if the array is read-only then the median can be found in \( O(n^{1+\varepsilon}) \) time using \( O(\frac{1}{\varepsilon}) \) extra-space, where \( \varepsilon \) is a small \((< 1)\) positive number to be fixed prior to the execution. Later Raman and
Ramnath [Raman 1999] proposed algorithms for various ranges of extra-space as follows:

(i) an $O(n \log^2 n)$ time algorithm using $O(\log n)$ extra-space,

(ii) an $O(n \log^2 n \log_2 n \log p)$ time algorithm using $O(\frac{\log n}{\log p})$ extra-space, where $p \geq 2$ is any integer parameter,

(iii) an $O(n \log n + n \log^2 n \log_2 p)$ time algorithm using $O(\frac{p^2 \log n}{\log p})$ extra-space, where $p \geq 2$ is any integer parameter.

Very recently, Elmasry et al. [Elmasry 2013] proposed an $O(n)$ time algorithm for this problem which takes only $O(n)$ bits apart from the read-only input. Chan [Chan 2010a] has shown that if $s$ extra-bits are given in addition to the input read-only array, then a lower bound on the expected time complexity of finding the median is $\Omega(n \log \log s \cdot n)$. This result rules out the possibility of an $O(n)$ time (randomized or deterministic) selection algorithm with poly-logarithmic extra-space.

2.2 Geometric problems

Brönnimann et al. [Brönnimann 2002] first started investigating space-efficient algorithms for geometric problems. After that a number of results appeared in this line of work. Here, we state the results of some important problems.

2.2.1 Range searching

Given a set of objects $O$, determining which objects of $O$ intersect with a query range $R$ is known as the range searching problem. In general, this type of problem is solved with some preprocessing of the input objects. The objects may be points, line-segments, polygons etc. Also, the query range may be different types, like axis-aligned rectangle, simplex, half-space, three-sided orthogonal range, sphere etc. [de Berg 2008].

Given a set of points in $\mathbb{R}^2$, a half-space range counting query can be performed using Willard’s partition tree [Edelsbrunner 1986]. Brönnimann et al.
2.2. Geometric problems

[Brönnimann 2004a] showed that partition tree can be constructed and stored in an in-place manner and answering a query takes $O(n^{.793})$ time and $O(1)$ extra-space.

Given a set of points in $\mathbb{R}^d$, one can answer a simplex and half-space range searching query using Matousek’s [Matousek 1992] near-optimal partition tree. Brönnimann et al. [Brönnimann 2004a] showed how to make this data structure in-place such that it can answer those queries. The complexity results are as follows.

Given an array of $n$ points in $\mathbb{R}^d$, one can permute the array in expected $O(n \log n)$ time using $O(1)$ extra-space so that (i) a half-space range query can be answered in $O(n^{1-1/\lfloor d/2 \rfloor + \varepsilon} + k)$ time and $O(1)$ extra-space, where $k$ is the number of reported answers, and (ii) a simplex range query can be answered in $O(n^{1-1/d + \varepsilon} + k)$ time and $O(1)$ extra-space.

Orthogonal range searching has a vast literature [Agarwal 2004, Agarwal 1998a]. Given a set of points in $\mathbb{R}^d$, the objective is to preprocess the points so that given a $d$-dimensional box, the points inside the box can be reported efficiently. Brönnimann et al. [Brönnimann 2004a] started looking into the problem in the context of space-efficient computation, where they indicated the possibility of in-place maintenance of 2d-tree with a set of points in $\mathbb{R}^2$ in the same input array without using any extra-space, and orthogonal range queries can be performed in $O(\sqrt{n})$ time.

In this thesis, we have studied three-sided orthogonal range query in $\mathbb{R}^2$. Given a set of points in $\mathbb{R}^2$, one can answer three-sided orthogonal range queries using McCreight’s [McCreight 1985] priority search tree in $O(\log n)$ time. This data structure uses $O(n)$ extra-space apart from the input. In Chapter 3, we show how one can construct a priority search tree in an in-place manner. This in-place priority search tree takes no extra-space apart from the array containing the input points. We also describe how one can answer different variations of three-sided orthogonal range queries using the in-place priority search tree in $O(\log n)$ time in an in-place manner. In a similar way, a kd-tree can be constructed and stored in an in-place manner, and can be used for answering four-sided orthogonal range queries.

Recently, Farzan et al. [Farzan 2012] studied the orthogonal range maxima
problem in $\mathbb{R}^2$ in a low space environment. Here, each point is associated with a priority. The objective is to report the point with highest priority inside an axis-parallel query box. They proposed a succinct way of maintaining the data structure using $O(n)$ words so that the query can be answered in $O(\log n \log \log n)$ time. This improves Chazelle’s result [Chazelle 1988] that requires $O(n^{1+\varepsilon})$ words to answer the query in $O((\log n)^{1+\varepsilon})$ time, where $0 < \varepsilon < 1$.

2.2.2 Convex hull

The convex hull of a set of points $P$ is defined as the minimal convex set containing $P$. Because of its practical and theoretical significance, the convex hull is one of the most extensively studied problem in space-efficient models. Brönnimann et al. [Brönnimann 2004b] showed that using Graham scan with in-place sorting, the convex hull of a set of $n$ points in $\mathbb{R}^2$ can be computed in an in-place manner in $O(n \log n)$ time. Here, the extra-space required is $O(1)$, and the output is available in the same input array. The main idea is to use the portion of the input array as a stack. In the same paper they also showed that Chan’s [Chan 1996] optimal output sensitive algorithm for computing the convex hull of a point set in 2D can be made in-place using $O(n \log h)$ time and with $O(1)$ extra-space where $h$ is the number of hull vertices. Recently, Vahrenhold [Vahrenhold 2012] showed that the prune-and-search algorithm by Kirkpatrick and Seidel [Kirkpatrick 1986] for computing the convex hull of a planar point set can also be made in-place maintaining the $O(n \log h)$ time complexity and using only $O(1)$ extra-space.

For the 3D convex hull, Chan and Chen [Chan 2010b] proved the existence of $O(n \log h)$ expected time randomized in-place algorithms, where $h$ is the number of hull vertices. In the same paper, they proposed a deterministic in-place algorithm for the 3D convex hull that runs in $O(n \log^2 n)$ time.

If a set of $n$ points in $\mathbb{R}^2$ is given in a read-only array, then the problem of computing the convex hull becomes difficult. Here, using well-known Jarvis March algorithm, one can compute the convex hull in $O(nh)$ time with $O(1)$ extra-space.
2.2. Geometric problems

If a simple polygon with \( n \) vertices is given in a read-only memory, then Barba et al. [Barba 2013] showed that the convex hull can be computed in \( O(n \log n) \) time with \( O(n \log n) \) extra-space, or \( \frac{n^2}{2^n} \) time with \( O(s) \) extra-space, where \( s \in o(\log n) \).

Chan and Chen [Chan 2007] considered the problem of computing the convex hull where the points are sorted by their \( x \)-coordinates in the read-only setting. They proposed a randomized algorithm which runs in \( O(\delta n) \) expected time and \( O(\delta n^\delta) \) extra-space for any fixed \( \delta > 0 \). They showed that the algorithm can be made deterministic if the running time is increased to \( O(2^{O(\delta n)}) \). In Chapter 6, we propose an \( O(n \log^5 n) \) time deterministic algorithm for this restricted problem which takes \( O(\log^2 n) \) extra-space.

2.2.3 Nearest neighbor problems

Proximity problems or closest point problems are very important problems in computational geometry. Given a set of \( n \) points in \( \mathbb{R}^2 \), the closest pair of points is defined as \( \min_{p_i, p_j \in P} d(p_i, p_j) \), where \( d(p_i, p_j) \) denotes the Euclidean distance between the points \( p_i \) and \( p_j \). Without any space restriction, many \( O(n \log n) \) algorithms are known for this problem. Bose et al. [Bose 2007] proposed an in-place divide-and-conquer algorithm to compute the closest pair of points in \( O(n \log n) \) time using \( O(1) \) extra-space. Chan and Chen [Chan 2008] described a data structure which uses no extra-storage other than the input array. Using this data structure, they showed that given any point \( p \in P \), the nearest neighbor query for \( p \) can be answered in \( O(\log^{1.71} n) \) time in an in-place manner. It is to be mentioned that though the data structure does not require extra storage, but the preprocessing algorithm is not in-place. Here, the preprocessing is done using extra storage to create the final data structure which is a permutation of the input points. Thus, this can be stored in the input array itself.

Given a set \( P \) of red points and another set \( Q \) of blue points, bichromatic closest pair is defined as \( \min_{p \in P, q \in Q} ||p - q|| \). Bose et al. [Bose 2007] gave a randomized \( O(n \log n) \) time \( O(1) \) extra-space in-place algorithm for this problem. Brönnimann et al. [Brönnimann 2004a] showed how one can compute it deterministically in \( O(n \log^2 n) \) time using \( O(\log^2 n) \) extra-space in
an in-place manner. In the same paper, they proposed an $O(n \log^2 n)$ time $O(\log^2 n)$ extra-space in-place algorithm to report the set of pairs \( \{p_q, q\} \) for all $q \in Q$, where $p_q$ denotes the nearest red neighbor of the blue point $q$.

2.2.4 Klee’s measure problem

Given a set of $d$-dimensional hyper-rectangles, calculating their union measure (volume) in $d$-dimension is known as Klee’s measure problem. Two dimensional version of this problem was optimally solved in $O(n \log n)$ time using $O(n)$ extra-space by Overmars and Yap [Overmars 1991].

Chen and Chan [Chen 2005] first proposed a space-efficient algorithm for this problem which takes $O(n \sqrt{n} \log n)$ time using $O(\sqrt{n})$ extra-space. Later, Vahrenhold [Vahrenhold 2007a] gave an in-place algorithm which takes $O(n \sqrt{n} \log n)$ time and $O(1)$ extra-space. The same algorithm solves the Klee’s measure problem for intervals on a real line in $O(n \log n)$ time with $O(1)$ extra-space.

2.2.5 Line segment intersection

In the line segment intersection problem, a set of $n$ line segments is given and all $k$ intersections induced by these line segments are desired to be reported. Bentley and Ottmann [Bentley 1979] gave a practical output sensitive algorithm for this problem which takes $O((n+k) \log n)$ time and $O(n)$ extra-space.

Chen and Chan [Chan 2010b] were the first to give space-efficient algorithm for this problem. They proposed $O((n+k) \log^2 n)$ time $O(\log^2 n)$ extra-space in-place algorithm and a destructive in-place algorithm which takes $O((n+k) \log n)$ time using $O(1)$ extra-space. Finally, Vahrenhold [Vahrenhold 2007b] proposed an $O(n \log^2 n + k)$ time and $O(1)$ extra-space in-place algorithm for this problem.

2.2.6 Slope selection

Given a set of points $P$, the slope selection problem is to select the line with $k$-th slope considering all lines induced by the pair of points in $P$. Optimally, the problem can be solved in $O(n \log n)$ time by a deterministic algorithm.
2.2 Geometric problems

[Cole 1989]. Blunk and Vahrenhold [Blunk 2006] were the first to propose an in-place algorithm for this problem which is randomized in nature. The running time of the algorithm is expected $O(n \log n)$ and it uses $O(1)$ extra-space.

2.2.7 Visibility and shortest path problems in simple polygon

Visibility is one of the foundational areas in computational geometry and it has applications in various domains, including robot motion planning, guarding art galleries, computer graphics, GIS, sensor network. Visibility algorithms are being embedded in the hardware of digital cameras, sensors, etc, where the constraint on the size of the instrument is very important. Thus, space-efficient algorithms for visibility problems always attract special attention.

The most fundamental problem is to report the visibility of a point inside a simple polygon. The lower bound on time is $\Omega(n)$ as we need to test all the vertices of the input polygon. Without any space restriction, the best-known algorithm for this problem runs in $O(n)$ time using $O(n)$ space [Lee 1983, Joe 1987, Ghosh 2007].

If the polygon is given in a read-only array, then in quadratic time one can find the visible vertices of a point using $O(1)$ extra-space. Two algorithms for this problem are proposed by Barba et al. [Barba 2011]. The first one is deterministic, and it requires $O(n \pi)$ time and $O(1)$ extra-space, where $\pi$ is the number of reflex vertices of the output visibility polygon. The second one is an algorithm which takes randomized $O(n \log r)$ time or deterministic $O(n \log^2 r)$ time using $O(\log r)$ space, where $r$ is the number of reflex vertices in the input polygon. It is to be noted that in the worst case both $r$ and $\pi$ can be $O(n)$. Later Barba et al. [Barba 2013] proposed algorithm for this problem which takes $O(n \log^2 n \log p)$ time $O(p \log n \log p)$ extra-space, where $2 \leq p \leq n$, or $O(n^2)$ time $O(s)$ extra-space, where $s \in o(\log n)$.

In the context of shortest path in simple polygon in read-only environment, the first result is proposed in [Asano 2011b, Asano 2011c]. Given the simple polygon in a read-only array, it is shown that the shortest geodesic path be-
tween a pair of points can be computed in $O(n^2)$ time using $O(1)$ extra-space. The query version of the shortest path problem is studied in [Asano 2013]. They showed that given $O(s)$ extra-space, the polygon can be preprocessed in $O(n^2)$ time, and given two arbitrary query points inside the polygon, the shortest path among them can be obtained in $O(n^2/s)$ time.

### 2.2.8 Minimum enclosing circle

The minimum enclosing circle problem has a long history. The problem is defined as follows. Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in $\mathbb{R}^2$, the objective is to compute the minimum radius circle containing all the points in $P$. In other words, we need to locate a point $\pi$ in $\mathbb{R}^2$ such that $\max_{i=1}^n d(p_i, \pi)$ is minimized, where $d(a, b)$ is the distance of two points $a$ and $b$ in some chosen metric. The problem is known as 1-center problem in the literature, and it was first proposed by Sylvester [Sylvester 1857]. Elzinga et al. with their work [Elzinga 1972] paved the way for solving this minimax problem with elementary geometry, and proposed an $O(n^2)$ time algorithm for the Euclidean 1-center problem for a point set $P$, where $|P| = n$. Note that (i) the MEC for the point set $P$ is the same as the MEC for the convex hull of $P$ (denoted by $CH(P)$), (ii) the center of the MEC of $P$ is either on the mid-point of a diameter of $CH(P)$ or one of the vertices of the farthest point Voronoi diagram of $P$ (denoted by $FVD(P)$), and (iii) $FVD(P) = FVD(CH(P))$. Since both computing $CH(P)$ and $FVD(P)$ need $O(n \log n)$ time [Preparata 1990], we have an obvious $O(n \log n)$ time algorithm for computing the MEC of the point set $P$. The best known result for computing the MEC is by Megiddo [Megiddo 1983b]. It uses a prune-and-search technique; the time and space complexities of this algorithm are both $O(n)$. Later Welzl [Welzl 1991] proposed an easy to implement randomized algorithm for computing the MEC that runs in expected $O(n)$ time. For the weighted version of the MEC problem, the first result is also by Megiddo [Megiddo 1983c] that runs in $O(n(\log n)^3(\log \log n)^2)$ time using the parametric search [Megiddo 1983a]. Later, Megiddo and Zemel [Megiddo 1986] proposed an $O(n \log n)$ time randomized algorithm for this problem that does not use parametric search. Finally, the weighted version of the MEC problem is also solved by Megiddo in
linear time [Megiddo 1989]. All of these algorithms use \( O(n) \) extra-space.

In the read-only model, the proximity problems in \( \mathbb{R}^2 \) are studied by Asano et al. [Asano 2011b]. If the input points are given in a read-only array, all the Delaunay triangles can be reported in \( O(n^2) \) time using \( O(1) \) extra-space. In the same paper, they mentioned that the farthest point Voronoi diagram can also be computed in \( O(n^2) \) time using \( O(1) \) extra-space. Thus, the same complexity results hold for finding the maximum empty circle and minimum enclosing circle problems in the read-only environment. The authors of [Asano 2011b] posed an open question of solving both the problems in sub-quadratic time using sub-linear extra-space in the read-only environment. In Chapter 5, we answered the question regarding the minimum enclosing circle positively.

In this connection, it needs to be mentioned that Chan and Pathak [Chan 2011] proposed an algorithm for computing an approximate minimum enclosing ball in \( \mathbb{R}^d \). It scans the input data only once, takes \( O(n \log n) \) amortized time and uses \( O(d) \) extra-space to compute an enclosing ball of a set of \( n \) points in \( \mathbb{R}^d \), that is of size \( \frac{1+\sqrt{3}}{2} \text{OPT} \), where \( \text{OPT} \) is the size of minimum enclosing ball. This is an improvement of the earlier result on the same problem that uses \( O(d) \) extra-space, executes a linear scan and reports a solution of size \( \frac{1+\sqrt{3}}{2} \text{OPT} \) [Agarwal 2010].

### 2.2.9 Linear programming

Linear programming which is a method for the optimization of a linear objective function, subject to a set of linear equality and linear inequality constraints, has tremendous application in business, economics, engineering optimization problems etc.. Formally, the d-dimensional linear programming problem can be stated as following:

\[
\begin{align*}
\min_{x_1, x_2, \ldots, x_d} & \quad c_1 x_1 + c_2 x_2 + \ldots + c_d x_d \\
\text{subject to} & \quad a_{1i} x_1 + a_{2i} x_2 + \ldots + a_{di} x_d \geq \beta_i, \ i \in I = \{1, 2, \ldots n\}.
\end{align*}
\]

Or equivalently,

\[
\begin{align*}
\min & \quad CX, \\
\text{subject to} & \quad AX \geq \beta,
\end{align*}
\]
where $X$ is the variable vector, $A$ is the coefficient matrix, $C$ and $\beta$ are the cost vector and requirement vector, respectively.

Linear programming problem can be solved in polynomial time [Karmarkar 1984]. Megiddo [Megiddo 1984] gave an algorithm for fixed $(d)$ dimensional linear programming problem which runs in $O(n)$ time using $O(n)$ space, where $n$ is the number of constraints in the problem. Here, the constant involved in the order notation is exponential in $d$.

Brönnimann et al. [Brönnimann 2002] proposed an in-place linear time algorithm for 2D linear programming problem with $O(1)$ extra-space. Chan and Chen [Chan 2007] proposed an $O(n)$ time randomized algorithm for the linear programming problem in fixed dimension using $O(\log n)$ extra-space in a read-only environment, where the matrix $A$ and the vectors $C$ and $\beta$ are given in read-only array.

Assuming that the constraints of the linear programming problem are given in a read-only array, we show that in deterministic $O(n \, \text{polylog}(n))$ time using $O(\text{polylog}(n))$ extra-space, 2D and 3D linear programming can be solved. We also show that in the in-place model one can compute those problems in $O(n)$ time with additional $O(1)$ extra-space. The details of these results are given in Chapter 7.

2.3 Graph theoretic problems

A lot of study is performed in solving different graph-theoretic problems considering the read-only input. But most of the research was carried out in proving the existence of an algorithm which will take $O(\log n)$ bits or $O(\log^2 n)$ bits (i.e $O(1)$ or $O(\log n)$ extra-space, respectively) apart from the read-only input array; less care is given on analyzing the running time.

In early 80’s, JáJá and Simon [JáJá 1982] showed that planarity testing, producing a plane embedding of planner graphs, finding minimum cost spanning trees, obtaining the connected, biconnected and triconnected components of a graph can be done using $O(\log n)$ extra-space apart from the read-only input.

Reingold [Reingold 2005] solved a long standing open problem by showing that S-T connectivity in an undirected graph is in log-space, i.e., it can be computed using $O(1)$ extra-space.
Combining the result of Reingold [Reingold 2005] with the result of Allender and Mahajan [Allender 2004], it is proved that whether a graph, given in a read-only memory, is planar or not can be decided using $O(1)$ extra-space. Later, Datta and Prakriya [Datta 2011] proved that it is possible to construct the planar embedding of a graph (if it is planar), given in a read-only memory, using only $O(1)$ extra-space.

Datta et al. [Datta 2009] showed that planar graph isomorphism is in log-space. In other words, given two planar graphs one can decide whether they are isomorphic to each other using only $O(1)$ extra-space.

For shortest path, maximum matching and reachability problems for general graph given in a read-only memory, no deterministic algorithm using $O(1)$ extra-space is known till date.

Asano et al. [Asano 2011c] showed that shortest path in a tree can be computed in linear time using $O(1)$ extra-space, where the tree is represented as doubly-connected edge list in a read-only memory. Asano and Doerr [Asano 2011a] presented an $O(n^{2+\frac{2+1}{\sqrt{m}+\ell}})$ time and $O(n^{\frac{1}{\sqrt{m}+\ell}}\sqrt{n})$ extra-space algorithm which can output the shortest path between any two vertices of a grid graph of size $O(\sqrt{n}) \times O(\sqrt{n})$ with positive edge weights where the input is given in a read-only memory.

If a planar straight line graph is given in a read-only memory, then its triangulation can be reported in $O(n^2)$ time using $O(1)$ extra-space [Asano 2013]. Thus, the triangulation of a simple polygon can also be done with the same complexity results when the vertices of the polygon are given in order in a read-only array.

Maximum matching for a bipartite graph is an important problem. Datta et al. [Datta 2012] showed that if a bipartite graph can be embedded on the surface of a constant genus, then both (i) existence of a perfect matching and (ii) computation of maximum matching can be performed in polynomial time using $O(\log n)$ extra bits. For bipartite planar graphs, the problem of checking the existence of a perfect matching is in $UL$ (the class of problems decidable by an unambiguous, nondeterministic, log-space bounded machine) [Datta 2010]. In the same paper, it is also shown that checking the existence of perfect matching in a bipartite planar graph is in $SPL$ (the class of problems which
are log-space reducible to the problem of checking whether the determinant of a matrix is 0 or not under the promise that the determinant is either 0 or 1).

For a bipartite graph, it was shown that the problem of checking the existence of a perfect matching is known to be hard for \( NL \) (the complexity class containing decision problems which can be solved by a nondeterministic Turing machine using a logarithmic amount of memory space) [Chandra 1984].

In Chapter 8.6.1, we present an \( O(n^3) \) time \( O(1) \) extra-space in-place algorithm for computing maximum matching in a bipartite graph \( G = (V_1, V_2, E) \) where the two sets of nodes \( V_1 \) and \( V_2 \) are stored in two arrays, and the existence of an edge between a pair of nodes can be checked on demand by an oracle in \( O(1) \) time.

### 2.3.1 Maximum clique and independent set of geometric intersection graphs

The geometric intersection graph \( G = (V, E) \) of a set of geometric objects \( S \) is a graph whose nodes \( V \) correspond to the set of objects in \( S \). Between a pair of nodes \( v_i \) and \( v_j \), there is an edge \( (v_i, v_j) \) if the corresponding objects in \( S \) intersect. The intersection of a pair of objects is defined depending on the problem specification. For example, sometimes proper containment is considered to be an intersection and sometimes it is not.

In sophisticated database query and VLSI physical design, several optimization problems are formulated using the intersection graph of axis-parallel rectangles [Asano 1986]. Similarly, the disk graph plays an important role in formulating different problems in mobile ad hoc network [Li 2003]. From now onwards, by rectangle intersection graph we will mean the intersection graph of axis-parallel rectangles.

The first polynomial time algorithm for finding the maximum clique in a rectangle intersection graph was proposed in [Lee 1982]. The best known algorithm for this problem runs in \( O(n \log n) \) time and \( O(n) \) space [Imai 1983, Nandy 1995]. The best known algorithm for finding the maximum clique region of a set of axis-parallel rectangles in \( d \)-dimensional space runs in \( O(n^{d-1}) \) time [Lee 1983].
Let us consider the intersection graph of a set of disks. Let $C$ be a subset of disks such that each pair of members in $C$ intersect. In the aforesaid graph the nodes corresponding to $C$ define a clique. However, the members of $C$ may not have a common intersection. Thus, a clique in a disk graph is usually referred to as *graphical clique*. In particular, if the members in a clique have common intersection region, then that clique is referred to as a *geometric clique*. Clark et al. [Clark 1990] proposed an $O(n^{4.5})$ time algorithm for computing the largest (graphical) clique in a unit disk graph. Later the time complexity of the problem was improved to $O(n^{3.5} \log n)$ [Breu 1996]. Given a set of points in $\mathbb{R}^2$, the problem of finding the position of a unit disk to contain maximum number of points can be mapped to finding the maximum geometric clique of a unit disk graph. Using the plane sweep method, this problem can also be solved in $O(n^2)$ time using $O(n^2)$ extra-space [Chazelle 1986b].

In Chapter 8, we first propose an in-place $O(n \log n)$ time algorithm for computing the maximum clique of an intersection graph of a set of $n$ intervals on the real line. We use this algorithm to design an in-place algorithm for finding the maximum clique of the intersection graph of a set of $n$ axis-parallel rectangles of arbitrary size in $O(n^2 \log n)$ time. For fixed height rectangles, the time complexity can be improved to $O(n \log n + nK)$, where $K$ is the size of the largest clique.

Next, we consider the maximum clique problem for the disk graph. Our proposed in-place algorithm for computing the largest geometric clique of the intersection graph of a set of disks of arbitrary radii needs $O(n^2 \log n)$ time. For graphical clique, our in-place algorithm works for unit disks only, and it runs in $O(n^2 + mK^3)$ time, where $n$ and $m$ are the number of vertices and edges in the unit disk graph, and $K$ is the size of the maximum clique in that graph.

We also show that the 2-factor approximation algorithm by Agarwal et al. [Agarwal 1998b] for computing the maximum independent set of a set of axis-parallel rectangles of fixed height can be made in-place easily, and it runs in $O(n \log^2 n)$ time using $O(1)$ extra-space.

Similarly, the 5-factor approximation algorithm of Marathe et al. [Marathe 1995] for computing the maximum independent set of a set of disks of arbitrary radii
can also be made in-place, and it runs in $O(n^2)$ time. A minor perturbation of the algorithm produces a 3-factor approximation solution for disks with unit radii.

2.3.2 Discrete piercing set for unit disk graph

The piercing set of a set of objects $S$ in $\mathbb{R}^2$ is a set of points $Q$ such that each object in $S$ contains at least one point in $Q$. Here, the optimization problem is to compute a piercing set for $S$ of minimum size. A clique $C$ in the geometric intersection graph $G$ of a set of objects $S$ implies that each pair of objects corresponding to the nodes in $C$ are intersecting. But, it does not imply that all of them have a common intersection region. In other words, a clique $C$ in $G$ does not imply that the objects corresponding to the members in $C$ can be pierced by a single point. However, if $S$ consists of a set of axis-parallel rectangles, then the minimum piercing set corresponds to the minimum clique cover of the intersection graph of the members in $S$.

The minimum clique cover problem for a set of axis-parallel unit squares in $\mathbb{R}^2$ is known to be NP-hard [Garey 1979]. Hochbaum and Maass [Hochbaum 1985] proposed a PTAS for the minimum clique cover problem for a set of axis-parallel unit squares with time complexity $n^{O(1/\varepsilon^2)}$. The time complexity was later improved to $n^{O(1/\varepsilon)}$ by Feder and Greene [Feder 1988], and then by Gonzalez [Gonzalez 1991]. Chan [Chan 2003] proposed a PTAS for squares of arbitrary size with time complexity $n^{O(1/\varepsilon^2)}$. In fact, this algorithm works for any collection of fat objects. Chan and Mahmood [Chan 2005] considered the problem for a set of axis-parallel rectangles of fixed height (but of arbitrary width), and proposed a PTAS with $n^{O(1/\varepsilon^2)}$ time complexity.

The minimum clique cover problem for unit disk graph also has a long history. The problem is known to be NP-hard [Cerioli 2004], and a 3-factor approximation algorithm is easy to obtain [Marathe 1995]. Recently, Dumitrescu and Pach [Dumitrescu 2011] proposed an $O(n^2)$ time randomized algorithm for the minimum clique cover problem with approximation ratio 2.16. They also proposed a polynomial time approximation scheme (PTAS) for this problem that runs in $O(n^{1/\varepsilon^2})$ time. It improves on a previous PTAS with $O(n^{1/\varepsilon^4})$ running time [Pirwani 2010].
As mentioned earlier, the minimum piercing set problem and the minimum clique cover problem are different for unit disk graph. The minimum piercing set problem for disks has many applications in wireless communication where the objective is to place the base stations to cover a set of radio terminals (sensors) distributed in a region. The minimum piercing set problem for unit disks is also NP-hard [Carmi 2007, Dumitrescu 2011]. Carmi et al. [Carmi 2007] proposed an approximation algorithm for this problem where the approximation factor is 38. In particular, if the points are distributed below a straight line $L$, and the base stations (of same range) are allowed to be installed on or above $L$ then a 4-factor approximation algorithm can be obtained provided each point below $L$ lie within a unit distance from at least one base station.

In the discrete version of the minimum piercing set problem for unit disks, two sets of points $P$ and $Q$ are given. The unit disks are centered at the points of $P$, and the piercing points need to be chosen from $Q$. The objective is to choose minimum number of points from $Q$ to pierce the disks centered at all the points in $P$. The problem is known to be NP-hard [Johnson 1982]. The first constant factor approximation result on this problem is proposed by Calinescu et al. [Calinescu 2004]. It uses linear programming relaxation method to produce an 108-factor approximation result. The approximation result is then improved to 72 in [Narayanappa 2006], 38 in [Carmi 2007], and 22 in [Claude 2010]. Finally, Das et al. [Das 2011] proposed an 18-factor approximation algorithm that runs in $O(n \log n + m \log m + mn)$ time, where $|P| = n$ and $|Q| = m$.

Another variation of the discrete piercing set problem for unit disks assumes $Q = P$. In other words, the unit disks corresponding to the points in $P$ need to be pierced by choosing minimum number points from $P$ itself. In the literature, the problem is referred to as the minimum dominating set problem for the unit disk graph (or MDS problem in short). Here, an undirected graph is constructed with nodes corresponding to the points in $P$. Between a pair of nodes there is an edge if the distance between the two points is less than or equal to their common radius. A vertex in the graph dominates itself and all its neighbors. The objective is to choose minimum number of vertices to dominate all the vertices in the graph.

The problem is known to be NP-hard [Clark 1990]. Ambühl et al. [Ambühl 2006]
first proposed an approximation algorithm for this problem. They con-
dered the weighted version of the problem where each node is attached with
a positive weight. The objective is to find the minimum weight dominating
set of the nodes in the graph. The approximation factor of their proposed
algorithm is 72. Huang et al. [Huang 2008] improved the approximation fac-
tor of the same problem to $6 + \varepsilon$. Dai and Yu [Dai 2009] further improved
the approximation factor to $5 + \varepsilon$. Recently, Zou et al. [Zou 2011] pro-
posed a polynomial time $4 + \varepsilon$ factor approximation algorithm. The theme
of the algorithms in [Huang 2008, Dai 2009, Zou 2011] are same. First, they
solve a sub-problem with approximation factor $\gamma$ ($\gamma = 6, 5, 4$ in [Huang 2008],
[Dai 2009] and [Zou 2011] respectively) and then using this result they solve
the original problem with approximation factor $\gamma + \varepsilon$. The time complexity
of their algorithms are of the form $O(\alpha(n) \times \beta(n))$, where $O(\alpha(n))$ is the time
complexity of the algorithm for the sub-problem and $O(\beta(n)) = O(n^{4(\lceil \frac{2\varepsilon}{\gamma} \rceil)^2})$
is the number of times the sub-problem needs to be invoked to solve the
corresponding original problem. Putting $\varepsilon = 1$, we can get $\gamma + 1$ factor ap-
proximation algorithm, but the time complexity becomes very high degree
degree polynomial function in $n$. Nieberg and Hurink [Nieberg 2005] proposed an
$O(n^2)$ time PTAS for computing the minimum dominating set for unit disk
graphs, where $c$ is an integer satisfying $(2c + 1)^2 < (1 + \varepsilon)^c$ for $0 < \varepsilon \leq 1$. It
accepts any undirected graph as input, and returns a $(1 + \varepsilon)$ factor approx-
imation solution for the dominating set problem, or a certificate indicating
that the input graph is not a unit disk graph. The fastest worst-case running
time is obtained by setting $\varepsilon = 1$ for a 2-approximation result, which runs in
$O(n^{\frac{c^2}{3}}) = O(n^{\frac{9^2}{2}}) = O(n^{81})$ time. Clearly, this algorithm is not tractable for
its time complexity. Recently, Gibson and Pirwani [Gibson 2010] proposed a
PTAS for minimum dominating set of (arbitrary size) disk graph which runs in
$n^{O(\frac{1}{\varepsilon^2})}$ time. Our present work is directed towards finding approximation
algorithms. For the unweighted version of the MDS problem, the best known
result is a 5-factor approximation algorithm proposed in [Carmi 2008], and it
works for disks of arbitrary radii. This result is then used for the $h$-piercing
problem, where the objective is to choose minimum number of points in $P$ to
pierce each disk by at least $h$ points. The proposed approximation factor is
$5(2^h - 1)$. 
In Chapter 9, we propose three almost similar type algorithms for the MDS problem (i.e., the discrete piercing problem with $Q = P$) assuming that square-root and ceiling operations can be done constant time. The first one produces a 12-approximation result in $O(n \log n)$ time. The second one produces a 4-approximation solution in $O(n^8 \log n)$ time, and the last one produces a 3-approximation solution in $O(n^{15} \log n)$ time. We can use this result to improve the approximation factor for the $h$-piercing problem [Carmi 2008] of constant radius disks to $3(2^h - 1)$ from $5(2^h - 1)$.

We also show that each of these algorithms can be made in-place with $O(1)$ extra-space keeping their time complexities invariant. This is of great advantage in the sense that the algorithms for different optimization problems in unit disk graph have wide applications in sensor network where the necessary softwares are embedded in tiny sensors, and the constraint in memory size is very much important due to their sizes.

Finally, we propose a PTAS for the discrete piercing set problem. Given a positive integer $k$, the proposed algorithm produces a solution with performance ratio $(1 + \frac{1}{k})^2$ in $n^{O(k)}$ time.
Chapter 3

An In-Place Min-Max Priority Search Tree

Contents

3.1 Introduction ................................................. 27
3.2 Constructing the in-place MinMaxPST .................. 31
  3.2.1 Improving the construction time .................... 35
3.3 Queries on the in-place MinMaxPST .................. 38
  3.3.1 LeftMostNE(x₀, y₀) ................................. 38
  3.3.2 Highest3SidedUp(x₀, x₁, y₀) .......................... 42
  3.3.3 Enumerate3SidedUp(x₀, x₁, y₀) ...................... 52
3.4 Enumerating all axis-parallel maximal empty rectangles ................................. 54
  3.4.1 Algorithm .................................................. 55
  3.4.2 Correctness and complexity ............................ 61
3.5 Conclusion .................................................... 62

3.1 Introduction

Let P be a set of n points in \( \mathbb{R}^2 \). A priority search tree, as introduced by McCreight [McCreight 1985], is a binary tree T with exactly one node for each point of P and that has the following two properties:
For each non-root node \( u \), the point stored at \( u \) has a smaller \( y \)-coordinate than the \( y \)-coordinate stored at the parent of \( u \).

For each internal node \( u \), the \( x \)-coordinate of every point in the left subtree of \( u \) is less than the \( x \)-coordinate of any point in the right subtree of \( u \).

The first property implies that \( T \) is a max-heap on the \( y \)-coordinates of the points in \( P \). The second property implies that \( T \) is a binary search tree on the \( x \)-coordinates of the points in \( P \), except that there is no relation between the \( x \)-coordinate of the point stored at \( u \) and that of any of its children. We use \( \text{MaxPST} \) to denote the priority search tree with the max-heap property on the \( y \)-coordinates. Similarly, \( \text{MinPST} \) denotes the priority search tree with the min-heap property on the \( y \)-coordinates. Because of heap property, both \( \text{MaxPST} \) and \( \text{MinPST} \) are balanced.

In order to use \( T \) as a binary search tree on the \( x \)-coordinates, McCreight stored at each internal node \( u \) one additional point \( p_u \) of \( P \), called *splitting point*, which is the point in the left subtree of \( u \) and having maximum \( x \)-coordinate value. Thus, both \( \text{MaxPST} \) and \( \text{MinPST} \) use \( O(n) \) space in addition to the space required for storing the coordinates of the members in \( P \). Since both \( \text{MaxPST} \) and \( \text{MinPST} \) are balanced trees, the following range queries can be answered efficiently:

1. **\text{LeftMostNE}(x_0, y_0)**: report the leftmost point of \( P \) in the north-east quadrant \([x_0, \infty) \times [y_0, \infty)\) of the query point \((x_0, y_0)\).

2. **\text{RightMostNW}(x_0, y_0)**: report the rightmost point of \( P \) in the north-west quadrant \((-\infty, x_0] \times [y_0, \infty)\) of the query point \((x_0, y_0)\).

3. **\text{HighestNE}(x_0, y_0)**: report the highest point of \( P \) in the north-east quadrant \([x_0, \infty) \times [y_0, \infty)\) of the query point \((x_0, y_0)\).

4. **\text{HighestNW}(x_0, y_0)**: report the highest point of \( P \) in the north-west quadrant \((-\infty, x_0] \times [y_0, \infty)\) of the query point \((x_0, y_0)\).

5. **\text{Highest3SidedUp}(x_0, x_1, y_0)**: report the highest point of \( P \) in the 3-sided query range \([x_0, x_1] \times [y_0, \infty)\).
6. **ENUMERATE3SIDEDUP**\((x_0, x_1, y_0)\): report all points of \(P\) in the 3-sided query range \([x_0, x_1] \times [y_0, \infty)\).

7. **LEFTMOSTSE**\((x_0, y_0)\): report the leftmost point of \(P\) in the south-east quadrant \([x_0, \infty) \times (-\infty, y_0]\) of the query point \((x_0, y_0)\).

8. **RIGHTMOSTSW**\((x_0, y_0)\): report the rightmost point of \(P\) in the south-west quadrant \((-\infty, x_0) \times (-\infty, y_0]\) of the query point \((x_0, y_0)\).

9. **LOWESTSE**\((x_0, y_0)\): report the lowest point of \(P\) in the south-east quadrant \([x_0, \infty) \times (-\infty, y_0]\) of the query point \((x_0, y_0)\).

10. **LOWESTSW**\((x_0, y_0)\): report the lowest point of \(P\) in the south-west quadrant \((-\infty, x_0) \times (-\infty, y_0]\) of the query point \((x_0, y_0)\).

11. **LOWEST3SIDEDDOWN**\((x_0, x_1, y_1)\): report the lowest point of \(P\) in the 3-sided query range \([x_0, x_1] \times (-\infty, y_1]\).

12. **ENUMERATE3SIDEDDOWN**\((x_0, x_1, y_1)\): report all points of \(P\) in the 3-sided query range \([x_0, x_1] \times (-\infty, y_1]\).

Using **MaxPST**, the queries 1-5 can be answered in \(O(\log n)\) time, whereas the query 6 takes \(O(\log n + m)\) time, where \(m\) is the number of points of \(P\) that are in the query range. Similarly, using **MinPST**, the queries 7-11 can be answered in \(O(\log n)\) time, whereas the query 12 takes \(O(\log n + m)\) time.

We define the **Min-Max Priority Search Tree** (**MinMaxPST**) for a set \(P\) of \(n\) points in \(\mathbb{R}^2\). It is a binary tree \(T\) with the following properties:

- For each internal node \(u\), all points in the left subtree of \(u\) have an \(x\)-coordinate which is less than the \(x\)-coordinate of any point in the right subtree of \(u\).

- The \(y\)-coordinate values of the nodes on even (resp. odd) levels are smaller (resp. greater) than the \(y\)-coordinate values of their descendants (if any), where the root is at level zero.

The first property implies that \(T\) is a binary search tree on the \(x\)-coordinates of the points in \(P\), except that there is no relation between the \(x\)-coordinates
of the points stored at $u$ and any of its children. The second property implies that $T$ is a min-max heap [Atkinson 1986] on the $y$-coordinates of the points in $P$.

We show that using the single structure $\text{MinMaxPST}$, we can answer all the above mentioned queries with the same time complexities.

In this chapter, we show that these results can also be obtained without storing the splitting point $p_u$ at each internal node $u$ of the tree. Thus, any node of the tree stores exactly one point of $P$, and the entire tree can be stored in the array containing the input points. Thus, we obtain an $in-place$ implementation of the priority search tree. We take for $T$ a binary tree of height $h = \lceil \log n \rceil$, such that the levels $0, 1, \ldots, h - 1$ are full and level $h$ consists of $n - (2^h - 1)$ nodes which are aligned as far as possible to the left. This allows us to store the tree, like in a standard heap, in an array $P[1, \ldots, n]$; the root is stored at $P[1]$, its left and right children in $P[2]$ and $P[3]$, etc. See Figure 3.1, where a set of points and the corresponding $\text{MinMaxPST}$ is shown.

In Sections 3.2 and 3.3, we will present algorithms for constructing the in-place $\text{MinMaxPST}$ and answering the above queries respectively. Each of these algorithms use, besides the array $P[1, \ldots, n]$, only $O(1)$ extra-space, in the sense that only a constant number of variables are used as the working space, and each one being an integer of $O(\log n)$ bits. The main result of this chapter is the following:
Theorem 3.1 Let $P$ be a set of $n$ points in $\mathbb{R}^2$.

1. The in-place MinMaxPST can be constructed in $O(n \log n)$ time using $O(1)$ extra-space.

2. Each of the queries, HighestNE, HighestNW, LowestSE, LowestSW, LeftMostNE, RightMostNW, LeftMostSE, RightMostSW, Highest3SidedUp and Lowest3SidedDown can be answered in $O(\log n)$ time using $O(1)$ extra-space.

3. The queries Enumerate3SidedUp and Enumerate3SidedDown can be answered in $O(\log n + m)$ time using $O(1)$ extra-space, where $m$ is the number of points in $P$ that lie in the query range.

In Section 3.4, as an application of the in-place priority search tree, we show how one can enumerate all the maximal empty axis-parallel rectangles among a set of $n$ points inside a rectangular region $R$ in $\mathbb{R}^2$ in $O(m \log n)$ time, where $m$ is the number of all maximal empty axis-parallel rectangles. Here we need to build an in-place MaxPST data structure on the set $P$ of points and use the procedures LeftMostNE, RightMostNW and Highest3SidedUp to achieve the desired time complexity for solving the problem using $O(1)$ extra-space. MaxPST can be constructed in an exactly similar manner as MinMaxPST.

For ease of presentation, we assume that no two points in the set $P$ have the same $x$ and $y$-coordinates. If they have the same $x$-coordinate, then the ties are broken by the $y$ coordinates. The $x$- and $y$-coordinates of a point $p$ in $\mathbb{R}^2$ will be denoted by $x(p)$ and $y(p)$, respectively. Most of the algorithms presented in this chapter have been implemented and the code is available at [Pratt 2011].

### 3.2 Constructing the in-place MinMaxPST

Let $P$ be a set of $n$ points in the plane. In this section, we present an algorithm for constructing the in-place MinMaxPST.
Let $h = \lceil \log n \rceil$ be the height of the MinMaxPST. Our algorithm constructs the tree level by level. After constructing the $(i - 1)$-th level of the tree, it maintains the following invariant:

- The sub-array $P[1, \ldots, 2^i - 1]$ stores levels $0, 1, \ldots, i - 1$ of the tree, and the points in the sub-array $P[2^i, \ldots, n]$ are stored in sorted order by their $x$-coordinates (see Figure 3.2(a)).

The algorithm starts by sorting the array $P[1, \ldots, n]$ by their $x$-coordinates. After this sorting step, the invariant holds with $i = 0$.

Consider the $i$-th iteration of the algorithm, where $i$ is an integer with $0 \leq i < h$. Let $A = n - (2^h - 1)$ be the number of nodes at level $h$ of the tree, and let $k = \lfloor A / 2^{h-i} \rfloor$. Level $i$ consists of $2^i$ nodes (see Figure 3.2(b)). If $k = 2^i$, then each of these nodes is the root of a subtree of size $2^{h+1-i} - 1$. Otherwise, we have $k < 2^i$, in which case level $i$ consists of, from left to right,

1. $k$ nodes, which are roots of subtrees, each of size $K_1 = 2^{h+1-i} - 1$,
2. one node, which is the root of a subtree of size $K_2 = 2^{h-i} - 1 + A - k2^{h-i}$,
3. $2^i - 1 - k$ nodes, which are roots of subtrees, each of size $K_3 = 2^{h-i} - 1$. 

Figure 3.2: Demonstration of the invariants of MinMaxPST
3.2. Constructing the in-place MinMaxPST

(a) In each block of the colored portions (non-PST part) of the array, the element having maximum (or minimum) y-coordinate are marked. The corresponding elements in the tree are marked by blue dots.

(b) The marked elements are moved in the appropriate position to get the MinMaxPST up to the $i$-th level.

(c) Shaded portion corresponds to MinMaxPST up to $i$-th level. The unshaded portion is sorted with respect to their x-coordinates.

Figure 3.3: Construction of $i$-th level of MinMaxPST
Algorithm 3.1: CONSTRUCTMINMAXPST

**Input:** An array \( P[1, \ldots, n] \) of points in \( \mathbb{R}^2 \).

**Output:** The min-max priority search tree of those points stored in \( P \).

1. \( h = \lfloor \log n \rfloor; \ A = n - (2^h - 1); \)
2. \( \text{level} = 0; \)
3. \( \text{HEAPSORT}(1, n); \)
4. \( \text{for} \ i = 0 \ \text{to} \ h - 1 \ \text{do} \)
5. \( \quad k = \lfloor A/2^{h-i} \rfloor; \)
6. \( \quad K_1 = 2^{h+1-i} - 1; \)
7. \( \quad K_2 = 2^{h-i} - 1 + A - k2^{h-i}; \)
8. \( \quad K_3 = 2^{h-i} - 1; \)
9. \( \quad \text{for} \ j = 1 \ \text{to} \ k \ \text{do} \)
10. \( \quad \ell = \text{index in } \{2^i + (j - 1)K_1, \ldots, 2^i + jK_1 - 1\} \text{ such that } y(P[\ell]) \text{ is maximum or minimum depending on the parity of level; } \)
11. \( \quad \text{swap } P[\ell] \text{ and } P[2^i + j - 1]; \)
12. \( \quad \text{if } k < 2^i \text{ then} \)
13. \( \quad \ell = \text{index in } \{2^i + kK_1, \ldots, 2^i + kK_1 + K_2 - 1\} \text{ such that } y(P[\ell]) \text{ is maximum or minimum depending on the parity of level; } \)
14. \( \quad \text{swap } P[\ell] \text{ and } P[2^i + k]; \)
15. \( \quad m = 2^i + kK_1 + K_2; \)
16. \( \quad \text{for} \ j = 1 \ \text{to} \ 2^i - k - 1 \ \text{do} \)
17. \( \quad \ell = \text{index in } \{m + (j - 1)K_3, \ldots, m + jK_3 - 1\} \text{ such that } y(P[\ell]) \text{ is maximum or minimum depending on the parity of level; } \)
18. \( \quad \text{swap } P[\ell] \text{ and } P[2^i + k + j]; \)
19. \( \text{HEAPSORT}(2^i+1, n); \)
20. \( \text{level} = \text{level} + 1; \)

See Figure 3.2(c) for an illustration. We divide the sub-array \( P[2^i, \ldots, n] \) into \( 2^i \) blocks: If \( k = 2^i \), then there are \( k \) blocks, each of size \( 2^{h+1-i} - 1 \). Otherwise, there are, from left to right, (i) \( k \) blocks of size \( K_1 \), (ii) one block of size \( K_2 \), and (iii) \( 2^i - 1 - k \) blocks of size \( K_3 \).

The algorithm scans the sub-array \( P[2^i, \ldots, n] \). In each of the \( 2^i \) blocks, it finds the highest or lowest point depending on whether they are in the even or odd level (Figure 3.3(a)). These highest or lowest points are swapped with the elements of the sub-array \( P[2^i, \ldots, 2^{i+1} - 1] \) (Figure 3.3(b)); note that these are the nodes of the \( i \)-th level of the MINMAXPST. At this moment, the invariant has not been restored yet, because the elements in the sub-array \( P[2^{i+1}, \ldots, n] \) may not be sorted by their \( x \)-coordinates. Therefore, the algorithm runs the heapsort algorithm on this sub-array. The final output
3.2. Constructing the in-place MinMaxPST

after $i$-th iteration is shown in Figure 3.3(c).

The complete algorithm for constructing the in-place min-max priority search tree is given in Algorithm 3.1. It uses algorithm \textsc{Heapsort}(m, n), which runs the heapsort algorithm on the sub-array $P[m, \ldots, n]$.

The correctness of this algorithm follows by observing that the invariant is correctly maintained. The initial sorting in line 3 takes $O(n \log n)$ time using $O(1)$ extra-space. Each of the $h = \lfloor \log n \rfloor$ iterations of the outer-most for-loop takes $O(n \log n)$ time and $O(1)$ extra-space. We can use one extra variable to maintain the value $2^i$, so that it does not have to be recomputed during the execution of the outer-most for-loop. Thus, the entire algorithm \textsc{ConstructMinMaxPST} takes $O(n \log^2 n)$ time and uses $O(1)$ extra-space.

3.2.1 Improving the construction time

We now show that the in-place MinMaxPST can be constructed in $O(n \log n)$ time. Consider again the $i$-th iteration of Algorithm 3.1. At the beginning of this iteration, (i.e., immediately after line 4), assign $s = P[2^{i+1}]$. In Figure 3.4(a) this is demonstrated, where the gray portion of the array corresponds to the MinMaxPST constructed up to the $(i-1)$-th level, and white portion is sorted according to the $x$-coordinates. As we have seen, immediately after line 18, level $i$ of the tree has been constructed, but the elements in $P[2^{i+1}, \ldots, n]$ may not be sorted by their $x$-coordinates. The reason is that the newly swapped elements in the sub-array $P[2^{i+1}, \ldots, n]$ have their $x$-coordinates smaller than $x(s)$. Note that the $x$-coordinates of all other elements in $P[2^{i+1}, \ldots, n]$ are greater than $x(s)$.

Assume that we have sorted the newly swapped elements by their $x$-coordinates. For any $j$ with $2^{i+1} \leq j \leq n$, define $f(j) = 0$ if $x(P[j]) < x(s)$, and $f(j) = 1$ if $x(P[j]) \geq x(s)$. Then sorting the sub-array $P[2^{i+1} \ldots n]$ by $x$-coordinates is equivalent to a stable sorting\footnote{Given a set of unsorted records having key values 0 and 1, where each record consists of some other information, a sorting of these records is called stable if it keeps the information of all the records with equal keys in the same relative order in the output as they were in the input.} of the bit-sequence $f(j)$, $2^{i+1} \leq j \leq n$. Katajainen and Pasanen [Katajainen 1992] have shown that this is possible in $O(n)$
time using $O(1)$ extra-space.

(a) Gray portion of the array corresponds to the MIN-MAXPST constructed so far and white portion is sorted according to the x-coordinate.

(b) Each red colored element is the point with the maximum (or minimum) $y$-coordinate value among elements in their respective block.

(c) Each green element corresponds to the first element in its block. Each red colored element is moved to the green position of its block by successive swapping.

(d) Green elements are the points with maximum (or minimum) $y$-value in their respective blocks. These are placed in the light-blue positions by swapping.

Figure 3.4: Improving the construction time of MINMAXPST - the $i$-th iteration

Thus, we modify lines 9-18 of the Algorithm 3.1 to sort the elements in $P[2^{i+1}, \ldots, n]$ as follows:

**Step 1:** In each of the $2^i$ blocks, find the highest (or lowest) point (see red colored elements in Figure 3.4(b)) and move it to the first position of its block.
by successive swapping. Note that all other points in this block remain sorted by their \(x\)-coordinates (see Figure 3.4(c)).

**Step 2:** For \(j = 1, 2, \ldots, 2^i\), swap \(P[2^i + j - 1]\) and the point at the first position of the \(j\)-th block (see Figure 3.4(d)).

**Step 3:** Run the heapsort algorithm on those elements that are the first elements in their block in the sub-array \(P[2^{i+1}, \ldots, n]\).

**Step 4:** Run Katajainen and Pasanen’s algorithm on the sub-array \(P[2^{i+1}, \ldots, n]\).

**Lemma 3.1** Given a set of points in \(\mathbb{R}^2\) in an array \(P\), the in-place construction of MINMAXPST in the same array \(P\) uses \(O(n \log n)\) time and \(O(1)\) extra-space.

**Proof:** Steps 1, 2, and 4 take \(O(n)\) time. In Step 3, the heapsort algorithm is run on a sequence of size at most \(2^i\), where the heap is stored in the first positions of those blocks that are involved in the sorting step (i.e., in the sub-array \(P[2^{i+1}, \ldots, n]\)). Thus, if the algorithm wants to access the \(m\)-th element in the heap, it computes, in \(O(1)\) time, the first position of the \(m\)-th block in the sub-array \(P[2^{i+1}, \ldots, n]\). It follows that Step 3 takes \(O(i \cdot 2^i)\) time.

The running time of the \(i\)-th iteration is \(O(n + i \cdot 2^i) = O(n + 2^i \log n)\), and therefore, the total time complexity of the algorithm is

\[
O \left( \sum_{i=0}^{h-1} (n + 2^i \log n) \right) = O(n \log n).
\]

The algorithm uses \(O(1)\) extra-space.

**Remark 3.1** The MAXPST and MINPST data structures can be constructed in a similar manner with the same time and space complexities, where we compute either the highest or the lowest point at all levels of the tree instead of alternating. It is to be noted that, even without space restriction, the best known running time to construct a priority search tree is also \(O(n \log n)\).
3.3 Queries on the in-place MINMAXPST

In this section, we present three query algorithms mentioned in Section 3.1. For ease of presentation, we describe the algorithms using the terminology of trees. We denote by $T$ the MINMAXPST that is implicitly stored in the array $P[1,\ldots,n]$, obtained by running the algorithm CONSTRUCTMINMAXPST. Recall that the root of $T$, denoted by $\text{root}(T)$, is stored at $P[1]$. Consider a node whose index in $P$ is $i$. If $2i \leq n$, then the left child of this node is stored at $P[2i]$. If $2i+1 \leq n$, then the right child of this node is stored at $P[2i+1]$. This node is a leaf if and only if $2i > n$. For any element $p \in P$, we denote by $T_p$ the subtree rooted at $p$. Furthermore, the left and right children of $p$ (if they exist) are denoted by $p_l$ and $p_r$, respectively.

3.3.1 LEFTMOSTNE($x_0, y_0$)

For two given real numbers $x_0$ and $y_0$, let $Q = [x_0, \infty) \times [y_0, \infty)$ be the north-east quadrant of the point $(x_0, y_0)$, and $P_Q$ be the set of points of $P$ that are in the region $Q$. If $P_Q \cap P \neq \emptyset$, define $p^*$ to be the leftmost point of $P_Q$. If $P_Q \cap P = \emptyset$, define $p^*$ to be the point $(\infty, \infty)$. Algorithm LEFTMOSTNE($x_0, y_0$) returns the point $p^*$.

As in the standard priority search tree [McCreight 1985], the search starts at the root of $T$. Since the $x$-coordinate of the partitioning line is not available in each node, two index variables $p$ and $q$ are required for navigation along the search path. Another variable $\text{best}$ is used to store the leftmost point in $Q$ obtained so far. At the end of the execution, $\text{best}$ will contain the final result. At any instant of time, the variables $\text{best}$, $p$, and $q$ satisfy the following invariant:

- If $P_Q \neq \emptyset$, then $p^* \in \{\text{best}\} \cup T_p \cup T_q$.
- If $P_Q = \emptyset$, then $p^* = \text{best}$.
- $p$ and $q$ are at the same level of $T$ and $x(p) \leq x(q)$.

The algorithm starts by initializing $\text{best} = (\infty, \infty)$ and $p = q = \text{root}(T)$. The algorithm uses a procedure UPDATELEFTMOST($t$), which takes an input
point \( t \in P \), and updates \( \text{best} \) as follows: if \( t \in Q \) and \( x(t) < x(\text{best}) \) then assign \( \text{best} = t \).

At each iteration, \( \text{UpdateLeftMost}(p) \) and \( \text{UpdateLeftMost}(q) \) are executed to update the variable \( \text{best} \); the variables \( p \) and \( q \) move down the tree according to the relative positions of their children and the query region \( Q \) as stated below. As soon as \( p \) reaches the leaf level, the algorithm terminates.

If \( p = q \), then irrespective of whether or not \( p \in Q \), we proceed along both left and right children of \( p \); thus, we assign \( q = p_r \) and \( p = p_l \). If \( p \) has only one child then we assign \( p = q = p_l \).

If \( p \neq q \), then we need to consider the following situations:

- **q is a leaf:** Here \( p \) may or may not be a leaf. If \( p \) is not a leaf, then the search proceeds to the children of \( p \).

- **q is not a leaf:** Here \( p \) is also not a leaf, and \( p \) has two children.

Now, if \( q \) has only one child then according to our inplace structure of the MinMaxPST, \( p, q \) lies in one level above the leaf level of \( T \). Here three situations need to be considered: (i) \( x(q_l) < x_0 \), (ii) \( x(p_r) < x_0 \) but \( x(q_l) > x_0 \), and (iii) \( x(p_r) > x_0 \). The assignments of \( p \) and \( q \) for executing the next iteration in these three situations are shown in Figures 3.5(a), 3.5(b) and 3.5(c) respectively. Note that here we need not have to consider the \( y \)-coordinate of \( p \) or \( q \) since they are in the last level, and if they are not inside \( Q \) then \( \text{best} \) will not be updated by the procedure \( \text{UpdateLeftMost} \).

If \( q \) has two children, we need to consider the \( y \)-coordinate of the children of both \( p \) and \( q \) to update the values of \( p \) and \( q \) for the next iteration.

Note that here \( x_0 \) (defining the vertical line \( L : x = x_0 \) of the region \( Q \)) may lie in one of the five intervals defined by \( I_0 = [-\infty, x(p_l)] \), \( I_1 = [x(p_l), x(p_r)] \), \( I_2 = [x(p_r), x(q_l)] \), \( I_3 = [x(q_l), x(q_r)] \), \( I_4 = [x(q_r), \infty] \).

- If \( x_0 \in I_0 \), then the \( x \)-coordinates of all the members in the set \( V = \{p_l, p_r, q_l, q_r\} \) are greater than \( x_0 \). In a MaxPST,
Algorithm 3.2: LEFTMOSTNE($x_0, y_0$)

Input: Real numbers $x_0$ and $y_0$ defining the north-east quadrant $Q$.

Output: The leftmost point $p^*$ in $Q \cap P$, if it exists; otherwise the point $(\infty, \infty)$.

1. $best = (\infty, \infty)$; $p = root(T)$; $q = root(T)$;
2. while $p$ is not a leaf do
3. $UpdateLeftMost(p)$; $UpdateLeftMost(q)$;
4. if $p = q$ then
5. if $p$ has one child then
6. $q = p_l$; $p = p_l$;
7. else
8. $q = p_r$; $p = p_l$;
9. else
10. if $q$ is leaf then
11. $q = p$;
12. else if $q$ has one child then
13. if $x(q_l) < x_0$ then
14. $p = q_l$; $q = q_l$;
15. else if $x(p_r) < x_0$ then
16. $p = p_r$; $q = q_r$;
17. else
18. $q = p_r$; $p = p_l$;
19. else
20. // $p$ and $q$ both have two children
21. $c_1 = p_l$; $c_2 = p_r$; $c_3 = q_l$; $c_4 = q_r$;
22. if $x_0 < x(c_1)$ then
23. $V = \{c_t | 1 \leq t \leq 4 \text{ and } c_t \text{ or one of its children is in } Q\}$;
24. if $V = \emptyset$ then
25. $q = c_1$;
26. else
27. $q = \text{leftmost point in } V$;
28. $p = q$;
29. else if $x_0 \geq x(c_4)$ then
30. $p = q = c_4$;
31. else
32. $i$ is the index such that $x(c_i) \leq x_0 < x(c_{i+1})$;
33. $p = c_i$;
34. $V = \{c_t | (i + 1) \leq t \leq 4 \text{ and } c_t \text{ or one of its children is in } Q\}$;
35. if $V = \emptyset$ then
36. $q = c_{i+1}$;
37. else
38. $q = \text{leftmost point in } V$;
39. $UpdateLeftMost(p)$; $UpdateLeftMost(q)$;
40. return $best$;
3.3. Queries on the in-place MinMaxPST

we consider the members in $V$ in increasing order of their $x$-coordinates. For each $\theta \in V$, we test whether $\theta \in Q$ (i.e., $y(\theta) > y_0$). If so, we set $p = q = \theta$ and do not consider the other members of $V$; otherwise, we consider the next member of $V$. Since we are working with a MinMaxPST, while considering $\theta \in \{p_l, p_r, q_l, q_r\}$ in this order, if either $\theta \in Q$ or $\theta \not\in Q$ but any one of its children lies in $Q$, then we set $p = q = \theta$ and do not check the other members in the list. If no such $\theta$ is found then the algorithm stops\(^2\).

- If $x_0 \in I_i$ for some $i = 1, 2, 3$, then we set $p$ to the left end point of $I_i$. We construct the ordered set $V$ with the right end point of $I_i$ and all the end-points of $I_j$, $i < j \leq 4$, and update $q$ with the leftmost element $\theta \in V$ as we did earlier (see Figure 3.5(d)). If no such $\theta$ is found then we set $q$ to the right-end of $I_i$.

\(^2\)In the pseudocode, we assigned $p = q = p_l$ (the right end-point of $I_0$). This will not cause any problem; here the algorithm continues up to the leaf level but best is not updated by the procedure $\text{UpdateLeftMost}$ any more.
• If $x_0 \in I_4$, then we assign $p = q = q_r$ since if there is any further update of best it will be any of the successors of $q_r$. Note that here we do not have to consult the $y$-coordinate of $q_r$ since if it or its successors are not inside $Q$ then best will not be updated by the procedure UPDATELEFTMOST.

The detailed steps of the algorithm are given in Algorithm 3.2. The following lemma states the correctness and the complexity result of this query algorithm.

**Lemma 3.2** Given an in-place MINMAXPST with a set of points in $\mathbb{R}^2$ in an array $P$, the proposed LEFTMOSTNE($x_0, y_0$) query algorithm correctly reports the resulting point in $O(\log n)$ time with $O(1)$ extra-space.

**Proof:** By a careful case analysis it can be observed that the invariant is correctly maintained. This implies the correctness of the algorithm. In each iteration of the while-loop, $p$ and $q$ move down the tree, except in line 12. In the latter case, however, $p$ will become a leaf in the next iteration. As a result, the while-loop makes $O(\log n)$ iterations. Since each iteration takes $O(1)$ time, the total time for algorithm LEFTMOSTNE is $O(\log n)$. It follows from the algorithm that it uses $O(1)$ extra-space. □

3.3.2 **HIGHEST3SIDEDUP($x_0, x_1, y_0$)**

Given three real numbers $x_0$, $x_1$ and $y_0$, the three-sided query range is defined by $Q = [x_0, x_1] \times [y_0, \infty)$. As in the earlier section, here also we will use $P_Q$ to denote the subset of points in $P$ that lie inside $Q$. If $P_Q \neq \emptyset$ then the highest point of $P$ in $Q$ is $p^*$ satisfying $y(p^*) = \max\{y(p)|p \in P_Q\}$. If $P_Q = \emptyset$, then define $p^* = (\infty, -\infty)$. Algorithm HIGHEST3SIDEDUP($x_0, x_1, y_0$), described in Algorithm 3.3, returns the point $p^*$.

As before, best will store the highest point in $Q$ found so far. Since $Q$ is bounded by two vertical sides, we use four index variables $p$, $p'$, $q$ and $q'$ to guide the search path. In addition, we use four bits $L$, $L'$, $R$ and $R'$; these correspond to the subtrees of $T$ rooted at the nodes $p$, $p'$, $q$, and $q'$, respectively; if a bit is equal to one, then the corresponding node is referred to as an active node (for example, if $L = 1$, then $p$ is an active node), and the
Algorithm 3.3: HIGHEST3SIDEDUP($x_0, x_1, y_0$)

**Input:** Real numbers $x_0$, $x_1$, and $y_0$ defining the region $Q = [x_0, x_1] \times [y_0, \infty)$.

**Output:** The highest point $p^*$ in $Q \cap P$, if it exists; otherwise the point $(\infty, -\infty)$.

1. $best = (\infty, -\infty)$;
2. if $x(root(T)) < x_0$ then
   3. $p = root(T); L = 1; L' = R = R' = 0$;
3. else if $x(root(T)) < x_1$ then
   4. $p' = root(T); L' = 1; L = R = R' = 0$;
   5. else
      6. $q = root(T); R = 1; L = L' = R' = 0$;
    8. while $L = 1 \lor L' = 1 \lor R = 1 \lor R' = 1$ do
      9. $Active = \{z \in \{L, L', R, R'\}| z = 1\}$;
     10. $Z^* = \{\tau(z) | z \in Active \& level(\tau(z)) \text{ is minimum}\}$;
     11. if $|Z^*| \geq 1$ then
        12. if $p' \in Z^*$ then
            13. CheckLeftIn($p'$);
        14. else if $q' \in Z^*$ then
            15. CheckRightIn($q'$);
        16. else if $p \in Z^*$ then
            17. CheckLeft($p$);
        18. else
            19. CheckRight($q$);
     20. return $best$;

Subtree rooted at that node may contain a candidate point for $p^*$. So the search is required to be performed in the subtree rooted at all active nodes. More formally, at any instant of time the variables satisfy the following invariants:

- If $L = 1$ then $x(p) < x_0$.
- If $L' = 1$ then $x_0 \leq x(p') \leq x_1$.
- If $R = 1$ then $x(q) > x_1$.
- If $R' = 1$ then $x_0 \leq x(q') \leq x_1$.
- If $L' = 1$ and $R' = 1$ then $x(p') \leq x(q')$. 
• If \( P_Q \neq \emptyset \), then either \( p^* = \text{best} \) or \( p^* \) is in the subtree rooted at any one of the active nodes. In other words, \( p^* \in \{\text{best}\} \cup (\cup_{z \in \text{Active}} \tau(z)) \), where

\[
\text{Active} = \{ z | z \in \{L, L', R, R' \} \text{ and } z = 1 \}
\]

and \( \tau(z) \) is defined as follows:

<table>
<thead>
<tr>
<th>( z )</th>
<th>( L )</th>
<th>( L' )</th>
<th>( R )</th>
<th>( R' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau(z) )</td>
<td>( p )</td>
<td>( p' )</td>
<td>( q )</td>
<td>( q' )</td>
</tr>
</tbody>
</table>

As mentioned earlier, if \( z \in \text{Active} \), then the corresponding variable \( \tau(z) \) is said to be active. If \( z \notin \text{Active} \), then \( \tau(z) \) is said to be inactive.

In the initialization, we set \( \text{best} = (\infty, -\infty) \). The other variables are assigned depending on the position of the root of \( T \) with respect to the query region \( Q \). More specifically,

- if \( x(\text{root}(T)) < x_0 \) then initialize \( L = 1, L' = R = R' = 0 \) and \( p = \text{root}(T) \).
- if \( x_0 < x(\text{root}(T)) \leq x_1 \) then initialize \( L' = 1, L = R = R' = 0 \) and \( p' = \text{root}(T) \).
- if \( x(\text{root}(T)) > x_1 \) then initialize \( R = 1, L = L' = R' = 0 \) and \( q = \text{root}(T) \).

We use \( \text{level}(t) \) to denote the level of node \( t \in T \). The \( \text{root}(T) \) is assumed to have level 0. At each iteration, we choose an active node having minimum level. If more than one such node is available, then we choose one of them arbitrarily. We define a variable \( \lambda = \min_{z \in \text{Active}} \text{level}(\tau(z)) \) and \( Z^* = \{ \tau(z) | z \in \text{Active}, \text{level}(\tau(z)) = \lambda \} \). In other words, \( Z^* \) is the set of active nodes having minimum level \( \lambda \). If \( |Z^*| = 0 \) (i.e., no active node is available) then the algorithm stops. Otherwise, if \( p' \) (resp. \( q' \)) lies in \( Z^* \), we set \( \text{dominating\_variable} = p' \) (resp. \( q' \)). Tie is broken arbitrarily. If \( p', q' \notin Z^* \) and \( |Z^*| > 0 \) then we assign \( p \) or \( q \) to \( \text{dominating\_variable} \) depending on which one is in \( Z^* \). The tie is resolved arbitrarily. Next, we call any one of the procedures \( \text{CheckLeft} \), \( \text{CheckLeftIn} \), \( \text{CheckRight} \) and \( \text{CheckRightIn} \), depending on whether \( \text{dominating\_variable} \) is equal to \( p \),
3.3. Queries on the in-place MinMaxPST

$p', q$, or $q'$, respectively. These procedures are used to decide the possible
directions of the search paths for the next iteration. We describe the details
of the procedures CHECKLEFT and CHECKLEFTIN in Sections 3.3.2.1 and
3.3.2.2. The other two procedures are symmetric to these.

Each of these procedures evokes UPDATEHIGHEST and EXUPDATEHIGHEST.
UPDATEHIGHEST($t$) updates best as follows: if $t \in Q$ and $y(t) > y(best)$ then
$best = t$ is set. For a node $t$, EXUPDATEHIGHEST($t$) is executed only when it
is certain that all the elements in the subtree rooted at $t$ are within the two
vertical lines at $x_0$ and $x_1$. The procedure EXUPDATEHIGHEST($t$) also updates
$best$ by checking $t$ and its children (if any one of them exists); in other words, it
evokes UPDATEHIGHEST($t$), UPDATEHIGHEST($t_l$) and UPDATEHIGHEST($t_r$).

3.3.2.1 CHECKLEFT

This procedure is called with the node $p$ as argument, provided it is active,
i.e., $L = 1$ and level($p$) is minimum among other active nodes. Depending on
the value of the $x$-coordinates of the two children of $p$, the following situations
may arise:

$x(p_l), x(p_r) < x_0$: See Figure 3.6(a). Here, we set $p = p_r$.

$x(p_l) < x_0, x_0 < x(p_r) < x_1$: See Figure 3.6(b). Here, we evoke UPDATEHIGHEST($p_r$),
and set $p' = p_r$ and $p = p_l$ copying the existing value of $p'$ in a tem-
porary location $temp$. Now, if $L' = 1$ and $R' = 1$, then we evoke
EXUPDATEHIGHEST($p'$); else if $L' = 1$ and $R' = 0$, then we set $q' =
temp$ and $R' = 1$.

$x(p_l) < x_0, x(p_r) > x_1$: See Figure 3.6(c). We set $q = p_r$, $p = p_l$ and $R = 1$.
Note that the existing values of $L'$ and $R'$ are both zero.

$x_0 < x(p_l) < x(p_r) < x_1$: See Figure 3.6(d). Here we set $L = 0$ and $p' = p_l$
copying the existing value of $p'$ in $temp$. Here, depending on the value
of $L'$ and $R'$, four situations may happen:

- $L' = 0, R' = 0$: We evoke UPDATEHIGHEST($p_l$) and UPDATEHIGHEST($p_r$),
  and set $q' = p_r$ and $L' = R' = 1$. 


Algorithm 3.4: CHECKLEFT(p)

Input: A node p such that \(x(p) < x_0\).

1. if \(p\) is a leaf then
   1.1. \(L = 0\)

2. else if \(p\) has one child then
   2.1. /* the only child is \(p_1\) */
   2.2. if \(x(p) \leq x_0\) then
   2.3. \(L' = 0\)
   2.4. else if \(p\) has one child
   2.5. /* the only child is \(p_1\) */
   2.6. if \(L' = 1 \land R' = 1\) then
   2.7. \(q' = p'; \quad R' = 1;\)
   2.8. else if \(L' = 1\) then
   2.9. \(p' = p_1; \quad L' = 1; \quad L = 0;\)
   2.10. else if \(x(p_1) < x_0\) then
   2.11. \(p = p_1;\)
   2.12. else
   2.13. \(q = p_1; \quad R = 1; \quad L = 0;\)

3. else /* \(p\) has two children */
4. if \(x(p) < x_0\) then
5. if \(x(p) < x_0\) then
6. \(p = p_1;\)
7. else if \(x(p) \leq x_1\) then
8. \(UPDATEHIGHEST(p_1);\)
9. if \(L' = 1 \land R' = 1\) then
10. \(EXUPDATEHIGHEST(p');\)
11. else if \(L' = 1\) then
12. \(p' = p_1; \quad L' = 1;\)
13. else if \(x(p_1) < x_0\) then
14. \(q = p_1; \quad p = p_1;\)
15. else
16. \(q = p_1; \quad R = 1; \quad L = 0;\)

17. \(L = 0; R = 1;\)

18. else
19. \(UPDATEHIGHEST(p_1);\)
20. if \(x(p) > x_1\) then
21. if \(x(p) > x_1\) then
22. \(q = p_1; \quad p' = p_1; \quad L = 0; \quad L' = R = 1;\)
23. else
24. \(UPDATEHIGHEST(p_1);\)
25. if \(R' = 1 \land L' = 1\) then
26. \(EXUPDATEHIGHEST(p_1); EXUPDATEHIGHEST(q');\)
27. if \(x(p) > x_1\) then
28. \(UPDATEHIGHEST(p_1);\)
29. if \(R' = 1 \land L' = 1\) then
30. \(EXUPDATEHIGHEST(p_1); \quad L' = 1;\)
31. else
32. \(UPDATEHIGHEST(p_1); \quad L' = 1;\)
33. else
34. \(EXUPDATEHIGHEST(p_1); \quad L' = 1;\)
35. else
36. \(EXUPDATEHIGHEST(p_1); \quad L = 0;\)
37. else
38. \(q = p_1; \quad L = 0; \quad R = 1;\)
3.3. Queries on the in-place MinMaxPST

Figure 3.6: Different cases for the procedure CheckLeft(p).

- $L' = 0, R' = 1$: We evoke UpdateHighest($p_l$) and ExUpdateHighest($p_r$), and set $L' = 1$.
- $L' = 1, R' = 0$: We evoke UpdateHighest($p_l$) and ExUpdateHighest($p_r$), and set $q' = temp$ and $R' = 1$.
- $L' = 1, R' = 1$: We evoke UpdateHighest($p_l$), ExUpdateHighest($p_r$), ExUpdateHighest($temp_r$) and ExUpdateHighest($temp_l$).

$x_0 < x(p_l) < x_1, x(p_r) > x_1$: See Figure 3.6(e). We evoke UpdateHighest($p_l$), and set $p' = p_l$, $q = p_r$, $L = 0$, $L' = 1$, and $R = 1$. Note that the existing value of $R' = 0$ will remain unchanged in this situation.

$x(p_l), x(p_r) > x_1$: See Figure 3.6(f). We set $q = p_l$, $L = 0$ and $R = 1$. Note that the existing values of $L'$ and $R'$ are both zero.
Observe that in each case the variables satisfy all the invariants. The detailed steps of the algorithm are given in Algorithm 3.4, where the special cases, when \( p \) is a leaf or \( p \) has only one child, are taken care of appropriately.

### 3.3.2.2 \textbf{CheckLeftIn}

This procedure is called with the node \( p' \) as argument, provided it is active, i.e., \( L' = 1 \) and \( \text{level}(p') \) is minimum among other active nodes. If we work on \( \text{MaxPST} \), then in this situation nothing needs to be done. Since we are working on \( \text{MinMaxPST} \), we may have \( p' \notin Q \) (i.e., \( x_0 \leq x(p') \leq x_1 \) and \( y(p') < y_0 \)), but the children of \( p' \) may lie in \( Q \). This is handled by setting the variables as follows for the next iteration.

![Different cases for the procedure CheckLeftIn(p').](image)

\( x(p'_l), x(p'_r) < x_0 \): We set \( p = p'_r, L = 1 \) and \( L' = 0 \) (see Figure 3.7(a)).
3.3. Queries on the in-place MinMaxPST

\[ x(p'_{l}) < x_{0}, \ x_{0} < x(p'_{r}) < x_{1}: \text{We evoke UpdateHighest}(p'_{r}), \text{and set } p = p'_{l}, \]
\[ p' = p'_{r}, \text{and } L = 1 \text{(see Figure 3.7(b))}. \]

\[ x(p'_{l}) < x_{0}, \ x(p'_{r}) > x_{1}: \text{We set } p = p'_{l}, q = p'_{r}, L = 1, L' = 0 \text{ and } R = 1. \text{ Note that the existing value of } R' \text{ is zero (see Figure 3.7(c))}. \]

\[ x_{0} < x(p'_{l}) < x(p'_{r}) < x_{1}: \text{We evoke UpdateHighest}(p'_{l}), \text{and set } p' = p'_{l}. \text{ If } R' \neq 1, \text{we evoke UpdateHighest}(p'_{r}), \text{and set } q' = p'_{r}, R' = 1. \text{ If } R' = 1, \text{then we execute ExUpdateHighest}(p'_{r}) \text{(see Figure 3.7(d))}. \]

\[ x_{0} < x(p'_{l}) < x_{1}, x(p'_{r}) > x_{1}: \text{We evoke UpdateHighest}(p'_{l}), \text{and set } q = p'_{r}, \]
\[ p' = p'_{l}, \text{and } R = 1 \text{(see Figure 3.7(e))}. \]

\[ x(p'_{l}), x(p'_{r}) > x_{1}: \text{We set } q = p'_{l}, L' = 0 \text{ and } R = 1. \text{ Note that the existing value of } R' \text{ is zero (see Figure 3.7(f))}. \]

We describe the pseudocode for CHECKLEFTIN\((p')\) in Algorithm 3.5.

Lemma 3.3 After an iteration,

(i) an active variable \(u\) may become inactive only when \(u\) is the dominating variable in that iteration.

(ii) if an inactive variable \(u\) becomes active, then level(\(u\)) > \(\lambda\), where \(\lambda\) is the level of the dominating variable in that iteration.

Proof: (i) Follows from the procedures CHECKLEFT, CHECKLEFTIN, CHECKRIGHT and CHECKRIGHTIN. Consider the procedure CHECKLEFT where \(p\) is the dominating variable. Observe that during the execution of this procedure only \(L\) is set to zero under certain conditions and no other bits, namely \(L', R, R'\) become zero. Thus only \(p\) can be set inactive. Similar thing takes place with \(p'_{l}, q\) and \(q'\) in the procedures CHECKLEFTIN, CHECKRIGHT and CHECKRIGHTIN respectively.

(ii) We explain this result with the procedures CHECKLEFT and CHECKLEFTIN. In the procedure CHECKLEFTIN, if an inactive variable \(u\) is made active, it is assigned a child of the dominating variable \(p'_{l}\); thus the level of \(u\) becomes \(\lambda + 1\). But the same is not always true in the procedure CHECKLEFT. Here,
Algorithm 3.5: CheckLeftIn\((p')\)

Input: A node \(p'\) such that \(x_0 \leq x(p') \leq x_1\).

1. if \(p'\) is a leaf then
   
   2. \(L' = 0\)

3. else if \(p'\) has one child then
   
   4. /* assume that the only child is \(p_l\) */
   
   5. if \(x_0 \leq x(p'_l) \leq x_1\) then
      
      6. \(p' = p'_l;\)
      
   7. else if \(x(p'_l) < x_0\) then
      
      8. \(p = p'_l; L' = 0; L = 1;\)
      
   9. else
      
   10. \(q = p'_l; R = 1; L' = 0;\)

11. else
   
   12. // \(p'\) has two children
   
   13. if \(x(p'_l) < x_0\) then
      
      14. if \(x(p'_r) < x_0\) then
         
         15. \(p = p'_l; L = 1; L' = 0;\)
         
      17. else if \(x(p'_r) \leq x_1\) then
         
         18. \(p = p'_l; p' = p'_r; L = 1;\)
         
      20. else
         
         21. \(q = p'_r; p = p'_l; R = 1; L = 1; L' = 0;\)
         
   22. else if \(x(p'_l) \leq x_1\) then
      
      23. if \(x(p'_r) > x_1\) then
         
         24. \(q = p'_r; p' = p'_l; R = 1;\)
         
      26. else
         
         27. \(p' = p'_l;\)
         
      28. if \(R' = 1\) then
         
         29. \(p'_r = p'_l;\)
         
      30. else
         
         31. \(q' = p'_l; p' = p'_r; R' = 1;\)
         
      32. else
         
         33. \(q = p'_l; L' = 0; R = 1;\)
   
   34. else
      
   35. \(q = p'_l; L' = 0; R = 1;\)
3.3. Queries on the in-place \textbf{MinMaxPST} \hfill 51

\(p\) is the \textit{dominating variable} and we assign \(q' = p'\) in line numbers 10, 26 and 39 of \textsc{CheckLeft}. It also needs to be mentioned that in the algorithm \textsc{Highest3SidedUp}, when both \(p, p'\) are \textit{active} and \(p\) is chosen as the \textit{dominating variable}, then \(\text{level}(p') > \text{level}(p)\). Thus, here also we have \(\text{level}(q') > \lambda\).

\begin{lemma}
\begin{enumerate}[(i)]
\item In each iteration of algorithm \textsc{Highest3SidedUp}, the value of \(\lambda\) (i.e., the minimum level among active nodes) increases or remains same.
\item A variable \(u \in \{p, p', q, q'\}\) can be a dominating variable with the level \(\theta\) \((0 \leq \theta \leq \log n)\) in at most one iteration.
\item There may exist at most four consecutive iterations during which the value of \(\lambda\) remains same.
\end{enumerate}
\end{lemma}

\textbf{Proof:} (i) In each iteration, we choose an \textit{active} node \(u \in \{p, p', q, q'\}\) having minimum level \(\lambda\). After the execution of the corresponding procedure, either \(u\) does not remain \textit{active} or it is assigned to a child of \(u\), which is in the next level. Now, the minimum level among \textit{active} nodes either increases or remains the same depending on whether there exists one or more node having level \(\lambda\).

(ii) Consider an iteration where \(u\) is chosen as the \textit{dominating variable} with \(\text{level}(u) = \theta\) for the first time. After this iteration, if \(u\) still remains active then \(\text{level}(u)\) becomes \(\theta + 1\). If \(u\) does not remain \textit{active} (see Lemma 3.3(i)), and after a few iteration, it comes back as \textit{active}, then its level becomes at least \(\theta + 1\) (by the part (i) of this lemma, and Lemma 3.3(ii)).

(iii) Follows from part (ii) of this lemma and the fact that the number of \textit{active} variables is at most 4. \(\square\)

\begin{lemma}
Given an in-place \textbf{MinMaxPST} with a set of points in \(\mathbb{R}^2\) in an array \(P\), algorithm \textsc{Highest3SidedUp}(\(x_0, x_1, y_0\)) correctly reports the point having maximum \(y\)-coordinate in the query region, in \(O(\log n)\) time with \(O(1)\) extra-space.
\end{lemma}

\textbf{Proof:} The space complexity trivially follows since we have used \(O(1)\) working variables (4 index variables \(p, p', q, q'\), 4 bit variables \(L, L', R, R'\), 4 integer
locations for storing levels of the variables \( p, p', q, q' \), and a scalar location \( \lambda \)). The time complexity follows from Lemma 3.4(iii) and the fact that the height of \( \text{MinMaxPST} \) is \( O(\log n) \).

### 3.3.3 Enumerate3SidedUp\( (x_0, x_1, y_0) \)

Here the query region \( Q \) is same as that in Highest3SidedUp, and the objective is to report all the points of \( P \) in the region \( Q \). This algorithm uses the same approach as algorithm Highest3SidedUp, and is presented in Algorithm 3.6. It uses the procedures EnumerateLeft, EnumerateLeftIn, EnumerateRight and EnumerateRightIn which are similar to CheckLeft, CheckLeftIn, CheckRight and CheckRightIn, respectively. The only difference is that all these procedures call the procedure Explore\( (t, y_0, \text{level}(t)) \) instead of ExUpdateHighest\( (t) \) and reporting of \( t \) is done instead of UpdateHighest\( (t) \).

**Algorithm 3.6: Enumerate3SidedUp\( (x_0, x_1, y_0) \)**

```plaintext
Input: Real numbers \( x_0, x_1 \), and \( y_0 \) defining the region
\[ Q = [x_0, x_1] \times [y_0, \infty) \].
Output: All elements of \( Q \cap P \).

1. \textbf{if} \( x(\text{root}(T)) < x_0 \) \textbf{then}
2. \hspace{1em} \[ p = \text{root}(T); \ L = 1; \ L' = R = R' = 0 \];
3. \textbf{else if} \( x(\text{root}(T)) < x_1 \) \textbf{then}
4. \hspace{1em} \[ p' = \text{root}(T); \ L' = 1; \ L = R = R' = 0 \];
5. \textbf{else}
6. \hspace{1em} \[ q = \text{root}(T); \ R = 1; \ L = L' = R' = 0 \];
7. \textbf{while} \( L = 1 \lor L' = 1 \lor R = 1 \lor R' = 1 \) \textbf{do}
8. \hspace{1em} \[ \mathcal{I} = \{ z \in \{L, L', R, R'\} | z = 1 \}; \]
9. \hspace{1em} \[ z = \text{element of } \mathcal{I} \text{ for which } \text{level}(N(z)) \text{ is minimum}; \]
10. \hspace{1em} \textbf{if} \( z = L \) \textbf{then}
11. \hspace{2em} \[ \text{EnumerateLeft}(p); \]
12. \hspace{1em} \textbf{else if} \( z = L' \) \textbf{then}
13. \hspace{2em} \[ \text{EnumerateLeftIn}(p'); \]
14. \hspace{1em} \textbf{else if} \( z = R \) \textbf{then}
15. \hspace{2em} \[ \text{EnumerateRight}(q); \]
16. \hspace{1em} \textbf{else}
17. \hspace{2em} \[ \text{EnumerateRightIn}(q'); \]
```
The algorithm \( \text{EXPLORE}(t, y_0, \text{level}(t)) \) reports all points in \( T_t \) whose \( y \)-coordinates are at least \( y_0 \). An in-order traversal is performed in \( T_t \). At each node, the corresponding point is reported if it lies in \( Q \). If two consecutive nodes on a path do not lie in \( Q \), then the traversal along that path does not progress further. A variable \( \text{level} \) is used during the traversal that indicates the level of the current node. This is needed since we are using MinMaxPST. Thus observing whether the current node is at even or odd level becomes important. The input parameter \( \text{level}(t) \) is used to initialize the variable \( \text{level} \). The procedure \( \text{EXPLORE} \) is described in Algorithm 3.7. It uses two variables \( \text{current} \) and \( \text{state} \) satisfying the following invariant:

- \( \text{current} \) is a node in \( T_p \).
- \( y(\text{current}) \geq y_0 \).
- \( \text{state} \in \{0, 1, 2\} \).
- If \( \text{state} = 1 \), then all elements of the set \( Q \cap (\{\text{current}\} \cup T_{\text{current}[l]} \) have been reported, where \( T_{\text{current}[l]} \) is the left child of \( \text{current} \) in the tree \( T \).
- If \( \text{state} = 2 \), then all elements of the set \( Q \cap T_{\text{current}} \) have been reported.

As in Section 3.3.2, it can be shown that the number of iterations of the while-loop of algorithm \( \text{ENUMERATE3SIDEDUP} \) is at most four times the height of \( T \). Thus, we have the following result.

**Lemma 3.6** Given an in-place MinMaxPST in an array \( P \) and a query range \( Q = [x_0, x_1] \times [y_0, \infty) \), the algorithm \( \text{ENUMERATE3SIDEDUP}(x_0, x_1, y_0) \) correctly computes all elements of \( P \) in the region \( Q \) in \( O(\log n + |P_Q|) \) time using \( O(1) \) extra-space, where \( P_Q \) is the set of points in the region \( Q \).

**Remark 3.2** The algorithms LeftMostSE, RightMostNW and RightMostSW are similar to algorithm LeftMostNE. Similarly, Lowest3SidedDown and Enumerate3SidedDown are analogous to Highest3SidedUp and Enumerate3SidedUp, respectively. The algorithms HighestNE, HighestNW can be formulated using Highest3SidedUp. Similarly, LowestSE and LowestSW can be formulated using Lowest3SidedDown.
Algorithm 3.7: EXPLORE(p, y₀, level)

Input: A node p and its level in T and a real number y₀.
Output: All points q in T_p for which y(q) ≥ y₀.

1. current = p;
2. state = 0;
3. while current ≠ p or state ≠ 2 do
   if state = 0 then
      if y(current) ≥ y₀ then
         report current;
      if (current has a left child) \ (y(current_l) ≥ y₀ ∨ (y(current) ≥ y₀ ∧ level mod 2 = 0)) then
         current = current_l; level = level + 1;
      else
         state = 1;
   else
      if state = 1 then
         if (current has a right child) \ (y(current_r) ≥ y₀ ∨ (y(current) ≥ y₀ ∧ level mod 2 = 0)) then
            current = current_r; level = level + 1; state = 0;
         else
            state = 2;
      else
         // state = 2 and current ≠ p
         if current is the left child of its parent then
            state = 1;
         current = parent(current); level = level - 1;

3.4 Enumerating all axis-parallel maximal empty rectangles

Let R = [x_min, x_max] × [y_min, y_max] be an axis-parallel rectangular region containing a set P of n points. An empty rectangle is a rectangle inside R that contains no point of P inside it. A maximal empty rectangle (MER) is an empty rectangle that is not properly contained in any other empty rectangle. Each edge of an MER either contains a point in P or coincides with the boundary of R.

Namaad et al [Naamad 1984] proposed an algorithm for this problem which
runs in $O(\min(n^2, m \log n))$ time, where $m$ is the number of all possible MERs. They showed that in the worst case $m$ may be $O(n^2)$; but if the points are randomly placed, then the expected value of $m$ is $O(n \log n)$.

Atallah and Frederickson [Atallah 1986] and Orłowski [Orłowski 1990] proposed simple algorithms for enumerating all the MERs that run in $O(m + n \log^2 n)$ and $O(m + n \log n)$ time respectively. If the objective is only to identify the largest MER, then efficient algorithms are available. Chazelle et al. [Chazelle 1986a] proposed an $O(n \log^3 n)$ time algorithm for this problem. Later Aggarwal and Suri [Aggarwal 1987] improved the time complexity to $O(n \log^2 n)$. The same time complexity result holds for the recognition of the largest MER among a set of arbitrary polygonal obstacles [Nandy 1994]. Boland and Urrutia [Boland 2001] gave an $O(n \log n)$ time algorithm for finding the largest MER inside an $n$-sided simple polygon. None of these algorithms is in-place; each of them uses $O(n)$ work-space apart from the input array.

We will consider the problem of designing an in-place algorithm for enumerating all the axis-parallel MERs using $O(1)$ extra-space. It is to be noted that we can use this algorithm for optimizing any measure among all the axis-parallel MERs.

### 3.4.1 Algorithm

We broadly divide axis-parallel MERs into the following three categories:

**Type-A**: The axis-parallel MERs whose top and bottom boundaries are aligned with the top and bottom boundaries of $R$ respectively.

**Type-B**: The axis-parallel MERs whose top boundary is aligned with the top boundary of $R$, but bottom boundary passes through a point in $P$.

**Type-C**: The axis-parallel MERs whose top boundary passes through a point in $P$. Its bottom boundary may or may not pass through a point in $P$.

Clearly, each axis-parallel MER belongs to exactly one of the above three types. Now, we will show that the MaxPST defined in Section 3.1, can be
used for solving our problem in an in-place manner using only $O(1)$ extra-space.

**Algorithm 3.8: ENUMERATEMER**

**Input:** An axis-parallel rectangular region $R = [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$ containing a set $P$ of $n$ points.

**Output:** All axis-parallel maximal empty rectangles inside $R$.

1. `ENUMERATETYPEA(R, P);`
2. `CONSTRUCTMAXPST(P);`
3. `ENUMERATETYPEB(R, P);`
4. `ENUMERATETYPEC(R, P);`

Algorithm `ENUMERATEMER`, given in Algorithm 3.8, enumerates all axis-parallel MERs inside $R$ among the set $P$ of points. It calls the procedures `ENUMERATETYPEA`, `ENUMERATETYPEB` and `ENUMERATETYPEC` to enumerate all Type-A, Type-B and Type-C axis-parallel MERs respectively. The `ENUMERATETYPEB` and `ENUMERATETYPEC` procedures need the MaxPST data structure. So, before executing these two procedures, we need to invoke the `CONSTRUCTMAXPST` procedure for building MaxPST. We describe the three procedures separately in the following subsections.

### 3.4.1.1 ENUMERATETYPEA($R, P$)

**Algorithm 3.9: ENUMERATETYPEA($R, P$)**

**Input:** An axis-parallel rectangular region $R = [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$ containing a set $P$ of $n$ points.

**Output:** All the Type-A MERs inside $R$

1. `HEAPSORT(P);`
2. `top = y_{\text{max}}; bottom = y_{\text{min}}; left = x_{\text{min}};`
3. for $i = 1$ to $n$ do
4.     `right = x(P[i]);`
5.     `report rectangle (left, right, top, bottom) ;`
6.     `left = right;`
7. `right = x_{\text{max}};`
8. `report rectangle (left, right, top, bottom) ;`

Type-A axis-parallel MERs are easy to obtain. First sort the points in $P$ in increasing order of their $x$-coordinate values in an in-place manner. Now
3.4. Enumerating all axis-parallel maximal empty rectangles

Each pair of consecutive points define a Type-$A$ axis-parallel MER. Algorithm EnumerateTypeA, given in Algorithm 3.9, computes all the Type-$A$ axis-parallel MERs.

### 3.4.1.2 EnumerateTypeB($R, P$)

Algorithm 3.10: EnumerateTypeB($R, P$)

| Input: An axis-parallel rectangular region $R = [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$ and the MaxPST of the set $P$ of $n$ points. |
| Output: All the Type-$B$ axis-parallel MERs inside $R$. |

for $i = 1$ to $n$ do

1. $top = y_{\text{max}}$;
2. $bottom = y(P[i])$;
3. $l = \text{RightMostNW}(x(P[i]), y(P[i]))$;
4. if $l = (\infty, -\infty)$ then
5. \hspace{1em} $left = x_{\text{min}}$;
6. else
7. \hspace{1em} $left = x(l)$;
8. $r = \text{LeftMostNE}(x(P[i]), y(P[i]))$;
9. if $r = (\infty, -\infty)$ then
10. \hspace{1em} $right = x_{\text{max}}$;
11. else
12. \hspace{1em} $right = x(r)$;
13. report rectangle $(left, right, top, bottom)$;

A Type-$B$ axis-parallel MER is an axis-parallel MER whose top boundary coincides with the top boundary of $R$ but bottom boundary passes through a point in $P$. The number of Type-$B$ axis-parallel MERs is at most $n$. The reason is as follows: Let the bottom boundary of a Type-$B$ axis-parallel MER pass through a point $p \in P$. Its left and right boundaries are uniquely defined by two points RightMostNW($x(p), y(p)$) and LeftMostNE($x(p), y(p)$) respectively. The algorithm EnumerateTypeB is stated in Algorithm 3.10.

For a given point $p \in P$, it uses MaxPST for getting the points RightMostNW($x(p), y(p)$) and LeftMostNE($x(p), y(p)$) in $O(\log n)$ time using $O(1)$ extra-space (see Section 3.3.1). Thus, the time required for computing all the Type-$B$ axis-parallel MERs present on $R$ is $O(n \log n)$. 

3.4.1.3 EnumerateTypeC(R, P)

Let \( p^* \in P \) be at the root of the MaxPST. We now describe the method of computing all the Type-C axis-parallel MERs with the point \( p^* \in P \) on its top boundary. Note that while computing the Type-C axis-parallel MERs with top boundary passing through the point \( p^* \), all the points \( \{q|y(q) \geq y(p^*)\} \) can be ignored. So, we first delete the point \( p^* \) from MaxPST.

Let us now consider a drop-down curtain whose top is fixed at \( p^* \), and its left and right sides are initialized by \( x_{\text{min}} \) and \( x_{\text{max}} \) respectively. Imagine lowering the curtain till either it reaches a point of \( P \) or reaches the bottom boundary of \( R \). If it reaches a point of \( p \in P \), then \( p \) satisfies the following:

(i) \( \text{left} < x(p) < \text{right} \),

(ii) \( y(p) < y(p^*) \),

(iii) \( y(p) \) is maximum among all the points in \( P \) satisfying (i) and (ii).

Note that if MaxPST contains all the points \( \{q|y(q) < y(p^*)\} \), then \( p \) can be computed by performing the \text{HIGHEST3SIDEDUP}(\text{left}, \text{right}, y_{\text{min}}) \) query in MaxPST. If \( p \neq \text{NIL} \), then we report a Type-C rectangle, and adjust left or right as follows: if \( x(p) < x(p^*) \) then \( \text{left} = x(p) \), otherwise \( \text{right} = x(p) \).

We further pull the curtain down till it meets a point of \( P \) or the bottom boundary of \( R \), and report the next Type-C axis-parallel MER. By repeating
Algorithm 3.11: ENUMERATETYPEC(R,P)

Input: An axis-parallel rectangular region $R = [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$ and the MaxPST of the set $P$ of $n$ points.

Output: All the Type-C axis-parallel MERs inside $R$.

1. $i = 0$;
2. while $i \neq n$ do
3. $p^* = \text{DELETETOP}(P)$;
4. $top = y(p^*); left = x_{\text{min}}; right = x_{\text{max}}$;
5. repeat
6. $h = \text{HIGHEST3SIDEDUP}(left, right, y_{\text{min}})$;
7. if $h = (\infty, -\infty)$ then
8. $bottom = y_{\text{min}}$;
9. else
10. $bottom = y(h)$;
11. report rectangle $(left, right, top, bottom)$;
12. if $h \neq (\infty, -\infty)$ then
13. if $x(h) < x(p^*)$ then
14. $left = x(h)$;
15. else
16. $right = x(h)$;
17. until $h = (\infty, -\infty)$;
18. $i = i + 1$;

the above steps, we report all Type-C axis-parallel MERs whose top boundary is anchored at $p^*$ (see Figure 3.8).

The iteration continues until MaxPST becomes empty. In Subsection 3.4.1.4, we show that deletion of the root from a MaxPST can be performed in $O(\log n)$ time with $O(1)$ extra storage, where $n$ is the number of points in the MaxPST. In Subsection 3.4.1.5, we mention the changes required in the algorithm HIGHEST3SIDEDUP to handle the dynamically changing MaxPST.

If $m'$ be the number of axis-parallel MERs reported with top boundary passing through $p^*$, then the time complexity of this iteration is $O(m' \log n)$. The detailed steps of this procedure is given in Algorithm ENUMERATETYPEC.

3.4.1.4 DELETETOP

The procedure DELETETOP works as follows. It starts from the root $P[1]$ by assigning $i = 1$. In each step, it considers both the children of $P[i]$, and
Algorithm 3.12: DELETETOP(P)

Input: The array P[1, ..., n] containing the max priority search tree T
Output: The top of T, and delete it from T

1 \( i = 1; \)
2 while \((2i \leq n) \land (y(P[2i]) < y(P[i]))\)
3 \( \lor ((2i + 1) \leq n) \land (y(P[2i + 1]) < y(P[i]))\) do
4 if \( 2i + 1 \leq n \) then
5 \( \text{if } y(P[2i]) < y(P[i]) \land y(P[2i + 1]) < y(P[i]) \) then
6 \( j = 2i + 1; \)
7 else
8 \( j = 2i; \)
9 \( \text{else if } y(P[2i + 1]) < y(P[i]) \) then
10 \( j = 2i + 1; \)
11 \( \text{else} \)
12 \( j = 2i; \)
13 else
14 \( j = 2i; \)
15 Swap \( P[i] \) and \( P[j] \) and set \( i = j; \)
16 Return(\( P[j] \));

choose the one, say \( P[j] \) having maximum \( y \)-coordinate. The tie, if arises, is broken arbitrarily. It then swaps \( P[i] \) and \( P[j] \), and moves to the next step by setting \( i = j \), provided \( P[j] \) is not a leaf node. Note that after the deletion of a point from MAXPST, it still remains in the array \( P \) maintaining the following invariants:

(i) MAXPST contains all the points \( p \in P \) satisfying \( y(p) < y(p^*) \), and
(ii) any point \( p \in P \) is in MAXPST if and only if \( y(p) < y(parent(p)) \).

Thus, the above two invariants say that a node \( P[j] \) has no left (resp. right) child if

- \( 2j > n \) (resp. \( 2j + 1 > n \)), or
- \( y \)-coordinate of \( P[2j] \) (resp. \( P[2j + 1] \)) is greater than the \( y \)-coordinate of the root of \( T \).
Note that in this dynamic setup under the deletion of the root of the MAXPST, the tree may not remain balanced after the deletion of some points. In other words, leaf nodes may appear in different levels. However, the length of the each search path will still be bounded by $O(\log n)$.

### 3.4.1.5 Highest3SidedUp

In Section 3.3.2, we described the procedure Highest3SidedUp; it is mentioned that if a node has only one child then that must be the left child. But, in the dynamic setup under the deletion, there may exist some node in MAXPST which has only the right child. Thus the necessary modification (in the case $p$ has only one child) is needed in the procedures CHECKLEFT, CHECKRIGHT, CHECKLEFTIN and CHECKRIGHTIN, that are invoked by Highest3SidedUp presented in Algorithm 3.3. However, the time complexity of the Highest3SidedUp query remains $O(\log n)$ using $O(1)$ extra-space.

### 3.4.2 Correctness and complexity

The correctness of the Algorithm 3.9 is easy to observe. The correctness of Algorithm 3.10 follows from the correctness of the procedures LEFTMOSTNE and RIGHTMOSTNW. Finally, the correctness of the Algorithm 3.11 follows from the invariants (i) and (ii) maintained by MAXPST after the deletion of its root node, as mentioned in Subsection 3.4.1.4. The following theorem states the final result.

**Theorem 3.2** Given a rectangular region $R$ containing a set of $n$ points, all the maximal empty axis parallel rectangles can be enumerated in $O(m \log n)$ time with $O(1)$ extra-space, where $m$ is the number of all maximal empty rectangles.

**Proof:** The algorithm ENUMERATEMER correctly enumerates all the axis-parallel MERs exactly once as each of ENUMERATETYPEA, ENUMERATE-TYPEB and ENUMERATETYPEC enumerates correctly each of the Type-A,
Type-B and Type-C rectangles respectively exactly once. The time complexity of each of the procedures \textsc{EnumerateTypeA}, \textsc{ConstructMaxPST} and \textsc{EnumerateTypeB} is \(O(n \log n)\). We now analyze the time complexity of the Algorithm \textsc{EnumerateTypeC}.

In each iteration of the while loop of \textsc{EnumerateTypeC}, we first delete the point \(p^*\) at the root using the procedure \textsc{DeleteTop}. This needs \(O(\log n)\) time since the root needs to move at most \(\log n\) levels for the deletion. Each iteration of the inner loop reports a Type-C axis-parallel MER with top boundary defined by \(p\). Here a point is identified that obstructs the corresponding curtain using the procedure \textsc{Highest3SidedUp}. This needs \(O(\log n)\) time. Thus, the Procedure \textsc{EnumerateTypeC} takes \(O(m' \log n + n \log n)\) time, where \(m'\) is the number of Type-C rectangles present in \(R\). Since, the number of Type-A and Type-B axis-parallel MERs are both \(O(n)\), the time complexity result follows. From the algorithm it is obvious that apart from the array \(P\), the extra-space used in the algorithm is \(O(1)\).

\begin{corollary}
A maximum area/perimeter MER can be computed in \(O(m \log n)\) time with \(O(1)\) extra-space, where \(m\) is the number of all possible axis-parallel MERs.
\end{corollary}

\section{Conclusion}

An in-place algorithm for constructing a MinMaxPST tree with a set of points in \(\mathbb{R}^2\) is presented in this chapter. The worst case time complexity of this algorithm is \(O(n \log n)\), it uses \(O(1)\) extra workspace, and stores the MinMaxPST in the same array that contains the input. All the standard query algorithms on the priority search tree [McCreight 1985] can be executed on this data structure in logarithmic time using \(O(1)\) extra-space. The implementation of this algorithm is available in [Pratt 2011]. As an application of this algorithm, we proposed an in-place algorithm of enumerating all the axis-parallel maximal empty rectangles (MER) among the point set \(P\) inside an axis-parallel rectangular region \(R\). It runs in \(O(m \log n)\) time and \(O(1)\) extra-space apart from the input array \(P\), where \(m\) is the number of axis-parallel MERs present on \(R\).
4.1 Introduction

Prune-and-search is an excellent algorithmic paradigm for solving various optimization problems. The main idea of prune-and-search is to reduce the search space by pruning a fraction of the input elements and recurse on the remaining valid input elements.

In this chapter, we provide a general scheme for prune-and-search technique and show how to implement it in space-efficient manner. Our technique can be applied to a large number of problems which accept prune-and-search.
4.2 General scheme for prune-and-search

The standard scheme for prune-and-search algorithm is given in Algorithm 4.1. Here, the search-space is reduced by pruning. In each iteration of the while-loop, a constant fraction $\frac{1}{\delta}$ ($\delta > 1$) of the input objects $S$ are pruned depending on the SPECIFIC-PRUNING-CONDITION computed in that iteration by INTERMEDIATE-COMPUTATION step. Finally, after the completion of the while-loop, the search-space becomes very small and by brute-force mechanism one can compute the optimum.

**Algorithm 4.1**: Prune-and-search($S$): A standard scheme

**Input**: A set of $n$ objects $S$ for the optimization problem  
**Output**: The optimum solution of the problem

1. while $|S| \geq \delta$ do
   2. Arbitrarily pair-up the objects of $S$ to form $\frac{|S|}{2}$ disjoint pairs. We denote this set of pairs as $PAIR$. 
   3. INTERMEDIATE-COMPUTATION;
   4. (*Pruning Step*)
   5. forall $(s_i, s_j) \in PAIR$ do
   6. if $(s_i, s_j)$ satisfies SPECIFIC-PRUNING-CONDITION then
   7. Prune $s_i$ or $s_j$ depending on their value;
   8. (*Finally, when $|S| < \delta$*) compute the optimum result in a brute-force manner.

Time complexity of this general prune-and-search type algorithm is $T(n) = T((1 - \frac{1}{\delta})n) + E(n) + O(n)$, where $E(n)$ is the time needed for INTERMEDIATE-COMPUTATION. Note that as this needs to distinguish between the pruned objects and active objects after each iteration, it needs at-least $O(n)$ flag bits and total space complexity is $S(n) = ES(n) + O(n)$, where $ES(n)$ is the space needed for INTERMEDIATE-COMPUTATION.

4.3 For in-place model

In the in-place model, swapping elements in the input array is permitted. Thus, after each iteration, we can move the pruned objects to the one end of the input array, and in the rest of the execution, the computation can
be done considering the valid portion of the array. Here, we keep objects $s_i$ and $s_j$ of each pair $(s_i, s_j) \in PAIR$ in consecutive locations of the array. Remember that we have to ensure that (i) \textsc{Intermediate-Computation} can be implemented in an in-place manner, and (ii) after \textsc{Intermediate-Computation}, both the objects $s_i$ and $s_j$ of each pair $(s_i, s_j) \in PAIR$ remain in the consecutive location of the array, i.e., they are identifiable; otherwise, in the Pruning Step, we will be in trouble. The time complexity would be: $T(n) = T((1 - \frac{1}{\delta})n) + E_I(n) + O(n)$, where $E_I(n)$ is the time needed for the in-place version of the \textsc{Intermediate-Computation}. Note that if $E_I(n) = E(n)$, then the time complexity remains same as the original. Extra-space complexity is $S(n) = ES_I(n) + O(1)$, where $ES_I(n)$ is the extra-space needed for the in-place \textsc{Intermediate-Computation} apart from the array containing the input. If $ES_I(n) = O(1)$, then the total space complexity is also $O(1)$.

Thus, we have the following theorem:

**Theorem 4.1** Any prune-and-search algorithm having the standard scheme given in Algorithm 4.1 with time complexity $T(n)$ can be implemented in an in-place manner using $O(1)$ extra-space with same time complexity $T(n)$, provided the following two conditions are satisfied in each iteration:

(i) \textsc{Intermediate-Computation} is implementable in an in-place manner using $O(1)$ extra-space without deteriorating its original time complexity, and

(ii) After in-place \textsc{Intermediate-Computation}, each paired objects $(s_i, s_j) \in PAIR$ can be identified correctly.

**Remark 4.1** The above theorem gives a direction on how to simulate a prune-and-search algorithm in an in-place manner. We refer Bose et al. [Bose 2007] to see how to simulate a divide-and-conquer algorithm in an in-place manner.
4.4 For read-only model

The problem in the read-only model is that one can not move the pruned objects to the other end. So, if one needs to distinguish the active objects from the pruned ones, the most trivial way is to use a bit array of size $O(n)$ which will act as a mark bit for active/pruned objects. We will show how to maintain the information about the pruned elements using $O(\log n)$ extra-space (i.e., $O(\log^2 n)$ bits). During an iteration, the set of all pruned objects (prior to this iteration) are identified by re-running all the tests developed in its preceding iterations. Since there are $O(\log n)$ levels of recursion, we need $O(\log^2 n)$ such re-runs. As a result, there is an overhead in the running time.

The total number of iterations of the while-loop is $K = O(\log n)$. In each iteration, the Specific-Pruning-Condition, the set $S$ and the set $PAIR$ are different. We use $SPC_t$, $S_t$ and $PAIR_t$ to denote the Specific-Pruning-Condition, the set $S$ and the set $PAIR$ of the $t$-th iteration, respectively, where $t \in \{1, 2, \ldots, K\}$.

Notice that in the $t$-th iteration of the while-loop if we remember the $SPC_t$, then in the $(t+1)$-th iteration using that $SPC_t$ we can distinguish the active objects (i.e., members of $S_{t+1}$) and the pruned ones provided we can correctly identify the members of $PAIR_t$. If each $SPC_t, t = 1, 2, \ldots, K$ can be stored using $O(1)$ space, then the total amount of extra space required for storing all of them will not be more than $O(\log n)$. If we could store $PAIR_t$ after the $t$-th iteration for the processing of $(t+1)$-th iteration, then the problem becomes easy. But, remember that we can not explicitly store the members of $PAIR_t$ after the $t$-th iteration. Because it will take $O(n)$ space.

**Pairing scheme:** Now, we will describe a pairing scheme which will enable us to recognize the members of $PAIR_t$ without explicitly storing them after the $t$-th iteration, where $t = 1, 2, \ldots, K$. The pairing scheme will satisfy the following invariants:

**Invariant 4.1**  
(i) If an object $s \in S$ is pruned at some iteration $t_1$, then it will not participate to form a pair in any $t_2$-th iteration, where $1 \leq t_1 < t_2 \leq K$.

(ii) If $(p, q) \in PAIR_t$ ($t < K$) and none of the objects $p$ and $q$ are pruned...
4.4. For read-only model

at the end of the $t$-th iteration, then $(p, q)$ will again form a valid pair at the $(t+1)$-th iteration, i.e., $(p, q) \in PAIR_{t+1}$.

(iii) If $(p, s)$ are paired at the $(t+1)$-th iteration of the while-loop, and $(p, q) \in PAIR_t$ ($t < K$) where $s \neq q$, then there exist some $r \in S_t$ such that $(r, s)$ were paired at the $t$-th iteration (i.e., $(r, s) \in PAIR_t$), and $q \notin r$ were pruned at the end of the $t$-th iteration.

Let the input objects be given in the read-only array $R[1, 2, \ldots, n]$. We denote the total number of variables and the time needed to enumerate all the members of $PAIR_t$ and $S_t$ as $w(t)$ and $f(t)$, respectively, where $t = 1, 2, \ldots, K$. The algorithm which can enumerate $PAIR_t$ and $S_t$ are denoted as $AP_t$ and $AS_t$, respectively.

In the first iteration of the while-loop, all the elements of the array $R[1, 2, \ldots, n]$ are members of $S_1$. In this iteration, $(R[2i-1], R[2i]), i = 1, 2, \ldots, \frac{n}{2}$ are paired, in other words, $PAIR_1 = \{(R[2i-1], R[2i]), i = 1, 2, \ldots, \frac{n}{2}\}$. So, the members of $PAIR_1$ and $S_1$ can be enumerated in $O(n)$ time using $O(1)$ variables. So, $f(1) = O(n)$ and $w(1) = 1$. So, we have the algorithms $AP_1$ and $AS_1$.

Let’s assume that we can correctly enumerate all the members of $PAIR_{t-1}$ using $w(t-1)$ variables and in $f(t-1)$ time. Now, we will show how to enumerate the members of $PAIR_t$ and $S_t$ using $AP_{t-1}$.

$AP_t$ and $AS_t$ will use a temporary variable $STATUS_{t-1}$. Initially, it is set to $-1$.

While a member $(p, q) \in PAIR_{t-1}$ is enumerated by $AP_{t-1}$, it is tested with $SPC_{t-1}$. Here two situations may arise.

- **Case 1:** one object from the pair $(p, q)$ is pruned: Without loss of generality, assume that $p$ is pruned. So, $p$ is ignored and $q$ becomes an active member of $S_t$. Again, here depending on the content of $STATUS_{t-1}$ following two sub-cases may arise.

  - **Case 1.1:** $STATUS_{t-1}$ contains $-1$: Here, the index of the active object $q$ in the input array $R$ is stored in $STATUS_{t-1}$.
• Case 1.2: STATUS_{t-1} contains a valid entry \((\neq -1)\): \(q\) is paired with the object residing at \(R[STATUS_{t-1}]\), i.e., \((q, R[STATUS_{t-1}]) \in PAIR_t\). STATUS_{t-1} is set to \(-1\).

• Case 2: None of the objects \(p, q\) is pruned: Here both the objects \(p, q\) become members of \(S_t\) and the pair is considered to be a valid pair for \(t\)-th iteration, i.e., \((p, q) \in PAIR_t\).

So, the total space required by both \(AP_t\) and \(AS_t\) is \(w(t) = w(t-1) + 1 = O(t)\) and time required is \(f(t) = f(t-1) + n = O(tn)\). Thus, we have the following lemma:

**Lemma 4.1** If the objects are given in a read-only array, then the members of \(S_t\) and \(PAIR_t\) can be enumerated in \(O(tn)\) time using \(O(t)\) extra-space, where \(1 \leq t \leq K\) and \(K\) is the total number of iterations of the while-loop.

As the number of iterations is \(O(\log n)\), we have the following theorem:

**Theorem 4.2** Any prune-and-search algorithm having the standard scheme given in Algorithm 4.1 can be implemented in a read-only environment using \(O(s)\) extra-space, provided \textsc{Intermediate-Computation} is implementable in the pairing scheme using \(O(s)\) extra-space, where \(s \geq \log n\).

**Remark 4.2** Using this pairing scheme, if we want to enumerate from a specific valid member to the next one, we have to remember the present status, i.e., if it is in the \(t\)-th iteration, then we have to know what are the contents of the variables \(STATUS_1, STATUS_2, \cdots, STATUS_{t-1}\).

### 4.5 Selection in the pairing scheme

Selection is a fundamental algorithm which is frequently used in the \textsc{Intermediate-Computation} of some prune-and-search algorithms. In this section, we consider the following two selection algorithms in the read-only environment.
4.5. Selection in the pairing scheme

- **ALGO-MR**: Munro and Raman’s [Munro 1996] selection algorithm which takes $O(n^{1+\epsilon})$ time and $O(\frac{1}{\epsilon})$ extra-space, where $\epsilon$ is a small fixed positive constant ($\sqrt{\log \log n} < \epsilon < 1$).

- **ALGO-RR**: Raman and Ramnath’s [Raman 1999] selection algorithm, which is slightly faster than ALGO-MR. Using $O(\log n)$ extra-space, it runs in $O(n \log^2 n)$ time.

First, we describe these algorithms without considering the pairing scheme. Next, we show how much space and time is needed if we implement them in the $t$-th iteration of the while-loop of the standard prune-and-search scheme given in Algorithm 4.1 using the pairing scheme stated in Section 4.4.

4.5.1 ALGO-MR

Given a set of $n$ points in $\mathbb{R}$ in a read-only array $P$, the algorithm ALGO-MR is designed by using a set of procedures $A_0, A_1, A_2, \ldots, A_k$, where procedure $A_i$ finds the median by evoking the procedure $A_{i-1}$ for $i \in \{1, 2, \ldots, k\}$. The procedures $A_0, A_1, A_2, \ldots, A_k$ are stated below.

**Procedure $A_0$**: In the first iteration, after checking all the elements in $P$, it finds the largest element $p_{(1)}$ in linear time. In the second iteration, it finds the second largest $p_{(2)}$ by checking only the elements which are less than $p_{(1)}$. Proceeding in this way, in the $j$-th iteration it finds the $j$-th largest element $p_{(j)}$ considering all the elements in $P$ that are less than $p_{(j-1)}$. In order to get the median we need to proceed up to $j = \lfloor \frac{n}{2} \rfloor$. Thus, this simple median finding algorithm takes $O(n^2)$ time and $O(1)$ extra-space.

**Procedure $A_1$**: It divides the array $P$ into blocks of size $\sqrt{n}$ and in each block it finds the median using Procedure $A_0$. After computing the median $m$ of a block, it counts the number of elements in $P$ that are smaller than $m$, denoted by $\rho(m)$, by checking all the elements in the array $P$. It maintains two best block medians $m_1$ and $m_2$, where $\rho(m_1) = \max\{\rho(m)|\rho(m) \leq \frac{n}{2}\}$, and $\rho(m_2) = \min\{\rho(m)|\rho(m) \geq \frac{n}{2}\}$. Thus, this iteration needs $O(n \sqrt{n})$ time. After this iteration, all the elements $P[i]$ satisfying $P[i] < m_1$ or $P[i] > m_2$ are considered as invalid. However, we do not need any mark bit; only we
need to remember $m_1$ and $m_2$. In the next iteration, we again consider same set of blocks, and compute the median ignoring the invalid elements.

Since, in each iteration $\frac{1}{4}$ fraction of the existing valid elements are considered as invalid, we need at most $O(\log n)$ iterations to find the median $\mu$. Thus the time complexity of this procedure is $O(n\sqrt{n}\log n)$.

Procedure $A_2$: It divides the whole array into $n^{1/3}$ blocks each of size $n^{2/3}$, and computes the block median using the procedure $A_1$. Thus, the overall time complexity of this procedure for computing the median is $O(n^{1+\frac{1}{3}}\log^2 n)$.

Proceeding in this way, the time complexity of the procedure $A_k$ will be $O(n^{(1+\frac{1}{k+1})}\log^k n)$. As it needs a stack of depth $k$ for the recursive evoking of $A_{k-1}, A_{k-2}, \ldots, A_0$, the space complexity of this algorithm is $O(k)$.

Setting $\varepsilon = \frac{1}{k+1}$, gives the running time as $O\left(n^{1+\varepsilon}\sqrt{\frac{\log n}{\log \log n}}\right)$. If we choose $\varepsilon$ such that $n^\varepsilon = \log^\frac{1}{\varepsilon} n$, then we have $\varepsilon = \sqrt{\frac{\log \log n}{\log n}}$. This gives the running time $O\left(n^{1+2\varepsilon}\sqrt{\frac{\log n}{\log \log n}}\right)$, which is of $O(n^{1+2\varepsilon})$. Thus the general result is as follows:

**Result 4.1** For a set of $n$ points in $\mathbb{R}$ given in a read-only array, the median can be found using ALGO-MR in $O(n^{1+\varepsilon})$ time with $O(\frac{1}{\varepsilon} t)$ extra-space, where $2\sqrt{\frac{\log \log n}{\log n}} \leq \varepsilon < 1$.

**ALGO-MR in the $t$-th iteration of the pairing scheme:** In the $t$-th iteration of the while-loop, using the pairing scheme, all the valid members can be enumerated in $O(tn)$ time using $O(t)$ extra-space (see Lemma 4.1 and Remark 4.2). As procedure $A_0$ needs to scan all the valid elements $n$ times to find the median, it needs $O(n)$ number of enumerations. So, $A_0$ will take $O(tn^2)$ time and $O(t)$ extra-space (as space can be reused). Similarly, $A_1$ takes $O(tn^{1+\frac{1}{2}}\log n)$ time and $O(t)$ extra-space. Finally, $A_k$ takes $O(tn^{(1+\frac{1}{k+1})}\log^k n)$ time and $O(kt)$ extra-space, since we have to remember $k$ levels of the recursion and each level needs to remember the status of the pairing scheme which is of size $O(t)$. We can choose the constant $k$ appropriately depending on the available work-space.

**Lemma 4.2** In the $t$-th iteration of the while-loop, using the pairing scheme, the median can be found using ALGO-MR in $O(n^{1+\varepsilon})$ time with $O(\frac{1}{\varepsilon} t)$ extra-space, where $\varepsilon$ is an user defined constant satisfying $2\sqrt{\frac{\log \log n}{\log n}} \leq \varepsilon < 1$. 
4.5.2 ALGO-RR

A set $P$ of $n$ points in $\mathbb{R}$ stored in a read-only array $P$ is partitioned into three groups by any pair of points $m_1, m_2 \in P \ (m_1 < m_2)$:

**Group(i):** all the points in $P$ with key value less than $m_1$,

**Group(ii):** all the points in $P$ with key value greater than $m_2$, and

**Group(iii):** all the points in $P$ with key-value in between $m_1$ and $m_2$.

A pair $m_1, m_2 \in A$ is called an approximate median pair for points in $P$ if number of points satisfying each of the three groups is strictly less than $\lceil \frac{n}{2} \rceil$.

ALGO-RR is basically an iterative algorithm; in each iteration, it finds approximate median pairs $m_1$ and $m_2$. In the next iteration, it considers elements of one group only by ignoring others and do the same. So, in $O(\log n)$ iterations, it finds the exact $k$-th median.

The approximate median pair is computed by divide-and-conquer algorithm. It partitions the whole read-only array into two equal parts and finds approximate median pair from each of these two parts (recursively). Let $m_1, m_2, m_3, m_4 \ (m_1 < m_2 < m_3 < m_4)$ be these four points of the two approximate median pairs. Finally, by scanning the whole read-only array, it determines how many points are smaller than each of these four points. Let $m_t$ be the largest amongst these four points such that the number of points smaller than it is strictly less than $\lceil \frac{n}{2} \rceil$. It sets $(m_t, m_{t+1})$ as the approximate median pair of the whole array.

Finding an approximate median pair in an iteration needs $O(n \log n')$ time, where $n'$ is the number of elements considered at that iteration. Hence, the total time complexity of this algorithm is $O(n \log^2 n)$. The algorithm will take $O(\log n)$ extra-space as the depth of recursion in each iteration is $O(\log n)$. Hence, we have the following result:

**Result 4.2** For a set of $n$ points in $\mathbb{R}$ given in a read-only array, the median can be found using ALGO-RR in $O(n \log^2 n)$ time with $O(\log n)$ extra-space.
ALGO-RR in the $t$-th iteration of the pairing scheme: In the pairing scheme, we can simulate the ALGO-RR. Using similar argument given for ALGO-MR, it can be shown that:

**Lemma 4.3** In the $t$-th iteration of the while-loop, using the pairing scheme, the median can be found using ALGO-RR in $O(tn \log^2 n)$ time with $O(t \log n)$ extra-space.

Combining both Lemma 4.2 and 4.3, we have the following:

**Lemma 4.4** In the $t$-th iteration of the while-loop, using the pairing scheme, the median can be found in

\[(i) \; O(tn^{1+\frac{1}{k} \log^k n}) \text{ time with } O(tk) \text{ extra-space, where } k \text{ is a fixed natural number,}\]

\[(ii) \; O(tn \log^2 n) \text{ time with } O(t \log n) \text{ extra-space.}\]

### 4.6 Conclusion

In this chapter, we give a general sketch for implementing any prune-and-search algorithm which follows the standard scheme, given in Algorithm 4.1, in space-efficient manner. In next three chapters (Chapter 5, 6 and 7), we study specific problems which accept prune-and-search solutions and show how to implement them in space-efficient manner. The considered problems are: (i) minimum enclosing circle of a point set in $\mathbb{R}^2$, (ii) convex hull of a sorted point set in $\mathbb{R}^2$, and (iii) linear programming problems with two and three variables. All these problems have tremendous practical usage apart from theoretical implication. We hope that our technique can be widely applied to many other problems as well.
Chapter 5
Minimum Enclosing Circle with Limited Work-space

Contents

5.1 Introduction ........................................... 73
5.2 Overview of Megiddo's algorithm .................... 74
5.3 In-place implementation of MEC ..................... 77
  5.3.1 In-place implementation of constrained-MEC ... 78
5.4 When the memory is read-only ....................... 83
  5.4.1 Constrained-MEC in read-only setup .......... 84
  5.4.2 (Unconstrained) MEC in the read-only setup .. 88
5.5 Conclusion ............................................. 89

5.1 Introduction

The minimum enclosing circle (MEC) for a set of points $P$ in $\mathbb{R}^2$ is defined to be a circle of minimum radius that encloses all the points in $P$. The problem of finding the minimum enclosing circle has several vital applications. One such example is in planning the location of placing a shared facility like a hospital, gas station, or sensor device etc. The center of the minimum enclosing circle will be the desired location for placing the facility. In the location theory community, this type of problem is known as the 1-center problem.

Let $P[1, \ldots, n]$ be an array containing a set $P$ of $n$ points in $\mathbb{R}^2$. The objective is to compute the center and radius of the circle of minimum radius that
contains all the points in \( P \). We assume that the input points are in general position, i.e., no two points have the same \( x \)- or \( y \)-coordinate; no three points are collinear; no four points are co-circular; and the distance of each pair of points is distinct. However, the assumptions can be relaxed by incorporating a few checks in the algorithm. First, we explain Megiddo’s linear time algorithm [Megiddo 1983b] that uses linear amount of extra-space. Next, we propose a linear time in-place algorithm for this problem using constant number of extra variables. Here, the data can be moved in the array \( P \), but at the end of the execution all the points are available in the array \( P \), possibly in a different permutation. An interested reader can find a similar result on linear programming with two variables in [Brönnimann 2004b]. Finally, we present an algorithm where the array \( P \) containing the input points is read-only.

5.2 Overview of Megiddo’s algorithm

Let \( P[1, \ldots, n] \) be an array containing \( n \) points. For the self-completeness, we will describe Megiddo’s linear time algorithm [Megiddo 1983b] for the MEC problem for the points in \( P \) in Algorithm 5.1. In each iteration of the while-loop, it invokes INTERMEDIATE-COMPUTATION-for-MEC which is described in detail in Algorithm 5.2. This in turn calls the procedure CONSTRAINED-MEC, given in Algorithm 5.3.

Let \( \pi^* \) be the center of the desired MEC. At each iteration, it identifies a pair of mutually perpendicular lines such that the quadrant in which \( \pi^* \) lies can be identified, and a constant fraction of points in \( P \) can be pruned.

The correctness of the algorithm is given in [Megiddo 1983b]. An iteration of the while-loop of the algorithm MEC(\( P \)) with the set of points \( P \) needs \( O(|P|) \) time, and it deletes at least \( \lfloor \frac{|P|}{16} \rfloor \) points from \( P \). Thus, Megiddo’s algorithm for the MEC problem executes the while-loop at most \( O(\log n) \) times. Its total running time is \( O(n) \) using \( O(n) \) extra-space apart from the input array.
5.2. Overview of Megiddo’s algorithm

Algorithm 5.1: MEC(\(P\))

Input: An array \(P[1, \ldots, n]\) containing a set \(P\) of \(n\) points in \(\mathbb{R}^2\).
Output: The center \(\pi^*\) of the minimum enclosing circle of the points in \(P\).

1. While \(|P| \geq 16\) do
2. Arbitrarily pair up the points in \(P\). Let \(PAIR = \{(P[2i-1], P[2i]), i = 1, 2, \ldots, \lfloor |P|/2 \rfloor\}\) be the set of aforesaid disjoint pairs;
3. Intermediate-Computation-for-MEC; (* It returns a quadrant Quad defined by two perpendicular lines \(L_H\) and \(L_V\) *)
4. (* Pruning step *)
5. For all pair of points \((P[2i], P[2i+1]) \in PAIR\) do
6. If the bisector line \(L_i\) defined by the pair \((P[2i], P[2i+1])\) does not intersect the quadrant Quad then
7. discard one of \(P[2i]\) and \(P[2i+1]\) from \(P\) which lies on the side of the quadrant Quad with respect to the bisector line \(L_i\);
8. (* Finally, when \(|P| < 16\) *) compute the minimum enclosing circle in brute force manner.

Algorithm 5.2: Intermediate-Computation-for-MEC(\(P\))

Input: An array \(P[1, \ldots, n]\) of points in \(\mathbb{R}^2\).
Output: The triplet \((L_H, L_V, Quad)\), where \(L_H\) and \(L_V\) are a pair of mutually perpendicular lines and \(Quad\) is one among the four quadrants defined by \(L_H\) and \(L_V\).

1. Step 1: Let \(L_i\) denote the bisector of the pair of points \((P[2i-1], P[2i]) \in PAIR\), and \(\alpha(L_i)\) denote the angle of \(L_i\) with the \(x\)-axis. Compute the median \(\mu\) of \(\{\alpha(L_i), i = 1, 2, \ldots, \lfloor |P|/2 \rfloor\}\);
2. Step 2: Arbitrarily pair up \((L_i, L_j)\) where \(\alpha(L_i) \leq \mu\) and \(\alpha(L_j) \geq \mu\). Let \(M\) be the set of these \(\lfloor |P|/4 \rfloor\) pairs of lines;
3. We split \(M\) into two subsets \(M_P\) and \(M_I\), where
4. \(M_P = \{(L_i, L_j) | \alpha(L_i) = \alpha(L_j) = \mu\} \) (* parallel line-pairs *) and
5. \(M_I = \{(L_i, L_j) | \alpha(L_i) \neq \alpha(L_j)\} \) (* intersecting line-pairs *)
6. For each pair \((L_i, L_j) \in M_P\) do
7. compute \(y_{ij} = \frac{y_j - y_i}{d_{ij}}\), where \(d_{ij}\) = distance of \(L_i\) from the line \(y = \mu x\)
8. For each pair \((L_i, L_j) \in M_I\) do
9. Let \(a_{ij}\) = point of intersection of \(L_i\) & \(L_j\), and \(b_{ij}\) = projection of \(a_{ij}\) on \(y = \mu x\). Compute
10. \(x_{ij}\) = signed distance of the pair of points \((a_{ij}, b_{ij})\), and
11. Next, compute the median \(y_m\) of the \(y_{ij}\) values corresponding to all the pairs in \(M\). Let \((x', y')\) be the corresponding intersection point;
12. Step 3: Consider the line \(L_H : y - y' = \mu(x - x')\), which is parallel to \(y = \mu x\) and passes through the point \((x', y')\);
13. Execute the algorithm Constrained-MEC(\(P, L_H\)) to decide on which side of the line \(L_H\) the center \(\pi^*\) of the minimum enclosing circle lies;
14. Step 4: Let \(M'_P = \{(L_i, L_j) \in M_P\} \) such that \(a_{ij}\) and \(\pi^*\) lie in the different sides of the line \(L_H\) ;
15. Compute the median \(x_m\) of \(x_{ij}\) values for the line-pairs in \(M'_P\). Let \((x'', y'')\) be the corresponding intersection point. Define a line \(L_V : y - y'' = \frac{1}{\mu}(x - x'')\) perpendicular to \(L_H\) and passing through the point \((x'', y'')\);
16. Step 5: Execute the algorithm Constrained-MEC(\(P, L_V\)), and decide in which side of \(L_V\) the point \(\pi^*\) lies ;
17. Step 6: Report \((L_H, L_V, Quad)\); Note that \(Quad\) can be encoded in 2 bits.
Algorithm 5.3: CONSTRAINED-MEC(\(P, L\))

Input: An array \(P\{1, \ldots, n\}\) of points in \(\mathbb{R}^2\), and a line \(L\) (* assumed to be vertical *).

Output: The center \(m^*\) of the minimum enclosing circle of the points in \(P\) on the line \(L\) and the side on which center of the unconstrained MEC lies.

1. **Step 1:** \(P' = P\);
2. **while** \(|P'| \geq 3\) **do**
3.  **Step 1.1:** Arbitrarily pair up the points in \(P'\). Let \(PAIR' = \{(P[2i−1], P[2i]), i = 1, 2, \ldots, \lfloor \frac{|P'|}{2} \rfloor \}\) be the set of aforesaid disjoint pairs;
4.  **Step 1.2:** Let \(\ell_i\) denote the perpendicular bisector of the pair of points \((P[2i−1], P[2i]) \in PAIR'\), \(i = 1, 2, \ldots, \lfloor \frac{|P'|}{2} \rfloor\).
5.  Let \(\ell_i\) intersect \(L\) at a point \(q_i\), and \(Q = \{q_i, i = 1, 2, \ldots, \lfloor \frac{|P'|}{2} \rfloor\}\);
6.  **Step 1.3:** Compute the median \(m\) of the \(y\)-coordinate of the members in \(Q\);
7.  **Step 1.4:** (* Test on which side (above or below) of \(m\) the center \(m^*\) of the constrained MEC lies (i.e., whether \(m^* < m\) or \(m^* > m\) as follows: *)
8.  Identify the point[s] \(F \in P'\) that is/are farthest from \(m\);
9.  **if** the projection of all the members in \(F\) on \(L\) are in different sides of \(m\) **then**
10. \(m^* = m\ (*\ center\ of\ the\ constrained\ minimum\ enclosing\ circle\ on\ the\ line\ \(L\)*);
11. **break**;
12. **else**
13. \(\text{(* i.e., the projection of all the members in } F\text{ on } L\text{ are in the same side (above or below) of } m\text{*)} m^*\text{ lies in that side of } m\text{ on the line } L\)
14. **Step 1.5:** (* Pruning Step *) Without loss of generality, assume, that \(m^* > m\). Then for each bisection line \(\ell_{p,q}\) (defined by the pair of points \((p, q) \in PAIR'\) that cuts the line \(L\) below the point \(m\), we can delete one point among \(p\) and \(q\) from \(P'\) such that the said point and the point \(m\) lie in the same side of \(\ell_{p,q}\).
15. **Step 2:** (* the case when \(|P'| = 2\*)
16. **if** the perpendicular bisector of the two members of \(P'\) intersects \(L\) **then**
17. \(m^* = \) the point of intersection ;
18. **else**
19. \(\text{(* the perpendicular bisector of the members of } P'\text{ is parallel with } L\text{*)}\)
20. \(m^* = \) projection of the farthest point of \(P'\) on \(L\).
21. **Step 3:** (* Decide in which side of \(L\) the center \(\pi^*\) of the unconstrained MEC lies *)
22. Let \(F\) be the set of points in \(P'\) that are farthest from \(m^*\);
23. **if** \(|F| = 1\ **then** \(\pi^* = m^*\) and the only point \(p_1 \in F\) lie in the same side of \(L\);
24. **else**
25. \(\text{(* we need to check whether the convex polygon formed by the points in } F\text{ contain } m^*\text{ or not as follows *)}\)
26. **Let** \(F_1\) and \(F_2\) be two subsets of \(F\) that lie in two different sides of \(L\) respectively;
27. \(F_1 \cup F_2 = F\).
28. Let \(m'\) be any point on \(L\) that is below \(m^*\).
29. **Find** two points \(p_1, p_2 \in F_1\) such that \(\triangle m'm^*p_1 = \max\{\triangle m'm^*p|p \in F_1\}\) and \(\triangle m'm'^*p_2 = \min\{\triangle m'm'^*p|p \in F_1\}\).
30. **Now consider** each point \(q \in F_2\) and test whether \(\pi \in \Delta p_1 q p_2\).
31. **Similarly, find** \(p_1, p_2 \in F_2\) such that \(\triangle m'm^*p_1 = \max\{\triangle m'm^*p|p \in F_2\}\) and \(\triangle m'm'^*p_2 = \min\{\triangle m'm'^*p|p \in F_1\}\).
32. **Consider** each point \(q' \in F_1\) and test whether \(m^* \in \Delta p_1 q'p_2\);
33. **If** any one of these triangles contain \(m^*\), then the convex polygon defined by the points in \(F\) contains \(m^*\). Here, the algorithm reports \(\pi^* = m^*\), and stops.
34. **Otherwise,** either \((p_1, p_2)\ or \((p_1, p_2)\) define the diagonal (farthest pair of points) in \(F\). Let \(q\) be the midpoint of the diagonal. Here, \(\pi^*\) and \(q\) will lie in the same side of \(L\);
5.3 In-place implementation of MEC

In this section, we will show that Megiddo's algorithm (stated in Section 5.2) can be made in-place with the same time complexity. It is to be noted that this algorithm follows the standard scheme for the Prune-and-Search mentioned in Algorithm 4.1. In order to show that this algorithm is implementable in an in-place manner, we have to show the following conditions are satisfied (see Theorem 4.1):

(i) **Intermediate-Computation-for-MEC** is implementable in an in-place manner using $O(1)$ extra-space,

(ii) After the in-place execution of **Intermediate-Computation-for-MEC**, each paired points $(p, q) \in \text{PAIR}$ can be identified correctly.

The in-place **Intermediate-Computation-for-MEC** maintains the following invariant which satisfies the condition (ii).

**Invariant 5.1** In each iteration of the while-loop of MEC($P$), throughout the execution of the procedure **Intermediate-Computation-for-MEC**, the pair of valid points $(p, q) \in \text{PAIR}$ defining $L_i$ (their perpendicular bisector), will remain in consecutive locations of the input array $P$, for each $i = 1, 2, \ldots \left\lfloor \frac{|P|}{2} \right\rfloor$, where $|P|$ denotes the number of valid points in that iteration.

Note that we may succeed in making all the steps of **Intermediate-Computation-for-MEC** in-place separately but there may be problems while integrating them together. For an example, one can easily be able to make the **Constrained-MEC** (Step 3 & 5 of the procedure **Intermediate-Computation-for-MEC**) in-place, but one needs to assure that after this, one will be able to figure out the chosen pair of bisectors satisfying the condition mentioned in Step 2 of the procedure **Intermediate-Computation-for-MEC**, as this will be required in Step 4 of the same procedure. We will ensure this integration.

We will extensively use the fact that the median of a set of $n$ numbers stored in an array of size $n$ can be computed in an in-place manner in $O(n)$ time using $O(1)$ extra-space [Carlsson 1995].
In Step 1 of the procedure INTERMEDIATE-COMPUTATION-for-MEC, we can compute the median angle $\mu$ in an in-place manner. Note that we need not have to store \( \{L_i, i = 1, 2, \ldots, \lceil \frac{|P|}{4} \rceil \} \) as one can compute them on demand with the knowledge of \((P[2i - 1], P[2i]) \in PAIR\).

Step 2 of the procedure INTERMEDIATE-COMPUTATION-for-MEC can be made in-place in \( O(|P|) \) time and \( O(1) \) extra-space as follows: identify \( \lfloor \frac{|P|}{4} \rfloor \) pairs \((L_i, L_j) (\alpha(L_i) \leq \mu \text{ and } \alpha(L_j) \geq \mu)\), and for each pair accumulate the tuple of four points \((P[2i - 1], P[2i], P[2j - 1], P[2j]) \in M \) in consecutive locations of the array. Note that this consecutive arrangement will help in computing \( x_{ij} \) and \( y_{ij} \) for \( L_i \) and \( L_j \) (see Step 2 of the Procedure INTERMEDIATE-COMPUTATION-for-MEC) on the fly. So, we maintain the following invariant:

**Invariant 5.2** During the execution of Steps 3-6 of the procedure INTERMEDIATE-COMPUTATION-for-MEC, the four points of each tuple \((p, q, r, s) \in M\) will remain in consecutive locations of the input array \( P \).

We store the number of input points in a variable \( n \), and use a variable \( \nu \) to denote the current size of the array \( P \) (i.e., the number of valid points after pruning). In each iteration of the while-loop in Algorithm MEC, after pruning, the deleted points are moved at the end of the array, and \( \nu \) is updated with the number of non-deleted points. We have already shown that Steps 1-2 of INTERMEDIATE-COMPUTATION-for-MEC can be made in-place. In the next subsection, we show that Steps 3-6 of INTERMEDIATE-COMPUTATION-for-MEC can also be made in-place satisfying Invariants 5.1 and 5.2 (see Lemma 5.2 in the Subsection 5.3.1). Thus, we have the following result.

**Theorem 5.1** Minimum enclosing circle for a set of \( n \) points in \( \mathbb{R}^2 \) can be computed in an in-place manner in \( O(n) \) time with \( O(1) \) extra-space.

### 5.3.1 In-place implementation of CONSTRAINED-MEC

In a particular iteration of the while-loop of the algorithm MEC, we have all non-deleted points stored in consecutive locations of the array \( P \) starting from its leftmost cell. In Step 3 & 5 of the procedure INTERMEDIATE-COMPUTATION-for-MEC, we use the procedure CONSTRAINED-MEC to compute the center \( m^* \) of the minimum enclosing circle for these points where \( m^* \)
is constrained to lie on the given line \( L \) and to decide on which side of the given line \( L \) the center \( \pi^* \) of the unconstrained MEC lies. Without loss of generality, let us assume that \( L \) is a vertical line. A straightforward way to implement this procedure in an in-place manner without maintaining Invariants 5.1 and 5.2 is as follows.

Find the median point \( m \) on the line \( L \) among the points of intersection of the lines \( \ell_i \) and \( L \) for \( i = 1, 2, \ldots, n \over 2 \) in an in-place manner using the algorithm given in [Carlsson 1995], where the points of intersection are computed on the fly. This needs \( O(n) \) time. Next, inspect all the points to decide whether \( m^* \) is above or below \( m \) as follows. Let \( F \) denote the set of points in \( P \) which are farthest from \( m \).

- If the projection of the members in \( F \) on the line \( L \) lie in both the sides of \( m \), then \( m^* = m \).
- If the projection of all the members in \( F \) on the line \( L \) lie in the same side (above or below) of \( m \), then \( m^* \) lies in that side of \( m \) on the line \( L \).

If \( m^* = m \) then the iteration in CONstrained-MEC stops; otherwise the following pruning step is executed for the next iteration. Without loss of generality, let \( m^* \) be above \( m \). We again scan each \( \ell_i = (P[2i-1],P[2i]) \) and compute its intersection with \( L \). If it is below \( m \), then we delete the one which is on the same side of \( m \) with respect to the bisector line \( \ell_i \). As we have \( n \over 4 \) intersection points below \( m \), we can delete (i.e., move at the end of the array) \( n \over 4 \) points from \( P \). The case where \( m^* \) is below \( m \) can be handled similarly. The entire procedure CONstrained-MEC needs \( O(n) \) time and \( O(1) \) extra-space, but after an iteration Invariants 5.1 and 5.2 may not remain valid.

To resolve this problem, we do the following. During the execution of CONstrained-MEC, if a point is deleted from a tuple \((p,q,r,s)\) in an iteration, it is considered to be invalid from next iteration onwards. We partition the array \( P \) containing all the points into five blocks namely \( A, B, C, D \) and \( E \) and use four index variables \( i_1, i_2, i_3, \) and \( i_4 \) to mark the ending of the first four blocks (see Figure 5.1). Block \( A \) consists of those tuple \((p,q,r,s)\) whose four
Ch 5. Minimum Enclosing Circle with Limited Work-space

Figure 5.1: Block partition of the array $P$

points are invalid. The block $B$ signifies all those tuples containing three invalid points. Similarly, block $C$, $D$ contain tuples with two and one invalid point(s), respectively. Block $E$ contains all tuples with no invalid point. We further partition the block $C$ into two sub-blocks $C_1$ and $C_2$, respectively. The tuples with first two invalid points are kept in $C_1$ and the tuples with first and third invalid points are stored in $C_2$. If a tuple has invalid points in second (resp. fourth) position, then these are swapped to first (resp. third) position. We use an index variable $i_c$ to mark the partition between $C_1$ and $C_2$. All the invalid points in a tuple belonging to blocks $B$ and $D$ are kept at the beginning of that tuple. In other words, during the entire execution of CONSTRANDED-MEC, we maintain the following invariant along with the Invariants 5.1 and 5.2.

**Invariant 5.3** The tuples with zero, one, two, three and four valid point(s) will be in the blocks $A$, $B$, $C$, $D$ and $E$, respectively, as mentioned above.

Now, we need (i) to form the bisector lines $\{\ell_i, i = 1, 2, \ldots \left[\frac{n}{2}\right]\}$, and then (ii) to find the median $m$ of the points of intersection of these bisector lines with $L$ in an in-place manner using the algorithm given in [Carlsson 1995]. If we form these bisector lines with two consecutive valid points in the array $P$, then the Invariants 5.1, 5.2 may not be maintained since (i) during the median finding $\ell_i$’s need to be swapped, and (ii) the points in a tuple may contribute to different $\ell_i$’s.

Here, three important things need to be mentioned:

**Formation of $\ell_i$ (i.e. members of PAIR)**: Each tuple in block $B$ contains only one valid point. Thus, we pair up two tuples to form one bisector line $\ell_i$ in Step 1 of the algorithm CONSTRANDED-MEC. Thus, we will have $\left\lfloor\frac{1}{2}\left(\frac{i_2-i_1}{4}\right)\right\rfloor$ bisectors. Let’s denote these set of bisectors by $\mathcal{L}_1$. 
5.3. In-place implementation of MEC

Similarly, \( C_1 \) and \( C_2 \) will produce \( \frac{i_2 - i_1}{4} \) and \( \frac{i_3 - i_4}{4} \) bisector lines respectively, and these are denoted as \( L_2 \) and \( L_3 \) respectively.

In block \( D \), each tuple \((p, q, r, s)\) contains three \textit{valid} points and the \textit{invalid} point is \( p \). In each of these tuples, we consider the pair of points \((r, s)\) to form a bisector line. Let us denote this set of bisectors by \( L_4 \), and the number of bisectors in this set is \( \frac{i_4 - i_3}{4} \).

Next, we consider each disjoint pair of consecutive tuples \((p, q, r, s)\) and \((p', q', r', s')\) in block \( D \), and define a bisector line with the \textit{valid} point-pair \((q, q')\). Thus we get \( \lfloor \frac{1}{2} (\frac{i_4 - i_1}{4}) \rfloor \) such bisectors, and name this set as \( L_5 \).

From each tuple \((p, q, r, s)\) in block \( E \), we get two bisectors. Here, we form two sets of bisectors, namely \( L_6 \) and \( L_7 \). \( L_6 \) is formed with \((p, q)\) of each tuple in block \( E \), and \( L_7 \) is formed with \((r, s)\) of each tuple in block \( E \). Each of these sets contains \( \lfloor \frac{n - i}{4} \rfloor \) bisectors.

Thus, we have seven sets of bisectors, namely \( L_i, i = 1, 2, \ldots, 7 \).

**Computing Median:** We compute the median of the points of intersection of the lines in each set of bisector lines \( L_i, i = 1, 2, \ldots, 7 \) with \( L \) separately. We use \( m_i \) to denote the median for \( i \)-th set. During the execution of in-place median finding algorithm of [Carlsson 1995], if a pair of lines \( \ell_i, \ell_j \in L_k \) are swapped then the corresponding entire tuple(s) are swapped. Thus, the tuples are not broken for computing the median and all the Invariants 5.1, 5.2 and 5.3 are maintained.

**Pruning Step:** We take two variables \( m' \) and \( m'' \) to store two points on the line \( L \) such that the desired center \( m^* \) of the minimum enclosing circle of \( P \) on \( L \) satisfies \( m' \leq m^* \leq m'' \). We initialize \( m' = -\infty \) and \( m'' = \infty \).

Now, we consider each \( m_i, i = 1, 2, \ldots, 7 \) separately; if \( m^* \) is above \( m_i \) and \( m' < m_i \), then \( m' \) is set to \( m_i \). If \( m^* \) is below \( m_i \) and \( m'' > m_i \), then \( m'' \) is set to \( m_i \).

We now prune points by considering the intersection of the bisector lines in \( \bigcup_{i=1}^{7} L_i \) with the line \( L \). If a bisector line \( \ell = (p, q) \in \bigcup_{i=1}^{7} L_i \) intersects \( L \) in the interval \([m', m'']\), then none of \( p, q \) becomes \textit{invalid}; otherwise,
one of the points \( p \) or \( q \) becomes \textit{invalid} as mentioned in Step 4 of the Procedure \textsc{Constrained-MEC}.

While considering the bisector lines in \( \mathcal{L}_1 \), a tuple in the block \( B \) may be moved to block \( A \) by swapping that tuple with the first tuple of block \( B \) and incrementing \( i_1 \) by 4.

While considering a bisector line \( \ell \in \mathcal{L}_2 \cup \mathcal{L}_3 \), if any one of its participating points is deleted then the corresponding tuple is moved to block \( B \) by executing one or two swaps of tuple and incrementing \( i_2 \) by 4.

Note that the bisector lines in \( \mathcal{L}_4 \) and \( \mathcal{L}_5 \) are to be considered simultaneously. For a pair of consecutive tuples \((p, q, r, s), (p', q', r', s') \in D\), we test the bisector lines \( \ell = (q, q') \in \mathcal{L}_5 \) and \( \ell' = (r, s) \in \mathcal{L}_4 \) and \( \ell'' = (r', s') \in \mathcal{L}_4 \) with \([m', m'']\). This may cause deletion of one or two points from \((p, q, r, s)\) (resp. \((p', q', r', s')\)). For the tuple \((p, q, r, s)\),

- if none of the points \( q, r, s \) becomes \textit{invalid}, then the tuple \((p, q, r, s)\) will remain in the set \( D \);
- if only \( q \) becomes invalid, then the tuple \((p, q, r, s)\) is moved to \( C_1 \) by two swaps of tuples; necessary adjustments of \( i_c \) and \( i_3 \) need to be done;
- if \( r \) or \( s \) only becomes \textit{invalid}, then the tuple \((p, q, r, s)\) is moved to \( C_2 \) (with a swap of \( r \) and \( s \) if necessary), and adjustment of \( i_3 \) is done;
- if \( q \) and \( r \) both become \textit{invalid}, then the tuple \((p, q, r, s)\) is moved to \( B \) with necessary adjustment of \( i_2, i_3 \);
- if \( q \) and \( s \) both become \textit{invalid}, then the tuple \((p, q, r, s)\) is moved to \( B \) (with swap among \( r \) and \( s \)) and necessary adjustment of \( i_2, i_3 \) need to be done.

The same set of actions may be necessary for the tuple \((p', q', r', s')\) also.

Similarly, the bisector lines in \( \mathcal{L}_6 \) and \( \mathcal{L}_7 \) are considered simultaneously. For a tuple \((p, q, r, s) \in E\), \( \ell = (p, q) \in \mathcal{L}_6 \) and \( \ell' = (r, s) \in \mathcal{L}_7 \). Here, none or one or two points from the tuple \((p, q, r, s)\) may be deleted. Depending on that, it may reside in the same block or may be moved to
block $D$ or $C_2$. The necessary intra-block movements can be done with one or two tuple-swap operations. Surely at most two swap operations inside the tuple may be required to satisfy Invariant 5.3.

Correctness and complexity results

**Lemma 5.1** The above Pruning Step ensures that Invariants 5.1, 5.2 and 5.3 are maintained, and at least $\frac{2}{3}$ points become invalid after each iteration of the while-loop of CONSTRAINED-MEC, where $n'$ is the number of valid points in $P$ at the beginning of that iteration.

**Proof:** The description of the Pruning Step justifies the first part of the lemma. For the second part, note that $m_i$ (the median of the intersection points of the members in $L_i$ with the line $L$) satisfies either $m_i \leq m'$ or $m_i \geq m''$. In both the cases, at least half of the lines in $L_i$ intersect the line $L$ outside the interval $[m', m'']$. Thus, the result follows.

The correctness of the algorithm follows from the fact that after an iteration of the while-loop of the CONSTRAINED-MEC, the valid points can be easily identified using our proposed scheme of maintaining the points in five different blocks as mentioned in Invariant 5.3. It also helps in forming the bisector lines, and pruning of points maintaining Invariants 5.1 and 5.2. The second part of Lemma 5.1 justifies the following result.

**Lemma 5.2** The CONSTRAINED-MEC can be computed in an in-place manner in $O(n)$ time with constant amount of extra-space maintaining Invariants 5.1 and 5.2.

### 5.4 When the memory is read-only

In this section, we show how one can compute the minimum enclosing circle efficiently for a set of points in $\mathbb{R}^2$ using few extra variables, when the input points are given in a read-only array $P$. Here, again we will use the basic algorithm $MEC$ of Megiddo as described in Section 5.2 which has the standard prune-and-search scheme as stated in Algorithm 4.1. According to
Theorem 4.2, this algorithm is implementable in read-only memory, provided the procedure \textsc{Intermediate-Computation-for-MEC} is implementable in the \textit{pairing scheme.}

Median finding (or Selection) and \textsc{Constrained-MEC} are the main steps in \textsc{Intermediate-Computation-for-MEC}. We already described how much time and space are required to find the median using the pairing scheme in Section 4.5. First, we show how one can compute \textsc{Constrained-MEC} using few extra variables when the input array is read-only. Next, we will show how to compute the MEC using this.

\subsection{\textsc{Constrained-MEC} in read-only setup}

Note that \textsc{Constrained-MEC} (given in Algorithm 5.3) also has the standard prune-and-search scheme stated in Algorithm 4.1.

At each iteration of the procedure \textsc{Constrained-MEC}, at least $\frac{1}{4}|P|$ points in $P$ are pruned (marked \textit{invalid}). Thus, the total number of iterations of the while-loop in the procedure \textsc{Constrained-MEC} is at most $K = O(\log |P|)$.

We use an array $M$ each element of which can store a real number, and an array $D$ each element of which is a bit. Both the arrays are of size $O(\log |P|)$.

After each iteration of the read-only algorithm, it needs to remember the median $m$ among the points of intersection of the bisector lines on the line $L$, and the direction in which we need to proceed from $m$ to reach the constrained center $m^*$. So, after executing the $t$-th iteration, we store $m$ at $M[t]$; $D[t]$ will contain 0 or 1 depending on whether $m^* > m$ or $m^* < m$. Note that $M[t]$ and $D[t]$ act as the SPC$_t$, where $t \in \{1, 2, \ldots, K\}$ (see Section 4.4).

We now explain the $t$-th iteration assuming that $(t-1)$ iterations are over. Here, we need to pair-up points in $P$ in such a way that all the \textit{invalid} elements up to the $(t-1)$-th iteration can be ignored correctly. We use one more array $\text{IndexP}$ of size $\log |P|$. At the beginning of each iteration of the while-loop of this procedure, all the elements in this array are initialized with $-1$. Note that the array $\text{IndexP}$ works as the array $\text{STATUS}$ (see Section 4.4).

Note that we have no space to store the mark bit for the \textit{invalid} points in the array $P$. We use the \textit{compute in lieu of store} paradigm, or in other words, we
5.4. When the memory is read-only

check whether a point is valid at the $t$-th iteration, by testing its validity in all the $i = 1, 2, \ldots, t - 1$ levels (previous iterations).

We start scanning the input array $P$ from the left, and identify the points that are tested as valid in the $i$-th level for all $i = 1, 2, \ldots, t - 1$. As in the in-place version of the CONSTRINED-MEC algorithm, here also we pair up these valid points for computing the bisector lines. Here, we notice the following fact:

Suppose in the $(t - 1)$-th iteration $(p, q)$ form a pair, and at the end of this iteration $p$ is observed as invalid. While executing the $t$-th iteration, we again need to check whether $p$ was valid in the $(t - 1)$-th iteration since it was not marked. Now, during this checking if we use a different point $q'$ ($\neq q$) to form a pair with $p$, it may be observed valid. So, during the checking in the $t$-th iteration, $(p, q)$ should be paired up for checking at the $(t - 1)$-th level.

Thus, our pairing scheme for points should be such that it must satisfy the following invariant.

**Invariant 5.4** If (i) two points $p, q \in P$ form a point-pair at the $i$-th level in the $t_1$-th iteration ($i < t_1$), and (ii) both of them remain valid up to $t_2$-th iteration where $t_2 > t_1$, then $p, q$ will also form a point-pair at the $i$-th level of the $t_2$-th iteration.

**Pairing Scheme:** We consider the point-pairs $(P[2\alpha - 1], P[2\alpha])$, $\alpha = 1, 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$ in order. For each pair, we compute their bisector $\ell_\alpha$, and perform the level 1 test using $M[1]$ and $D[1]$ to see whether both of them remain valid at iteration 1. In other words, we observe where the line $\ell_\alpha$ intersects the vertical line $x = M[1]$, and then use $D[1]$ to check whether any one of the points $P[2\alpha - 1]$ and $P[2\alpha]$ becomes invalid or both of them remain valid.

If the test succeeds, we perform level 2 test for $\ell_\alpha$ by using $M[2]$ and $D[2]$. We proceed similarly until (i) we reach up to $t$-th level and both the points remain valid at all the levels, or (ii) one of these points is marked invalid at some level, say $j$ ($< t - 1$). In Case (i), the point pair $(P[2\alpha - 1], P[2\alpha])$ participates in computing the median value $m_t$. In case (ii), we state the course
of action assuming \(P[2\alpha]\) remains valid and \(P[2\alpha - 1]\) becomes invalid\(^1\). Here, two situations need to be considered depending on the value of \(IndexP[j]\). If \(IndexP[j] = -1\) (no point is stored in \(IndexP[j]\)), we store \(2\alpha\) in \(IndexP[j]\). If \(IndexP[j] = \beta(\neq -1)\) (index of a valid point), we form a pair \((P[2\alpha], P[\beta])\) and proceed to check starting from \(j + 1\)-th level (i.e., using \(M[j + 1]\) and \(D[j + 1]\)) onwards until it reaches the \(t\)-th level or one of them is marked invalid in some level between \(j\) and \(t\). Both the situations are handled in a manner similar to Cases (i) and (ii) as stated above.

**Lemma 5.3** Invariant 5.4 is maintained throughout the execution.

**Proof:** Follows from the fact that the tests for the points in \(P\) at different levels \(i = 1, 2, \ldots, t - 2\) at both the \((t - 1)\)-th and \(t\)-th iterations are same. At the \((t - 1)\)-th level of the \((t - 1)\)-th iteration, we compute \(M[t - 1]\) and \(D[t - 1]\) with the valid points. At the \((t - 1)\)-th level of the \(t\)-th iteration, we prune points that were tested valid at the \((t - 1)\)-th iteration using \(M[t - 1]\) and \(D[t - 1]\). \(\square\)

**Observation 5.1** At the end of the \(t\)-th iteration,

(i) Some cells of the \(IndexP\) array may contain valid indices \((\neq -1)\).

(ii) In particular, \(IndexP[t - 1]\) will either contain \(-1\) or it will contain the index of some point \(\beta\) in \(P\) that has participated in computing \(M[t - 1]\) (i.e., remained valid up to level \(t - 1\)).

(iii) If in this iteration \(IndexP[t - 1] = \beta\) (where \(\beta\) may be a valid index or \(-1\)), then at the end of all subsequent iterations \(j (> t)\) it will be observed that \(IndexP[t - 1] = \beta\).

**Proof:** Part (i) follows from the pairing scheme. Parts (ii) & (iii) follow from Lemma 5.3. \(\square\)

\(^1\)Similar action is taken if \(P[2\alpha - 1]\) remains valid and \(P[2\alpha]\) becomes invalid.
Lemma 5.4 In the t-th iteration, the time complexity for enumerating all valid points is $O(nt)$.

Proof: Follows from the fact that each valid point in the t-th iteration has to qualify as a valid point in the tests of all the $t-1$ levels. For any other point the number of tests is at most $t-2$. \hfill \Box

The main task in the t-th iteration is to find the median of the points of intersection of all the valid pairs in that iteration with the given line $L$. In Chapter 4.5, we have shown how to compute the median in this pairing scheme. So combining with the Lemma 4.4, we have the following result:

Lemma 5.5 In the read-only environment, the t-th iteration of the while-loop of CONSTRAINED-MEC can be computed in (i) $O(tn^{1+\frac{1}{k+1}} \log^k n)$ time using $O(tk)$ extra-space, where $k$ is a fixed natural number, or (ii) $O(tn \log^2 n)$ time using $O(t \log n)$ extra-space.

At the end of the $O(\log n)$ iterations, we could discard all the points except at most $|IndexP| + 3$ points, where $|IndexP|$ is the number of cells in the array $IndexP$ that contain valid indices of $P$ ($\neq -1$). This can be at most $O(\log n)$ in number. We can further prune the points in the $IndexP$ array using the in-place algorithm for CONSTRAINED-MEC proposed in Section 5.3.1. Thus, we have the following result:

Lemma 5.6 In the read-only environment, the procedure CONSTRAINED-MEC can be computed in (i) $O(n^{1+\frac{1}{k+1}} \log^{k+2} n)$ time using $O(k \log n)$ extra-space, where $k$ is a fixed natural number, or (ii) $O(n \log^4 n)$ time using $O(\log^2 n)$ extra-space.

Proof: Follows from the fact that total number of iterations of the while-loop is $O(\log n)$. \hfill \Box
5.4.2 (Unconstrained) MEC in the read-only setup

We use the read-only variation of the CONSTRAINED-MEC algorithm (described in Subsection 5.4.1) for solving the minimum enclosing circle problem. Here, we need to maintain three more arrays $M, D$ and $I$, each of size $O(\log n)$. $M[t]$ contains two mutually perpendicular lines and $D[t]$ (a two bit space) indicates the quadrant in which the center $\pi^*$ of the unconstrained MEC lies in the $t$-th iteration of the while-loop of the algorithm MEC. So, the arrays $M, D$ act as the SPC and the array $I$ plays the role of STATUS as described in Chapter 4.4.

Total number of iteration of the while-loop of MEC is $K_1 = O(\log n)$. We refer this while-loop as outer-loop. In each iteration of this outer-loop, CONSTRAINED-MEC and median finding is evoked in the INTERMEDIATE-COMPUTATION-for-MEC. The time and space complexities of median finding is stated in Lemma 4.4.

So, consider the procedure CONSTRAINED-MEC. Each CONSTRAINED-MEC consists of $K_2 = O(\log n)$ iteration of its while-loop. We refer this while-loop as inner-loop. So, the total number of iterations of this inner-loop for MEC is $O(\log^2 n)$. From Lemma 5.5, one can see the following:

**Lemma 5.7** In the $t_1$-th ($1 \leq t_1 \leq K_1$) iteration of the outer-loop, the $t_2$-th ($1 \leq t_2 \leq K_2$) iteration of the inner-loop takes (i) $O((t_1 + t_2)n^{1+\frac{1}{k+1}} \log^k n)$ time using $O((t_1 + t_2)k)$ extra-space, where $k$ is a fixed natural number, or (ii) $O((t_1 + t_2)n \log^2 n)$ time using $O((t_1 + t_2) \log n)$ extra-space.

**Proof:** In $t_1$-th iteration of the outer-loop, the enumeration of all the valid points need $O(nt_1)$ time as it requires to read the arrays $M, D$ and $I$ up to first $t_1$-th location. On the other hand, in the $t_2$-th iteration of the inner-loop, by reading the arrays $M, D$ and $I$ up to $t_2$-th location, CONSTRAINED-MEC needs another $O(nt_2)$ time to recognize its own valid points from those which remain valid up to $t_1$-th iteration of the outer-loop. So, in the $t_1$-th iteration of the outer-loop, the $t_2$-th iteration of the inner-loop, the time complexity of enumerating all valid points for the inner-loop of CONSTRAINED-MEC takes $O(n(t_1 + t_2))$. Thus, similar to Lemma 5.5, the result follows. \qed
Hence, we have the following result:

**Theorem 5.2** The minimum enclosing circle of a set of \( n \) points in \( \mathbb{R}^2 \) given in a read-only array can be computed in (i) \( O(n^{1+rac{1}{k+1}} \log^{k+3} n) \) time using \( O(k \log n) \) extra-space, where \( k \) is a fixed natural number, or (ii) \( O(n \log^5 n) \) time and \( O(\log^2 n) \) extra-space.

## 5.5 Conclusion

Asano et al. [Asano 2011b] proposed an open problem of computing the minimum enclosing circle of a set of \( n \) points in \( \mathbb{R}^2 \) given in a read-only array in sub-quadratic time. In this chapter, we give an answer to that open question. Our in-place MEC as well as read-only MEC algorithm significantly improve the previously known best results. It will be worthy to study whether one can further improve the \( \text{time} \times \text{extra-space} \) complexity of MEC in a read-only setting.
6.1 Introduction

Convex hull is one of the most important structures in computational geometry. Convex hull for a set $P$ of points in $\mathbb{R}^2$ is defined as the smallest convex set that contains $P$.

Given a set of $n$ real numbers $N = \{x_1, x_2, \ldots, x_n\}$, one can map them to $n$ points $P = \{(x_1, x_1^2), (x_2, x_2^2), \ldots (x_n, x_n^2)\}$ in $\mathbb{R}^2$. Now a convex-hull ordering for the points in $P$ implies a sorted sequence for the real numbers of $N$. This is a folklore fact [Preparata 1990]. Borodin et al. [Borodin 1981] showed that the lower bound of the time-space product for the sorting is $\Omega(n^2)$. Thus, the time-space product lower bound for convex hull is also $\Omega(n^2)$.

Here, we consider a constrained version of the problem where the points in the read-only array $P$ are sorted with respect to their $x$-coordinates. Proceeding
in the similar way as described in Section 5.4.1 (for computing minimum enclosing circle), we will show how Kirkpatrick and Seidel's [Kirkpatrick 1986] deterministic prune-and-search algorithm for computing convex hull can be implemented in this framework.

### 6.2 Overview of Kirkpatrick and Seidel’s algorithm

The algorithm in [Kirkpatrick 1986] computes upper-hull and lower-hull separately and reports them. For self-completeness, we briefly describe this algorithm. The basic steps of computing upper-hull of a set of points \( P \) is given in Algorithm 6.1. Lower-hull can be computed in a similar way. Algorithm 6.1 follows divide-and-conquer paradigm. It uses a procedure \textsc{Compute-Bridge} to compute the bridge between two disjoint subsets of \( P \) using the prune-and-search technique. The details of this procedure is described in Algorithm 6.2.

**Algorithm 6.1: KS-Upper-Hull\(^{(*)} \) (\(^{(*)} \text{KS stands for Kirkpatrick and Seidel} \))**

**Input:** A set of points \( P \) in \( \mathbb{R}^2 \) sorted according to \( x \)-coordinates

**Output:** The upper-hull of \( P \)

1. (**Uses divide-and-conquer technique \(^{(*)} \)**)
2. **STEP 1:** Find the point \( p_m \in P \) having median \( x \)-coordinate;
3. **STEP 2:** Partition \( P \) into two subsets \( P_L \) and \( P_R \) where \( P_L \) contains all the points in \( P \) whose \( x \)-coordinate is less than or equal to \( x(p_m) \) and \( P_R = P \setminus P_L \);
4. **STEP 3:** \((a,b) = \text{Compute-Bridge}(P_L, P_R)\); (**This procedure computes the bridge between \( P_L \) and \( P_R \); \( a \in P_L, b \in P_R \)**)
5. **STEP 4:** Report \((a,b)\);
6. **STEP 5:** Compute \( P_L' = P_L \setminus P_L' \) where \( P_L' = \{ p \in P_L | x(p) > x(a) \} \), and \( P_R' = \{ p \in P_R | x(p) < x(b) \} \);
7. **STEP 6:** KS-Upper-Hull\(^{(*)} \)(\( P_L' \));
8. **STEP 7:** KS-Upper-Hull\(^{(*)} \)(\( P_R' \));

The straight-forward implementation of the algorithm KS-Upper-Hull when input is in a read-only memory, requires \( O(n) \) space for the procedure \textsc{Compute-Bridge} as it needs to remember which points were pruned in the previous iterations. In addition, the algorithm KS-Upper-Hull\(^{(*)} \)(\( P \)) reports the hull-
6.2. Overview of Kirkpatrick and Seidel’s algorithm

Algorithm 6.2: Compute-Bridge$(P_ℓ, P_r)$

**Input:** Two sets of points in $\mathbb{R}^2$, $P_ℓ$ and $P_r$ sorted according to $x$-coordinates

**Output:** The bridge between $P_ℓ$ and $P_r$

(* Uses prune-and-search technique *)

1. **STEP 1:**
   2. while $|P_ℓ| > 1$ and $|P_r| > 1$
      3. **STEP 1.1:** Arbitrarily pair-up points in $P_ℓ \cup P_r$; Let $L$ be the set of these pairs. Each such pair of points $(p,q) \in L$ will signify a line $pq$ which will pass through $p,q$. We denote the slope of the line $pq$ as $\alpha(pq)$.
      4. (** Intermediate-Computation Starts **) 
         5. **STEP 1.2:** Consider the slopes of these $|P_ℓ \cup P_r|$ lines and compute their median. Let $\alpha_m$ be the median slope.
         6. **STEP 1.3:** Compute the supporting line of $P_ℓ$ and $P_r$ with slope $\alpha_m$; Suppose these are at points $a(\in P_ℓ)$ and $b(\in P_r)$ respectively.
         7. **STEP 1.4:** Now compare $\alpha(ab)$ with $\alpha_m$. Here, one of the three cases may arise: (i) $\alpha(ab) = \alpha_m$, (ii) $\alpha(ab) < \alpha_m$ or (iii) $\alpha(ab) > \alpha_m$.
      8. if $\alpha(ab) = \alpha_m$ then
         9. $ab$ is the required bridge between the points in $P_ℓ$ and $P_r$. So, the procedure returns $(a, b)$.
      10. (** Intermediate-Computation Ends **) 
      11. **STEP 1.5:** (**Pruning Step**) 
      12. Decide whether $\alpha(ab) < \alpha_m$ or $\alpha(ab) > \alpha_m$. Without loss of generality, assume that $\alpha(ab) < \alpha_m$, then we will consider all the pairs $(p,q) \in L$ whose $\alpha(pq) \geq \alpha_m$. We can ignore one point among $(p,q)$ which is to the left of the other one. So, at least $\frac{|P_ℓ \cup P_r|}{4}$ points are ignored for further consideration.
      13. **STEP 2:**
      14. Find the bridge in brute-force manner and return the bridge.
edges in an arbitrary fashion (not in order along the boundary of the convex hull) and takes $O(\log n)$ extra-space for the recursions. In the next sections, we describe how to (i) report the hull edges in order, and (ii) implement the procedure \textsc{Compute-Bridge} using limited work-space in the read-only environment.

### 6.3 Reporting the hull-edges in-order

Now, we will show how to report the hull vertices in clockwise order using no more than $O(\log n)$ extra-space. Consider the recursion tree $T$ of the algorithm \textsc{KS-Upper-Hull}. Its each node represents the reporting of a hull-edge. In the algorithm \textsc{KS-Upper-Hull}, as the reporting is done according to pre-order traversal of the tree $T$, the hull edges are not reported in clockwise order. In order to report them in clockwise order, we need to traverse the recursion tree in in-order manner, i.e. STEP 3 and STEP 4 of the Algorithm 6.1 should be in between STEP 5 and STEP 6. But, if we do this, then we can not evoke \textsc{KS-Upper-Hull}(P_{\ell}) on the updated point-set $P_{\ell}$. To resolve this problem, we will compute the bridge in STEP 3 itself but will not report it then. We push it in the stack in STEP 3 and pop it from stack in between STEP 5 and STEP 6. The size of this stack depends on the depth of the recursion tree which is $O(\log h)$, where $h$ is the number of hull-edges. The details of this change is given in the procedure \textsc{Read-Only-Upper-Hull(start, end)} (Algorithm 6.3).

Thus we have the following result:

**Lemma 6.1** Given a set of $n$ sorted points $P$ in a read-only array, reporting of the hull edges can be done in clockwise order using only $O(\log n)$ extra-space (assuming that the procedure \textsc{Compute-Bridge} takes no more than $O(\log n)$ extra-space).

### 6.4 \textsc{Compute-Bridge} with limited work-space

The set of input points $P$ is partitioned into two parts $P_{\ell}$ and $P_r$ by choosing the point $P[m]$ having median $x$-coordinate. Since the array is sorted
with respect to the $x$-coordinates, this needs $O(1)$ time. Now the procedure \textsc{Compute-Bridge} is executed to compute the bridge of $P_L$ and $P_R$.

Note that \textsc{Compute-Bridge} given in Algorithm 6.2 has the standard prune- and-search scheme stated in Algorithm 4.1. So, we have to show that the \textsc{Intermediate-Computation}, i.e. Step 1.2, 1.3 and 1.4 of \textsc{Compute-Bridge} can be computed in the read-only environment using the pairing scheme.

In each iteration of the while-loop of the procedure \textsc{Compute-Bridge}, $\frac{1}{4}$th of the points from the set $P_L \cup P_R$ are pruned. The points which are pruned in $t$th iteration are not considered in any $j$th iteration, where $j > t$. After an iteration of the while-loop, either the bridge is returned or the iteration continues until $|P_L| = 1$ or $|P_R| = 1$. So, the number of iterations of the while loop is at most $O(\log n)$, where $n$ is the total number of points in $P$.

While executing the $t$th iteration, we want to remember the points which were pruned in the previous $t-1$ iterations, $t \in \{1, \ldots, \log n\}$. As in Chapter 5.4.1, here also we take an array $M$ of size $O(\log n)$ and another bit-array $B$ of size $O(\log n)$. At each $t$th iteration, ignoring all the pruned points, we pair the valid points and consider the slopes of all the lines defined by the paired points. We compute the median slope value $\mu_t$ of these lines and store it at $M[t]$. If the supporting lines are at points $a$ and $b$ and the slope $\alpha(\overline{ab})$ is greater (resp. less) than $\mu_t$, then we set $B[t] = 1$ (resp. 0); $\alpha(\overline{ab}) = \mu_t$ implies
that we already got the bridge. Thus, $B[t]$ signifies whether $M[t]$ is greater than or less than the slope $\alpha^*$ of the desired bridge.

Now, we will describe the pairing scheme (exactly similar to the pairing scheme stated in Chapter 4.4) which will satisfy the following invariants:

**Invariant 6.1**

(i) If a point $p$ is pruned at some iteration $t$, then it will not participate to form a pair in any $j$-th iteration, where $j > t$.

(ii) If $(p, q)$ are paired at the $t$-th iteration of the while-loop, none of the points $p, q$ is pruned at the end of this iteration and we need to go for $t + 1$-th iteration, then $(p, q)$ will again form a valid pair at $t + 1$-th iteration.

(iii) If $(p, q)$ is a valid pair at $t$-th iteration and $(p, s)$ ($s \neq q$) are paired at $t + 1$-th iteration of the while-loop, then there exist some $r$ such that $(r, s)$ were paired at $t$-th iteration, and $q$ and $r$ were pruned at the end of $t$-th iteration.

The iteration starts with the points $\{P[start], P[start + 1], \ldots, P[end]\}$. In the first iteration of the while-loop, we consider the consecutive points, i.e, $(P[start], P[start + 1]), (P[start + 2], P[start + 3]), \ldots$ as valid pairs. Assume that first $t - 1$ iterations of the while-loop are over, and we are at the beginning of the $t$-th iteration; $M[j]$ contains median slope of $j$-th iteration and $B[j]$ contains 0 or 1 depending on $M[j] > \alpha^*$ or $M[j] < \alpha^*$ for all $1 \leq j \leq t - 1$.

Now, we want to detect all the valid points and pair them up maintaining the Invariant 6.1. As in Chapter 5.4.1, here also, we use another array $IndexP$ of size $O(\log n)$ initializing all its elements with -1.

We consider the point-pairs $(P[start + 2\nu], P[start + 2\nu + 1]), \nu = 0, 1, \ldots, \left\lfloor \frac{end-start+1}{2} \right\rfloor$ in order. For each pair, we compute the slope $\gamma = \alpha(P[start + 2\nu], P[start + 2\nu + 1])$ of the corresponding line, and perform the level 1 test using $M[1]$ and $B[1]$ to see whether both of them remain valid at iteration 1. If the test succeeds, we perform level 2 test by using $M[2]$ and $B[2]$. We proceed similarly until (i) we reach up to $t - 1$-th level and both the points remain valid at all the levels, or (ii) one of these points becomes invalid at some level, say $j$ ($< t - 1$). In Case (i), the point pair $(P[start + 2\nu], P[start + 2\nu + 1])$
forms a valid pair and participates in computing the median value \( m_t \). In case (ii), suppose \( P[start + 2 \nu] \) remains valid and \( P[start + 2 \nu + 1] \) becomes invalid at level \( j \). Here, two situations need to be considered depending on the value of \( \text{Index}P[j] \). If \( \text{Index}P[j] = -1 \) (no point is stored in \( \text{Index}P[j] \)), we store \( (start + 2 \nu) \) in \( \text{Index}P[j] \). If \( \text{Index}P[j] = \beta (\neq -1) \) (index of a valid point), we form a pair \( (P[start + 2 \nu], P[\beta]) \) and proceed to check starting from \( j + 1 \)-th level (i.e., using \( M[j + 1] \) and \( B[j + 1] \)) onwards until it reaches the \( t \)-th level or one of them is marked invalid in some level between \( j \) and \( t \). Both the situations are handled in a manner similar to Cases (i) and (ii) as stated above. Thus, each valid point in the \( t \)-th iteration has to qualify as a valid point in the tests of all the \( t - 1 \) levels. For any other point the number of tests is at most \( t - 2 \). Thus the amortized time complexity of enumerating all valid pairs in the \( t \)-th iteration is \( O(nt) \).

The main task in INTERMEDIATE-COMPUTATION is to find the median of the slope of lines corresponding to valid pair of points. In Lemma 4.4, we show how to compute median in this pairing scheme and how much time and space is required for that. At the end of \( O(\log n) \) iterations, we could discard all the points except at most \( O(\log n) \) valid points (available in cells of the array \( \text{Index}P \) having value \( \neq -1 \)). We can further prune the points in the \( \text{Index}P \) array using the in-place algorithm for COMPUTE-BRIDGE described in [Vahrenholz 2012]. Thus, we have the following result:

**Lemma 6.2** The procedure COMPUTE-BRIDGE can be correctly computed in the read-only environment in (i) \( O(n^{1+\frac{1}{k+1}} \log^{k+2} n) \) time with \( O(k \log n) \) extra-space, where \( k \) is a fixed natural number, or (ii) \( O(n \log^4 n) \) time with \( O(\log^2 n) \) extra-space.

**Proof:** The correctness of this read-only version of the procedure COMPUTE-BRIDGE follows from the fact that the Invariant 6.1 is correctly maintained. Considering the fact that we need to perform \( \log n \) iterations to compute a bridge, and time complexity of the \( t \)-th iteration from Lemma 4.4, the time complexity can be justified by adding the worst case time requirements for all the iterations. Note that the time required by the in-place algorithm for further pruning the entries in the array \( \text{Index}P \) is \( O(\log n) \). The space
complexity obviously follows from the space requirement of median finding in Lemma 4.4.

\[ \square \]

6.5 Correctness and complexity

**Theorem 6.1** Given a set of \( n \) sorted points \( P \) in \( \mathbb{R}^2 \) in a read-only array, the convex-hull of \( P \) can be computed in (i) \( O(n^{1+\frac{1}{k+1}} \log^{k+3} n) \) time with \( O(k \log n) \) extra-space, where \( k \) is a fixed natural number, or (ii) \( O(n \log^5 n) \) time with \( O(\log^2 n) \) extra-space.

**Proof:** The correctness of the algorithm \textsc{Read-Only-Upper-Hull} follows from the correctness of Kirkpatrick and Siedel's algorithm [Kirkpatrick 1986], as we are following the basic structure of this.

The procedure \textsc{Compute-Bridge} is evoked \( h \) times, where \( h \) is the number of hull-edges. Consider the recursion tree of the algorithm \textsc{Read-Only-Upper-Hull}. Note that the depth of this tree is \( O(\log n) \) (more specifically, \( \log h \)). As there are at most \( \log n \) levels, so the total time complexity of the algorithm \textsc{Read-Only-Upper-Hull} is \( \log n \) times the total time complexity of each level. Again the total time complexity of a level is the sum of time complexities of all the subproblems solved in that level, while inputs are disjoint subsets of \( P \). Thus the time complexity follows.

For the recursion of \textsc{Read-Only-Upper-Hull} we need \( O(\log n) \) space. At each node of the recursion tree we need the space as stated in Lemma 6.2. However, we can re-use the same space for different nodes of the recursion tree while computing the bridge. Hence the total space complexity of the algorithm is justified.

\[ \square \]

6.6 Conclusion

In this chapter, we consider a restricted variation of the convex hull of a 2D point set where the points are sorted with respect to their \( x \)-coordinates. This is an important geometric optimization problem. We show how to solve the
problem efficiently, when the input is read-only and only $\text{poly}(\log n)$ amount of extra-space is given.
Chapter 7

Low Dimensional Linear Programming

Contents

7.1 Introduction .................................................. 101
7.2 Linear programming in \( \mathbb{R}^2 \) .......................... 102
  7.2.1 Read-only linear programming algorithm in \( \mathbb{R}^2 \) ... 102
7.3 Linear programming in \( \mathbb{R}^3 \) .......................... 104
  7.3.1 Testing-Line .............................................. 106
  7.3.2 Read-only linear programming in \( \mathbb{R}^3 \) ................ 108
7.4 Conclusion .................................................. 108

7.1 Introduction

Linear programming is one of the most important tools in solving a large number of optimization problems. In this chapter, we consider the problem of solving the linear programming problem in a read-only setup, i.e., the constraints are given in a memory where swapping of elements or modifying any entry is not permissible. Megiddo proposed linear time prune-and-search algorithms for the problem in both \( \mathbb{R}^2 \) and \( \mathbb{R}^3 \) using linear extra-space [Megiddo 1983b]. Proceeding in the similar way as described in Section 5.4.1, we will show that Megiddo’s algorithms for linear programming in both \( \mathbb{R}^2 \) and \( \mathbb{R}^3 \) can be implemented when the constraints are stored in a read-only array using limited work-space.
7.2 Linear programming in $\mathbb{R}^2$

Here the optimization problem is as follows:

$$\min_{x_1, x_2} c_1 x_1 + c_2 x_2$$

subject to:

$$a'_i x_1 + b'_i x_2 \geq \beta_i, \ i \in I = \{1, 2, \ldots, n\}.$$  

For ease of designing a linear time algorithm, Megiddo transformed it to an equivalent form as stated below:

$$\min_{x, y} y$$

subject to:

$$y \geq a_i x + b_i, \ i \in I_1,$$

$$y \leq a_i x + b_i, \ i \in I_2,$$

$$|I_1| + |I_2| \leq n.$$  

The detailed steps of Megiddo’s-2D-LP algorithm is given as $\text{MEGIDDO'S-2D-LP}(I, c_1, c_2)$ in Algorithm 7.1. The justification of all the steps are available in [Megiddo 1983b].

It maintains an interval $[a, b]$ of feasible values of $x$ (i.e., $a \leq x \leq b$). At the beginning of the algorithm, $a = -\infty$ and $b = \infty$. After each iteration of the algorithm, either it finds out that at some $x = x_m$ ($a \leq x_m \leq b$) the optimal solution exists (so the algorithm stops) or the interval $[a, b]$ is redefined (the new interval is either $[a, x_m]$ or $[x_m, b]$) and at least $\frac{n}{4}$ constraints are pruned for the next iteration. So, total number of iteration of the while-loop is $K = O(\log n)$.

7.2.1 Read-only linear programming algorithm in $\mathbb{R}^2$

We will give step by step description of implementing $\text{MEGIDDO'S-2D-LP}$ in a read-only setup.

The straight-forward conversion of one form into another mentioned in STEP 1 would take $O(n)$ extra-space. Note that remembering only the objective function $y = c_1 x_1 + c_2 x_2$, will enable one to reformulate the newer version of the constraints on-the-fly substituting $x_2$ in terms of $x_1$ and $y$ (replacing $x_1$ by $x$) in each constraint. So, we need not to worry about storing this new form.
Algorithm 7.1: MEGIDDO’S-2D-LP($I$, $c_1$, $c_2$)

Input: A set of $n$ constraints $a_i x_1 + b_i x_2 \geq \beta_i$, for $i \in I = \{1, 2, \ldots, n\}$

Output: The value of $x_1, x_2$ which minimizes $c_1 x_1 + c_2 x_2$

(* Uses standard prune-and-search scheme *)

STEP 1: Convert the form into the following:

\[ \min y, \text{ subject to } x, y \]

STEP 2: The problem can be solved directly.

STEP 2.1: Arbitrarily pair-up the constraints of $I_1$ (resp. $I_2$). Let $M_1$ (resp. $M_2$) be the set of aforesaid pairs, where $|M_1| = \frac{|I_1|}{2}$ and $|M_2| = \frac{|I_2|}{2}$.

Each pair of constraints in $M_1 \cup M_2$ are denoted by $(i, j)$ where $i$ and $j$ indicate the $i$-th and $j$-th constraints.

STEP 2.2: For each pair $(i, j) \in M_1 \cup M_2$ do

- if $a_i \neq a_j$ then Compute $x_{ij} = \frac{b_i - b_j}{a_i - a_j}$

Find the median $x_m$ among all $x_{ij}$’s which are in the interval $[a, b]$;

STEP 2.3: Test whether optimum $x^*$ satisfies $x^* = x_m$ or $x^* > x_m$ or $x^* < x_m$ as follows:

STEP 2.3.1: Compute $g = \max_{i \in I_1} a_i x_m + b_i$; $h = \min_{i \in I_2} a_i x_m + b_i$;

STEP 2.3.2: Compute

\[ s_g = \min \{ a_i | i \in I_1, a_i x_m + b_i = g \}; \quad S_g = \max \{ a_i | i \in I_1, a_i x_m + b_i = g \}; \]

\[ s_h = \min \{ a_i | i \in I_2, a_i x_m + b_i = h \}; \quad S_h = \max \{ a_i | i \in I_2, a_i x_m + b_i = h \}; \]

STEP 2.3.3: (* $g \leq h \Rightarrow x_m$ is feasible ; $g > h \Rightarrow x_m$ in infeasible region *)

if $g > h$ then

- if $s_g > S_h$ then (* $x_m < x^*$ *) $b = x_m$
- if $S_g < s_h$ then (* $x_m > x^*$ *) $a = x_m$
- else Report there is no feasible solution of the LP problem; Exit

else

- if $s_g > 0$ & $s_g \geq S_h$ then (* $x_m < x^*$ *) $b = x_m$
- if $S_g < 0$ & $S_g \leq s_h$ then (* $x_m > x^*$ *) $a = x_m$
- else report optimum solution $x_1 = x_m$ & $x_2 = \frac{b - c_1 x_1}{c_2}$; Exit

STEP 2.4: *(Pruning step - the case where iteration continues. Without loss of generality assume that $x^* > x_m$; *)

For each pair $(i, j) \in M_1 \cup M_2$ do

- If $a_i = a_j$ then ignore one of the two constraints;
- If $a_i \neq a_j$ and $x_{ij} < x_m$ then ignore one of the two constraints;

STEP 3: *(The case when $|I_1 \cup I_2| \leq 4$)*

The problem can be solved directly.
Observe that rest of this algorithm follows standard prune-and-search scheme stated in Algorithm 4.1.

So, according to Lemma 4.1, we can enumerate all the valid constraints after each iteration of the while-loop using the pairing scheme, and in \( t \)-th \((1 \leq t \leq K = O(\log n))\) iteration enumeration of all the valid constraints will take \( O(nt) \) time.

Now, in order to implement MEGIDDO’S-2D-LP \((I,c_1,c_2)\) in the read-only environment using limited work-space, we have to show that INTERMEDIATE-COMPUTATION (i.e, STEP 2.2 and 2.3) is implementable in the pairing scheme with limited work-space. Observe that STEP 2.3 can be implemented easily once we can recognize the valid pairs/constraints, and STEP 2.2 is the median computation step. So, combining Lemma 4.1 and 4.4, we have the following:

**Theorem 7.1** MEGIDDO’S-2D-LP \((I,c_1,c_2)\) can be correctly computed in the read-only environment in (i) \( O(n^{1+\frac{1}{k+1}} \log^{k+2} n) \) time with \( O(k \log n) \) extra-space, where \( k \) is a fixed natural number, or (ii) \( O(n \log^4 n) \) time with \( O(\log^2 n) \) extra-space.

### 7.3 Linear programming in \( \mathbb{R}^3 \)

In the same paper [Megiddo 1983b], Megiddo proposed a linear time algorithm for linear programming problem with three variables. The problem is stated as follows:

\[
\min_{x_1,x_2,x_3} d_1 x_1 + d_2 x_2 + d_3 x_3 \\
\text{subject to:} \\
d'_i x_1 + b'_i x_2 + c'_i x_3 \geq \beta_i, \ i \in I = \{1, 2, \ldots n\}.
\]

As earlier, Megiddo transformed the problem into the following equivalent form:

\[
\min_{x,y,z} z \\
\text{subject to:} \\
z \geq a_i x + b_i y + c_i, \ i \in I_1, \\
z \leq a_i x + b_i y + c_i, \ i \in I_2, \\
0 \geq a_i x + b_i y + c_i, \ i \in I_3, \\
|I_1| + |I_2| + |I_3| \leq n.
\]
Algorithm 7.2: MEGIDDO’S-3D-LP($I, c_1, c_2$)

**Input:** A set of $n$ constraints $a_i^j x_1 + b_i^j x_2 + c_i^j x_3 \geq \beta_i$, for $i \in I = \{1, 2, \ldots, n\}$.

**Output:** The value of $x_1, x_2, x_3$ which minimizes $c_1^j x_1 + c_2^j x_2 + c_3^j x_3$

1. (* Uses standard primal-search scheme *)
2. **STEP 1:** Convert the given form into following:
   \[
   \min_{x_1, x_2, x_3}, \text{subject to } (i) \ y \geq a_i^1 x_1 + b_i^1 x_2 + c_i^1, \ i \in I_1, \ (ii) \ y \leq a_i^2 x_1 + b_i^2 x_2 + c_i^2, \ i \in I_2, \ (iii) \ y \leq a_i^3 x_1 + b_i^3 x_2 + c_i^3, \ i \in I_3, \ |I_1| + |I_2| + |I_3| \leq n.
   \]
3. **STEP 2:**
   1. While $|I_1| \cup |I_2| \cup |I_3| \geq 16$ do
      1. **STEP 2.1:** Arbitrarily pair-up the constraints $a_i^1 x_1 + b_i^1 x_2 + c_i^1$, $a_i^2 x_1 + b_i^2 x_2 + c_i^2$ where $i; j$ are from same set $I_k, k \in \{1, 2, 3\}$. Let $L_1, L_2, L_3$ be the set of aforementioned pairs and $L = L_1 \cup L_2 \cup L_3$.
      2. **STEP 2.2:** Let $L_C = \{(i, j) \in L | (a_i, b_i) \neq (a_j, b_j)\}$ and $L_P = \{(i, j) \in L | (a_i, b_i) = (a_j, b_j)\}$.
      3. Compute the median $\mu$ of the slopes $a_{(i,j)}$ of all the straight lines $L_{ij}$:
         \[
         \mu = \frac{a_{(i,j)} + a_{(i,j)^*}}{2}, \text{where } d_i = \text{distance of } L_i \text{ from the line } y = \mu x.
         \]
      4. **STEP 2.3:** Arbitrarily pair up $(L_{ij}, L_{ij}^*)$ where $a_{(i,j)} \leq \mu \leq a_{(i,j)^*}$ and $(i, j), (i', j') \in L_C$.
      5. Let $M_P = \{(L_{ij}, L_{ij}^*) \in M | a_{(i,j)} = \mu = a_{(i,j)^*}\}$ (* parallel line-pairs *) and $M_I = \{(L_{ij}, L_{ij}^*) \in M | a_{(i,j)} \neq a_{(i,j)^*}\}$ (* intersecting line-pairs *).
      6. For each pair $(L_{ij}, L_{ij}^*) \in M_P$ do:
         1. Compute $y_{ij} = \frac{a_{(i,j)} + a_{(i,j)^*}}{2}$, where $d_i$ is distance of $L_i$ from the line $y = \mu x$.
      7. For each pair $(L_{ij}, L_{ij}^*) \in M_I$ do:
         1. Let $a_{ij}$ be the intersection point of $L_{ij}$ and $L_{ij}^*$, and $b_{ij}$ be the projection of $a_{ij}$ on $y = \mu x$.
         2. Compute $y_{ij} = \text{signed distance of the pair of points } (a_{ij}, b_{ij}) \text{ and } x_{ij} = \text{signed distance of } b_{ij} \text{ from the origin}$.
     8. Next, compute the median $y_{0_1}$ of the $y_{ij}$ values corresponding to all the pairs in $M$. Let $(x', y')$ be the corresponding intersection point.
     9. **Step 2.4:** Consider the line $L_H : y - y' = \mu(x - x')$, which is parallel to $y = \mu x$ and passes through the point $(x', y')$.
     10. Test on which half-plane defined by the line $L_H$ contains the optimum by evoking **Testing-Line($L_H$)**.
     11. **Step 2.5:** Let $M_F = \{(L_{ij}, L_{ij}^*) \in M | a_{ij} \neq \mu \text{ lie in different sides of } L_H\}$.
     12. Compute the median $\nu$ of $x_{ij}$-values for the line-pairs in $M_F$. Let $(x'', y'')$ be the corresponding intersection point. Define a line $L_V : y - y'' = \frac{1}{\mu}(x - x'') \text{ perpendicular to } L_H$.
     13. Execute the procedure **Testing-Line($L_V$)** and decide in which side of $L_V$ the optimum lies;
     14. We consider $L_H$ and $L_V$ as horizontal and vertical lines respectively; let $Quad$ be the quadrant defined by $L_H$ and $L_V$ in which the optimum lies;
     15. **Step 2.6:** (* Pruning step *)
         1. For all $(i, j) \in L_1 \cup L_2 \cup L_3$ do
             1. If $(a_i, b_i) = (a_j, b_j)$ then
                 1. Discard one of the constraints depending on their $c$ value;
                 2. If the corresponding line $L_{ij}$ does not intersect the quadrant $Quad$ then
     26. **STEP 3:** (*The case when $|I_1| \cup |I_2| \cup |I_3| \leq 16$ *)
     27. The problem can be solved directly by brute-force manner.

The Algorithm MEGIDDO’S-3D-LP pairs-up constraints $(C_k^i, C_k^j)$, where $C_k^i, C_k^j$ are from the same set $I_k, k \in \{1, 2, 3\}$. So, there are at most $\frac{n^2}{27}$ pairs. Let $C_k^i$ (resp. $C_k^j$) corresponds to $a_i x + b_i y + c_i$ (resp. $a_j x + b_j y + c_j$). If $(a_i, b_i) = (a_j, b_j)$, then we can easily ignore one of the constraints. Otherwise (i.e., $(a_i, b_i) \neq (a_j, b_j)$), each pair signifies a line $L_{ij}$: $a_i x + b_i y + c_i = a_j x + b_j y + c_j$ which divides the plane into two halves. Let $L$ be the set of lines obtained in
this way. We compute the median \( \mu \) of the slopes of the members in \( \mathcal{L} \). Next, we pair-up the members in \( \mathcal{L} \) such that one of them have slope less than \( \mu \) and the other one have slope greater than \( \mu \). Let \( \Pi \) be the set of these paired lines. Each of these pairs will intersect. We compute the intersection point \( a = (x', y') \) having median \( y_m \) among the \( y \)-coordinates of these intersection points. Next, we execute the procedure TESTING-LINE with respect to the line \( L_H : y - y' = \mu (x - x') \) (having slope \( \mu \) and passing through the point \( a \)). This determines in which side of \( L_H \) the optimum solution lies. The details of TESTING-LINE is stated in the next subsection. Next, we identify the pairs in \( \Pi \) which intersect on the other side of the optimum solution. Among these pairs, we compute the intersection point \( b = (x'', y'') \) having median of the \( x \)-coordinates of their intersections, and execute TESTING-LINE with the line \( L_V \) having slope \( \frac{1}{\mu} \) and passing through \( b \). \( L_H \) and \( L_V \) determines a quadrant \( Q \) containing the optimum solution. Now consider the paired lines in \( \Pi \) that intersect in the quadrant \( Q' \), diagonally opposite to \( Q \). Let \((L_{ij}, L_{k\ell})\) be a pair of lines in \( \Pi \) that intersect in \( Q' \). For at least one of the lines \( L_{ij} \) and \( L_{k\ell} \), it is possible to correctly identify the side containing the optimum solution without executing TESTING-LINE (see [Megiddo 1983b]). Thus, for each of such lines we can prune one constraint. As a result, after each iteration it can prune at-least \( \frac{n}{16} \) constraints for next iteration or report the optimum. The detailed description of the algorithm MEGIDDO’S-3D-LP is given in Algorithm 7.2.

### 7.3.1 TESTING-LINE

The procedure TESTING-LINE, described in [Megiddo 1983b], works as follows. It takes a straight line \( L \) in the \( x-y \) plane and decides which of the two half-planes, defined by the line \( L \), contains the optimum solution. For deciding this, TESTING-LINE needs to solve at most three linear programming in \( \mathbb{R}^2 \). Remember that following is our targeted linear programming in \( \mathbb{R}^3 \):

\[
\begin{align*}
\min & \quad z \\
\text{subject to:} & \quad z \geq a_i x + b_i y + c_i, \ i \in I_1, \\
& \quad z \leq a_i x + b_i y + c_i, \ i \in I_2, \\
& \quad 0 \geq a_i x + b_i y + c_i, \ i \in I_3, \\
& \quad |I_1| + |I_2| + |I_3| \leq n.
\end{align*}
\]
Define \( f(x, y) = \max \{ \max \{ a_i x + b_i y + c_i, i \in I_1 \} - \min \{ a_i x + b_i y + c_i, i \in I_2 \}, \max \{ a_i x + b_i y + c_i, i \in I_3 \} \} \).

Transform the coordinate system in such a way that the line \( L \) coincides with the \( x \)-axis. So, the objective of Testing-Line is to decide whether \( y > 0 \) or \( y < 0 \). First, solve the following linear programming:

\[
\begin{align*}
\min_z \quad & z \\
\text{subject to:} & \quad z \geq a_i x + c_i, \ i \in I_1, \\
& \quad z \leq a_i x + c_i, \ i \in I_2, \\
& \quad 0 \geq a_i x + c_i, \ i \in I_3, \\
& \quad |I_1| + |I_2| + |I_3| \leq n.
\end{align*}
\]

Let \( x = x^* \) be reported by solving the above linear programming in \( \mathbb{R}^2 \). Translate the \( x \)-coordinate such that \( x^* = 0 \). Define

\[ I_1^* = \{ i \in I_1 | c_i = \max \{ c_j : j \in I_1 \} \} . \]

Similarly, \[ I_3^* = \{ i \in I_3 | c_i = \max \{ c_j : j \in I_3 \} \} . \]

If \( \max \{ c_i : i \in I_1 \} - \min \{ c_i : i \in I_2 \} \geq 0 \), then define

\[ I_2^* = \{ i \in I_2 | c_i = \min \{ c_j : j \in I_2 \} \} . \]

Otherwise, \[ I_2^* = \phi. \]

Now, two cases may arise depending on the value of \( f(0, 0) \leq 0 \) or \( f(0, 0) > 0 \). Assume that \( f(0, 0) \leq 0 \). For this case, solve the following:

\[
\begin{align*}
\min_{\lambda, \eta} \quad & \eta \\
\text{subject to:} & \quad \eta \geq a_i \lambda + b_i, \ i \in I_1^*, \\
& \quad \eta \leq a_i \lambda + b_i, \ i \in I_2^*, \\
& \quad 0 \geq a_i \lambda + b_i, \ i \in I_3^*.
\end{align*}
\]

If \( \eta < 0 \) is reported, then Testing-Line decides \( y > 0 \). Otherwise, it performs the following linear programming:
\[
\begin{align*}
\min_{\lambda, \eta} \eta \\
\text{subject to:} \\
\eta &\leq a_i \lambda + b_i, \ i \in I_1^*, \\
\eta &\geq a_i \lambda + b_i, \ i \in I_2^*, \\
0 &\leq a_i \lambda + b_i, \ i \in I_3^*.
\end{align*}
\]

If \( \eta > 0 \), then Testing-Line decides \( y < 0 \), otherwise the point \((x^*, 0)\) is an optimal solution.

The case \( f(0, 0) > 0 \) is handled in a similar way. The details of this can be found in [Megiddo 1983b].

### 7.3.2 Read-only linear programming in \( \mathbb{R}^3 \)

Observe that Megiddo’s-3D-LP, given in Algorithm 7.2, follows the standard prune-and-search scheme stated in Algorithm 4.1. We will show that Intermediate-Computation (i.e. STEP 2.2 to STEP 2.5) can be implemented in the read-only environment using the pairing scheme stated in Section 4.4. Note that STEP 2.2 and 2.3 are the steps involving computation of median and STEP 2.4 and 2.5 are regarding Testing-Line. In Chapter 4, we have already shown that median can be computed using the pairing scheme in the read-only environment (see Lemma 4.4). In Section 7.2 (see Theorem 7.1), we have shown that linear programming in \( \mathbb{R}^2 \) can be solved in read-only environment. As Testing-Line performs at most three linear programming in \( \mathbb{R}^2 \), we can easily implement Testing-Line in read-only environment using the pairing scheme. Thus, we have the following theorem.

**Theorem 7.2** Linear programming with three variables can be implemented in a read-only model in (i) \( O(n^{1+\frac{1}{k+1}} \log^{k+3} n) \) time using \( O(k \log n) \) extra-space, where \( k \) is a fixed natural number, or (ii) \( O(n \log^5 n) \) time and \( O(\log^2 n) \) extra-space.

### 7.4 Conclusion

Linear programming is one of the most useful techniques for solving several optimization problems. In this chapter, we show the deterministic way to
solve two and three dimensional linear programming when the input is given in a read-only memory.

It needs to be mentioned that Brönnimann et al. [Brönnimann 2004b, Theorem 5] showed that Megiddo’s-2D-LP can be implemented in an in-place manner in linear time. Using the technique given in Chapter 5.3, it can be shown that both Megiddo’s-2D-LP and Megiddo’s-3D-LP can be solved in an in-place manner in linear time using only $O(1)$ extra-space.
Chapter 8

Maximum Clique of Geometric Intersection Graphs

Contents

8.1 Introduction ........................................... 111
8.2 Maximum clique of an interval graph ................. 113
8.3 Maximum clique for rectangles of arbitrary size .... 115
8.4 Maximum clique for fixed height rectangles ......... 116
  8.4.1 Processing of a strip S .......................... 117
  8.4.2 Processing of Case 3 .............................. 119
8.5 Geometric clique for disks of arbitrary radii ....... 120
8.6 Graphical clique for unit disk graph ................. 122
  8.6.1 Maximum matching in the bipartite graph $G_{B_{ij}}$ . 124
  8.6.2 Complexity analysis ............................. 134
8.7 Maximum independent set ............................ 135
8.8 Conclusion ............................................. 136

8.1 Introduction

The geometric intersection graph $G = (V, E)$ of a set of geometric objects $S$ is a graph whose nodes $V$ correspond to the set of objects in $S$. Between a pair of nodes $v_i$ and $v_j$, there is an edge $(v_i, v_j)$ if the corresponding objects in $S$ intersect. The intersection of a pair of objects is defined depending on the problem specification.
A clique of a graph $G = (V, E)$ is defined as a set $C \subseteq V$ such that for all pair of members of $C$, there is an edge. Similarly, an independent set of a graph $G = (V, E)$ is a set $I \subseteq V$ such that no two members of $I$ have an edge. So, trivially any single object is a clique and also an independent set of size one. The optimization problem is to find the clique (resp. independent set) of maximum size.

In this chapter, first, we propose an in-place $O(n \log n)$ time algorithm for computing the maximum clique of an intersection graph of a set of $n$ intervals on the real line. We use this algorithm to design an in-place algorithm for finding the maximum clique of the intersection graph of a set of $n$ axis-parallel rectangles of arbitrary size in $O(n^2 \log n)$ time. For fixed height rectangles, the time complexity can be improved to $O(n \log n + nK)$, where $K$ is the size of the largest clique.

Next, we consider the intersection graph of a set of disks. Let $C$ be a subset of disks such that each pair of members in $C$ intersect. In the aforesaid graph, the nodes corresponding to $C$ define a clique. However, the members of $C$ may not have a common intersection (see Figure 8.1(a)). Thus, a clique in a disk graph is usually referred to as graphical clique. In particular, if the members in a clique have common intersection region, then that clique is referred to as a geometric clique. For demonstration, see Figure 8.1.

![Figure 8.1: (a) Graphical clique and (b) geometric clique of disk graph](image)

Our proposed in-place algorithm for computing the largest geometric clique of the intersection graph of a set of disks of arbitrary radii needs $O(n^2 \log n)$ time. For graphical clique, our in-place algorithm works for unit disks only, and it runs in $O(n^2 + mK^3)$ time, where $n$ and $m$ are the number of vertices and edges in the unit disk graph, and $K$ is the size of the maximum clique in that graph.

Notice that the $(time \times extra-space)$ is less than or equal to the best-known
results for all the problems we have considered excepting the last one.

To compute the largest graphical clique of unit disk graph, we need the maximum matching of a bipartite graph (see [Clark 1990]). We first propose an $O(n^3)$ time in-place algorithm for computing a maximum matching in a bipartite graph $G = (V_1, V_2, E)$ where the two sets of nodes $V_1$ and $V_2$ are stored in two arrays, and the existence of an edge between a pair of nodes can be checked on demand by an oracle in $O(1)$ time.

The in-place algorithm for bipartite matching problem is of independent interest, since to the best of our knowledge, no space-efficient algorithm for computing the maximum matching in a bipartite graph was known in the literature. The most useful $O(n^{2.5})$ time algorithm using max-flow [Papadimitriou 2006] uses $O(E)$ extra bits to store the flow values. The $O(n^{2.37})$ time algorithm based on Gaussian elimination [Mucha 2004] needs the inverse of the incidence matrix, which also needs $O(n^2)$ space to compute and store. If the incidence matrix is given in a read-only (bit-)matrix of dimension $O(|V_1| \times |V_2|)$, then using an array of size $O(|V_1| + |V_2|)$ as extra-space, our proposed algorithm can compute the maximum matching. Here each array element stores the identity of a node which may be $O(\log(|V_1| + |V_2|))$ bit long.

8.2 Maximum clique of an interval graph

Here a set of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ is given in an array. Each element $I_i$ of the array is a tuple $(\ell_i, r_i)$, that represents the coordinates of the left and right end points of $I_i$ on the real line. The goal is to design an in-place algorithm for finding the maximum clique of the interval graph corresponding to the intervals in $\mathcal{I}$. In other words, we need to compute a point on the real line at which maximum number of intervals in $\mathcal{I}$ overlap. From now onwards, $\mathcal{I}$ will denote the array containing the given set of intervals.

Sweeping a set of intervals using a heap data structure has been proposed earlier by Vahrenhold [Vahrenhold 2007a]. We use the same approach. We maintain a list $L$ and a max-heap $H$ in the same array $\mathcal{I}$ during the execution. The list $L$ contains a subset of intervals in $\mathcal{I}$ sorted in increasing order of their $\ell$-values, and the heap $H$ contains a subset of intervals in $\mathcal{I}$ in order of their
r-values. More specifically, the array \( I \) has three parts: (i) the heap \( H \), (ii) the list \( L \), and (iii) \( D \) - the elements that are deleted from both \( L \) and \( H \). Thus, \( H \cup D \cup L = I \). We maintain two index variables \( \lambda \) and \( \mu \); \( H = I[1, 2, \ldots, \lambda] \), \( L = I[\mu + 1, \mu + 2, \ldots, n] \), and the elements \( D = I[\lambda + 1, \lambda + 2, \ldots, \mu] \) are deleted from both \( H \) and \( L \). The root of the heap is the first element of the array \( I \). The deletions from \( L \) and \( H \) are as follows:

- After the deletion of the root element from \( H \), \( \lambda \) is reduced by 1.
- We delete an element from \( L \) by (i) incrementing \( \lambda \) and \( \mu \), (ii) swapping \( I[\mu] \) and \( I[\lambda] \), and then (iii) positioning \( I[\lambda] \) in its appropriate position in the heap \( H \) with respect to its \( r \)-value.

Initially, \( I \) contains the ordered list \( L \); the heap \( H \) and the array \( D \) are empty. We use two scalar variables \( \chi \) and \( \pi \) to store the size and the point on the real line representing the maximum clique. The elements in \( L \) are considered for processing in order (i.e., in increasing order of their \( \ell \)-values). Each time the first element of \( L \) and the element stored at the root of \( H \) are considered; the one having minimum key value (\( r \)-value for \( H \) and \( \ell \)-value for \( L \)) is processed. When an element in \( L \) (say \( I_i = (\ell_i, r_i) \)) is processed, it is deleted from \( L \), and inserted in \( H \) with respect to \( r_i \). When an element (the root) of \( H \) corresponding to an interval (say \( I_j = (\ell_j, r_j) \)) is processed, a maximal clique is observed at \( r_j \). Now, if \( |H| = n - \lambda > \chi \), then \( \chi \) and \( \pi \) are set with \( |H| \) and \( r_j \). Observe that \( I_j \) will never contribute to any other clique; thus \( I_j \) needs to be deleted from the data structure. Also note that \( I_j \) is already deleted from \( L \); we delete \( I_j \) from \( H \) and move it to \( D \). This phase of the algorithm terminates after considering all the elements in \( L \). Finally, another scan of all the intervals in \( I \) is needed to report the members in \( I \) that contain \( \pi \) to form the maximum clique.

Observe that we have used only constant number of variables and the time complexity of the stated algorithm is \( O(n \log n) \). Hence, we have the following result:

**Theorem 8.1** The clique of maximum size, for a given a set of \( n \) intervals, can be found in an in-place manner in \( O(n \log n) \) time using \( O(1) \) extra-space.
8.3 Maximum clique for rectangles of arbitrary size

Here, a set \( R \) of \( n \) axis-parallel rectangles of arbitrary size is given in \( \mathbb{R}^2 \). Since axis-parallel rectangles satisfy Helly property\(^1\), the members of a clique in the rectangle intersection graph have a nonempty intersection. Thus, the objective is to find a point in \( \mathbb{R}^2 \) on which maximum number of rectangles overlap. Each rectangle \( R_i \) is specified by a tuple \( (\alpha_i, \beta_i) \), where \( \alpha_i = (x_{\alpha_i}, y_{\alpha_i}) \) and \( \beta_i = (x_{\beta_i}, y_{\beta_i}) \) are the coordinates of the top-left and bottom-right corners of \( R_i \). We use \( R \) as an array storing the set of rectangles \( \{R_1, R_2, \ldots, R_n\} \).

The plane sweep algorithm for finding the maximum clique of a set of axis-parallel rectangles works as follows [Imai 1983, Nandy 1995]. Sweep a horizontal line \( \ell \) from top to bottom. When the top boundary of a rectangle \( R_i \) is faced by the line \( \ell \), it becomes active. When the bottom boundary of \( R_i \) is faced by the line \( \ell \), we consider only those active rectangles which overlap on the bottom boundary of \( R_i \). Each of these rectangles defines an interval corresponding to its portion of overlap with the bottom boundary of \( R_i \). Compute the maximum clique among those intervals. Now, \( R_i \) becomes inactive. The sweep continues until all the bottom boundaries are processed.

Our in-place algorithm follows the same technique. We execute \( n \) iterations to consider the rectangles in \( R \) in a top-to-bottom order of their bottom boundaries as stated below. We use a variable \( \gamma \) to store the \( y \)-coordinate of the last processed bottom boundary. In the \( i \)-th iteration, we compare the bottom boundaries of all the rectangles \( R \) with \( \gamma \) to identify those rectangles which are not yet processed, and among them, we identify the one, say \( R_i \), having maximum \( y \)-coordinate. We swap it with the rectangle stored at \( R[1] \).

Next, we execute the following steps:

- Identify all the rectangles in \( R \) that overlap on the bottom boundary of \( R_i \), and move them at the beginning of the array \( R \). If there are \( m_i \) such rectangles, they will occupy the positions \( R[2, \ldots, m_i + 1] \). This can be done in \( O(n) \) time. We use two pointers \( \pi_1 \) and \( \pi_2 \), where \( \pi_1 \)

\(^1\)A set of convex objects in \( \mathbb{R}^2 \) is said to satisfy Helly property [Imai 1983] if the members of a subset of this set are mutually intersecting then they have a common intersection region, and vice versa.
moves from $R_2$ towards right until it finds a rectangle $R_j$ that does not overlap on $R_i$, and $\pi_2$ moves from $R_n$ towards left until it finds a rectangle $R_k$ that overlaps on $R_i$. $R_j$ and $R_k$ are swapped. The move proceeds until $\pi_1 \geq \pi_2$ is attained.

- Each of these $m_i$ rectangles define an interval on the bottom boundary of $R_i$. We compute the clique of maximum size in this interval graph using the algorithm proposed in Section 8.2 in $O(m_i \log m_i)$ time.

The fact that $\sum_{i=1}^n m_i = O(n^2)$ lead to the following theorem stating the time complexity result of our proposed algorithm.

**Theorem 8.2** The maximum clique of the intersection graph of a set of $n$ axis-parallel rectangles can be computed in an in-place manner in $O(n^2 \log n)$ time using $O(1)$ extra-space.

### 8.4 Maximum clique for fixed height rectangles

Now, we consider a constrained version of the maximum clique problem of rectangle intersection graph, where the height of all the rectangles in $\mathcal{R}$ are same, say $\delta$. Here, each rectangle $R_i$ is represented by a triple $(\alpha(R_i), \beta(R_i), \omega(R_i))$; $\alpha(R_i)$, $\beta(R_i)$ and $\omega(R_i)$ represent the $x$-coordinates of left and right vertical boundaries, and $y$-coordinate of the top boundary of $R_i$ respectively.

**Observation 8.1** If we split the plane into $m$ horizontal strips by $m + 1$ horizontal lines $\ell_1, \ell_2, \ldots, \ell_m$ such that

- each of the two consecutive lines are at least $\delta$ distance apart,
- none of the lines is aligned with the boundary line of any member of $\mathcal{R}$,
- each member of $\mathcal{R}$ is stabbed by exactly one line in $\{\ell_1, \ell_2, \ldots, \ell_m\}$, and
- each line stabs at least one member of $\mathcal{R}$,

then each member in $\mathcal{R}$ spans at most two consecutive strips.
First, we split the region into horizontal strips satisfying Observation 8.1. Note that this splitting is similar to the splitting described in [Agarwal 1998b]. We can achieve this by sorting the members in $\mathcal{R}$ with respect to their $\omega$-values. It will take $O(n \log n)$ time and $O(1)$ extra-space.

Next, we compute the maximum clique in each strip, and report the one having maximum size. We maintain two global counters $\chi$ and $\pi$ to contain the size of the largest clique $C$ and a point in the region representing the largest clique, during the entire execution. We now describe the method of processing a strip.

### 8.4.1 Processing of a strip $S$

Let $\mathcal{R}_t$ and $\mathcal{R}_b$ denote the two sub-arrays of $\mathcal{R}$ that contain all the rectangles stabbed by the top and bottom boundaries of the strip $S$. We can designate the sub-arrays $\mathcal{R}_t$ and $\mathcal{R}_b$ in $\mathcal{R}$ using two integer tuples $(m_1, m_2)$ and $(n_1, n_2)$, respectively. We compute the largest clique on the right boundary of each rectangle in $\mathcal{R}_t \cup \mathcal{R}_b$, and finally note down the largest one among them as $C_S$.

Consider the processing of right boundary of a rectangle $\rho \in \mathcal{R}_t \cup \mathcal{R}_b$. Let $R_1^*, R_2^*, \ldots, R_k^* \in \mathcal{R}_t$ and $R_1', R_2', \ldots, R_\ell' \in \mathcal{R}_b$ be the set of all rectangles whose left boundaries are to the left of the right boundary of $\rho$ and right boundaries are to the right of right boundary of $\rho$. Note that $R_1^*, R_2^*, \ldots, R_k^* \in \mathcal{R}_t$ form a clique at a point on the top boundary of $S$, and $R_1', R_2', \ldots, R_\ell' \in \mathcal{R}_b$ form a clique at a point on the bottom boundary of $S$. Thus, we have the following result.

**Lemma 8.1** $|C_S| \geq \max\{k, \ell\}$. 

We process the members in $\mathcal{R}_t \cup \mathcal{R}_b$ in increasing order of their $\alpha$-values (left boundaries) in a merge like fashion. This needs sorting of $\mathcal{R}_t$ and $\mathcal{R}_b$ in increasing order of their $\alpha$-values. During the execution, we arrange the rectangles in $\mathcal{R}_t$ (resp. $\mathcal{R}_b$) into three portions as stated below (see Figure 8.2(a)).
A: All the rectangles, whose both left and right boundaries are processed, This portion stays at the left side of \( R_t \) (resp. \( R_b \)),

B: The rectangles whose left boundary is processed but the right boundary is not yet processed. This portion stays at the middle of \( R_t \) (resp. \( R_b \)) in decreasing order of their \( \omega \)-values.

C: The rectangles whose both left and right boundaries are not processed. This portion stays at the end of \( R_t \) (resp. \( R_b \)) in increasing order of their \( \alpha \)-values.

We maintain four index variables \( i_t, i_b, j_t \) and \( j_b \), where \( i_t \) and \( j_t \) (\( i_t \leq j_t \)) indicate the portion \( B \) of the array \( R_t \), and \( i_b \) and \( j_b \) (\( i_b \leq j_b \)) indicate the portion \( B \) of the array \( R_b \). We also use two scalar variables \( \rho_t \) and \( \rho_b \) to maintain the rectangle having leftmost right boundary (\( \beta \)-value) among the members in the portion \( B \) in \( R_t \) and \( R_b \) respectively. Initially, the elements of \( R_t \) and \( R_b \) are sorted in increasing order of their \( \alpha \)-values, \( i_t, i_b, j_t, j_b \) are all set to 1, \( \rho_t = R_t[1] \) and \( \rho_b = R_b[1] \). Next, we start processing the elements of \( R_t \) and \( R_b \) in a merge like fashion. At each step, we compare the \( \alpha(R_t[j_t+1]), \alpha(R_b[j_b+1]), \beta(\rho_t) \) and \( \beta(\rho_b)^2 \). Here the following four situations may arise,

**Case 1**: \( \alpha(R_t[j_t+1]) \) is minimum: \( R_t[j_t+1] \) is moved (from the C part) to the appropriate position in the B part of \( R_t \) with respect to its \( \omega \)-value using a sequence of swap operations. If the \( \beta(R_t[j_t+1]) < \beta(\rho_t) \), then \( \rho_t \) is updated by \( R_t[j_t+1] \). Finally, \( j_t \) is incremented by 1.

---

\( ^2R_t[j_t+1] \) and \( R_b[j_b+1] \) are the first element of the C part of the respective array.
8.4. Maximum clique for fixed height rectangles

Case 2: \( \alpha(R_b[j_b + 1]) \) is minimum: This situation is handled as in Case 1.

Case 3: \( \beta(\rho_t) \) is minimum: In the next subsection, we explain the processing of this situation.

Case 4: \( \beta(\rho_b) \) is minimum: This situation is handled as in Case 3.

8.4.2 Processing of Case 3

As mentioned earlier, all the rectangles \( R_t[i_t], R_t[i_t + 1], \ldots, R_t[j_t] \) overlap at the point of intersection \( a \) of \( \rho_t \) and the top-boundary of \( S \). We initialize a variable \( count \) with \((j_t - i_t + 1)\). Next, we process the members of the \( B \) portion of \( R_t \) (i.e., from the index position \( i_t \) to \( j_t \)), and the members of the \( B \) portion of \( R_b \) (i.e., from the index position \( i_b \) to \( j_b \)) together in decreasing order of their \( \omega \)-values in a merge-like fashion. We initialize two index variables \( \theta_t \) and \( \theta_b \) with \( i_t \) and \( i_b \) respectively. At each step, if \( \omega(R_t[\theta_t]) - \delta > \omega(R_b[\theta_b]) \) then \( count \) is decreased by one, and \( \theta_t \) is incremented; otherwise (i) \( count \) is increased by one, (ii) if \( count > \chi \), then \( \chi \) is set with \( count \), \( \pi \) is set with the point of intersection of \( \rho_t \) and the rectangle \( R_b[\theta_b] \), and then (iii) \( \theta_b \) is incremented. The process terminates when \( \theta_b \) reaches \( j_b \) or \( \omega(R_b[\theta_b]) < \omega(\rho_t) - \delta \) or \( \omega(R_t[\theta_t]) - \delta < \omega(\rho_t) - \delta \). At the end of processing \( \rho_t \), we need to perform the following operations: (i) move \( \rho_t \) to the \( i_t \)-th position of \( R_t \) (i.e., the \( A \) part of \( R_t \)) using a sequence of swap operations, (ii) set \( \rho_t \) by sequentially inspecting the members in \( R_t \) from index position \((i_t + 1)\) to \( j_t \), and (iii) increment \( i_t \). See Figure 8.2(b) for the demonstration.

Lemma 8.2 The time complexity of processing a strip is \( O(n_S \log n_S + n_S|C_S|) \), where \( n_S = |R_t| + |R_b| \) and \( C_S \) is the largest clique in the strip \( S \).

Proof: While processing a strip \( S \), initial sorting of the members in \( R_t \) (resp. \( R_b \)) with respect to their \( \alpha \)-values need \( O(|R_t| \log |R_t|) \) (resp. \( O(|R_b| \log |R_b|) \)) time. For each occurrence of Case 1 (resp. Case 2) (i.e., a left boundary of a member in \( R \in R_t \) (resp. \( R_b \)) we need to position \( R \) in the \( B \) part of \( R_t \) (resp. \( R_b \)) with respect to its \( \omega \)-value. This needs at most \( O(j_t - i_t) \) (resp. \( O(j_b - i_b) \)) swaps. By Lemma 8.1, the size of the \( B \) part of both \( R_t \) and \( R_b \) is at most \( O(|C_S|) \) at any instant of time. We now analyze the time complexity.
of processing an instance of Case 3, i.e., the right boundary of a rectangle \( \rho_t \in \mathcal{R}_t \).

While computing the largest clique along the right boundary of \( \rho_t \), we inspect the members of \( \mathcal{R}_t \) from index position \( i_t \) to \( j_t \), and the members of \( \mathcal{R}_b \) from index position \( i_b \) to \( j_b \) whose top boundaries are above the bottom boundary of \( \rho_t \). Both these numbers are less than \(|C_S|\) in number (by Lemma 8.1). Next, moving \( \rho_t \) to the end of the portion \( A \) of \( \mathcal{R}_t \) needs at most \( j_t - i_t \) swaps. The resetting of \( \rho_t \) with the existing members of \( B \) part of \( \mathcal{R}_t \) for further processing in strip \( S \) needs another \( j_t - i_t \) computations. Thus, the total time complexity for processing the right boundary of \( \rho_t \) is \( O(|C_S|) \). The same arguments hold for processing the right boundary of a member in \( \mathcal{R}_b \). Thus, processing the entire strip \( S \) needs \( O(n_S \log n_S + n_S|C_S|) \) time, where \( n_S = |\mathcal{R}_t| + |\mathcal{R}_b| \).

Note that each rectangle appears in exactly two strips, and if \( K \) be the size of the largest clique then \( K \geq |C_S| \) for all strips \( S \). Now, we have the following theorem.

**Theorem 8.3** The maximum size clique of an intersection graph of fixed height rectangles can be computed in an in-place manner in \( O(n \log n + nK) \) time using \( O(1) \) extra-space.

### 8.5 Geometric clique for disks of arbitrary radii

We now follow the same method as in Section 8.2 to compute the maximum size geometric clique of a set of circular disks of arbitrary radii. Here the input is an array containing a set of disks \( \mathcal{C} = \{C_1, C_2, \ldots, C_n\} \). Each element \( C_i \in \mathcal{C} \) is a triple \((\alpha_i, \beta_i, r_i)\), where \((\alpha_i, \beta_i)\) is the coordinate of the center of \( C_i \) and \( r_i \) is its radius.

**Observation 8.2** Any geometric clique of a disk graph corresponds to a closed convex region bounded by arc segments of some/all the disks participating in it.

Let us consider a disk \( C_i \); \( \Delta(C_i) \) be the boundary (circumference) of \( C_i \). Each disk in \( \mathcal{C} \setminus \{C_i\} \) that properly intersects \( C_i \), contributes a non-closed arc along
8.5. Geometric clique for disks of arbitrary radii

Figure 8.3: (a) Closed and (b) non-closed arc, and (c) the circular arcs along ∆(Ci) after considering all the disks intersecting ∆(Ci).

Δ(Ci). If a disk Cj ∈ C\{Ci} properly contains Ci, then it contributes a closed arc along Δ(Ci). In Figures 8.3(a) and 8.3(b), the closed and non-closed arcs around Ci (for another enclosing (resp. intersecting) circle) are shown. If a disk Cj ∈ C\{Ci} is properly contained in Ci, it does not contribute any arc along Δ(Ci). Thus, after considering all the disks in C that intersect or contain Ci, we have a circular arc graph Gi with the arcs around Δ(Ci) (see Figure 8.3(c)). Observation 8.2 says that the maximum clique of the disk graph with the set of disks C corresponds to the maximum clique of the circular-arc graph Gi for some i ∈ {1, 2, ..., n}. We consider each disk Ci ∈ C, and compute ηi, the size of the maximum clique of the circular-arc graph Gi. Finally we report η = maxn i=1 ηi.

Finding maximum clique of the circular-arc graph on ∆(Ci)

While processing Ci, it is swapped with the first element of the array C. Next, a scan among the elements of C is performed to accumulate all the disks that properly intersect Ci. Let us name this set of disks as Ci, ni = |Ci|. These are all placed in the locations C[2...ni]. During this traversal, we count the number µi of disks that properly contain Ci. Next, we compute νi = the size of the maximum clique of the circular-arc graph of the non-closed arcs contributed by the disks in Ci on Δ(Ci) as stated below. Finally, we compute ηi = µi + νi.

We fix a point θ on Δ(Ci). For each disk C ∈ Ci, the left and right end-points (κleft and κright) of the arc generated by C on Δ(Ci) is computed as follows:
If $\theta \in C$, then $\kappa_{\text{left}}$ (resp. $\kappa_{\text{right}}$) is the point of intersection of $C$ and $C_i$ in anti-clockwise (resp. clockwise) direction from $\theta$.

If $\theta \notin C$, then $\kappa_{\text{left}}$ (resp. $\kappa_{\text{right}}$) is the closest (resp. farthest) point of intersection of $C$ and $C_i$ from $\theta$ in the clockwise direction.

We sort the members of $C_i$ in clockwise order of their left end-points ($\kappa_{\text{left}}$). Note that we do not store the arcs along $\Delta(C_i)$. While comparing a pair of arcs, we compute the points of intersection of the corresponding disks with $C_i$. Next, we process the end-points of the arcs in an ordered manner as in Section 8.2, implementing both a heap $H$ and a list $L$ in the portion $C_i$ of the array $C$ along with a list of elements deleted from both $H$ and $L$. However unlike Section 8.2, after processing all the left end-points, if $H$ contains some non-deleted elements, the algorithm does not stop. It again sorts all the deleted elements in $H$ in clockwise order of their left end-point, and continues the processing considering this list as $L$. The processing continues until all the elements of $H$ are processed. Thus, $\eta_i = \text{size of the maximum clique around } C_i$, is computed.

**Theorem 8.4** The geometric clique of maximum size among a set of $n$ disks of arbitrary radii can be computed in an in-place manner in $O(n^2 \log n)$ time using $O(1)$ extra-space.

### 8.6 Graphical clique for unit disk graph

Let $C = \{C_1, C_2, \ldots, C_n\}$ be a set of unit disks stored in an array $C$ of size $n$. Each element $C_i \in C$ stores the coordinate of its center $c_i = (\alpha_i, \beta_i)$. We show that the algorithm proposed in [Clark 1990] for computing the largest clique in the intersection graph $G = (C, E)$ of a set of unit disks can be made in-place. Here the vertices in $G$ correspond to the members in $C$; $(C_i, C_j) \in E$ if and only if the disks $C_i$ and $C_j$ intersect, i.e., the distance $d(c_i, c_j) \leq 1$.

Let $\chi \subseteq C$ be a set of disks forming the largest clique, and $c_i, c_j$ be the farthest pair of centers among the disks in $\chi$. All the centers of the members in $\chi$ lie in the region $R_{ij}$, formed by the intersection of circles of radius $d(c_i, c_j)$ centered
at \( c_i \) and \( c_j \), as shown in Figure 8.4. We use \( R_{ij}^1 \) and \( R_{ij}^2 \) to denote the parts of \( R_{ij} \) lying in two different sides of the line segment \([c_i, c_j]\), and \( C_{ij}^1 \) and \( C_{ij}^2 \) to denote the circles of \( C \) whose centers lie in \( R_{ij}^1 \) and \( R_{ij}^2 \), respectively. Note that the Euclidean distance between each pair of centers in \( C_{ij}^k \cup \{C_i, C_j\} \) is less than or equal to 1 for both \( k = 1, 2 \). Thus, if we form a bipartite graph \( G_{B_{ij}} = (C_{ij}^1, C_{ij}^2, E_{ij}) \), where an edge between a pair of vertices implies that their distance is greater than 1, then the set \( \chi \) corresponds to the maximum independent set in the graph \( G_{B_{ij}} \). Note that \( C_{ij}^k \) itself forms an independent set in \( G_{B_{ij}} \) for each \( k = 1, 2 \).

The algorithm for computing the largest graphical clique initializes \( \chi \) with 0, and then considers each pair of disks \( C_i, C_j \in C \). If \( C_i \) and \( C_j \) intersect, then it computes \( C_{ij}^1 \) and \( C_{ij}^2 \). If \( \chi \) denotes the size of the largest clique obtained so far, and \( |C_{ij}^1 \cup C_{ij}^2 \cup \{C_i, C_j\}| \leq \chi \), then the disks centered at \( R_{ij} \) will not produce a clique of size greater than \( \chi \). Otherwise, we accumulate the centers of all the members \( C_{ij}^1 \cup C_{ij}^2 = \{c_k | d(c_i, c_k) \leq 1 \land d(c_j, c_k) \leq 1\} \), and move them at the positions \( \{C[k], k = 1, 2, \ldots, m\} \), where \( m = |C_{ij}^1 \cup C_{ij}^2| \). Next, we arrange the members in \( C_{ij}^1 \) and \( C_{ij}^2 \) such that \( C_{ij}^1 = \{C[k], k = 1, 2, \ldots, \mu\} \), and \( C_{ij}^2 = \{C[k], k = \mu + 1, \mu + 2, \ldots m\} \). Consider the bipartite graph \( G_{B_{ij}} \) using \( C_{ij}^1 \) and \( C_{ij}^2 \) as the two sets of vertices, and between a pair of vertices there is an edge if and only if the corresponding disks do not intersect. Compute the maximum matching in the graph \( G_{B_{ij}} \), as defined below.

**Definition 8.1 (Maximum matching)** A matching of a graph \( G = (V, E) \) is a set of edges \( E' \) (\( E' \subseteq E \)) such that no two of edges in \( E' \) have common vertices. A maximum matching is a matching containing maximum number of edges.

Compute the maximum matching \( M \) in the graph \( G_{B_{ij}} \). Compute \( \chi' = |R_{ij}|-
|M|. If |χ'| > |χ|, then replace χ by χ', and remember i, j in a pair of integer locations i*, j*.

Here two points need to be mentioned:

(i) We need to systematically choose (C_i, C_j) among the members in C using a pair of index variable i and j as follows:

- Choose each C_i among the members in C in lexicographic order of their (x, y)-coordinates.
- For each C_i, choose C_j in increasing order of the distance of their corresponding centers c_i and c_j. As soon as C_i and C_j do not intersect in some step, stop considering the next member as C_j.
- For each pair C_i, C_j, C_i ∩ C_j ≠ ∅, we need to compute the largest clique among the disks in C whose centers lie in R_{ij}.

(ii) After considering all the pairs of vertices, we need to execute the same (maximum matching) algorithm for the pair of disks C_{i*} and C_{j*}. To report the largest clique do the following: Report unmatched vertices of the set C_{i*, j*}. For each of these reported vertex, discard all its neighbors among the matched vertices in the other set C_{i*, j*}. For these discarded vertices, we report the matched vertices of the other set. Note that these reported vertices are not connected with any unmatched vertex of C_{i*, j*} (because in that case, we can increase the size of the matching which contradicts). Similar things are to be done for the unmatched vertices of the other set C_{i*, j*}.

In the next subsection, we will show how to compute maximum matching in the bipartite graph G_{B_{ij}}.

8.6.1 Maximum matching in the bipartite graph G_{B_{ij}}

In a matching E', all the edges of E' are called matched edges and all the vertices incident to E' are called matched vertices. The vertices other than the matched vertices are denoted as exposed vertices and the edges other than the matched edges are called free edges. For any (u, u') ∈ E', u and u' are mate
8.6. Graphical clique for unit disk graph

with respect to the matching $E'$. A path $p = (u_1, u_2, u_3, \ldots, u_k)$, where $u_i \in V$ for all $i = 1, 2, \ldots, k$, is called an alternating path if $(u_1, u_2), (u_3, u_4), \ldots$ are free edges and $(u_2, u_3), (u_4, u_5), \ldots$ are matched edges. An alternating path is an augmenting path if it starts and ends at exposed vertices, i.e., if both $u_1$ and $u_k$ are exposed vertices. Observe that if an augmenting path exists, then one can increase the size of the matching by reversing the role of free edges and matched edges of that augmenting path.

As defined in the previous section, $G_{B_{ij}} = (C^1_{ij}, C^2_{ij}, E_{ij})$ is a bipartite graph, where for a pair of vertices there is an edge if the corresponding disks do not intersect. The members in $C^1_{ij}$ and $C^2_{ij}$ are in $C[1, 2, \ldots, \mu]$, and $C[\mu + 1, \mu + 2, \ldots, m]$, respectively. Remember that we do not have to store the edges explicitly as we can check the existence of an edge for a pair of vertices in constant time by computing their distance.

In this section, we show how to compute maximum matching in the bipartite graph $G_{B_{ij}}$ in an in-place manner. We use the well-known augmenting path algorithm for computing maximum matching in the bipartite graph [Papadimitriou 2006]. This algorithm repeatedly finds augmenting path with respect to present matching and increment the size of the matching by 1 in each step. It will stop when no augmenting path is found.

At any instant of time, we use $A^1_{ij}$ (resp. $A^2_{ij}$) to denote the set of matched vertices, and $B^1_{ij}$ (resp. $B^2_{ij}$) to denote the set of exposed vertices in $C^1_{ij}$ (resp. $C^2_{ij}$), $|A^1_{ij}| = |A^2_{ij}| = \alpha$. Thus, $A^1_{ij}[k] = C[k]$ and $A^2_{ij}[k] = C[\mu + k]$, and $A^1_{ij}[k]$ and $A^2_{ij}[k]$ are mate of each other in the present matching, where $k = 1, 2, \ldots, \alpha$. The sets $B^1_{ij}$ and $B^2_{ij}$ consist of $C[\alpha + 1, \alpha + 2, \ldots, \mu]$ and $C[\mu + \alpha + 1, \mu + \alpha + 2, \ldots, m]$, respectively. Note that we do not need extra array for storing $A^1_{ij}$, $A^2_{ij}$, $B^1_{ij}$ and $B^2_{ij}$. For ease of presentation, we will use these terms.

Let $|B^1_{ij}| \leq |B^2_{ij}|$. We consider each exposed vertex $w = C[k] \in B^1_{ij}$ ($k \in \{\alpha + 1, \ldots, \mu\}$) and try to compute an augmenting path with a sequence of matched and free edges that starts at $w$, and ends at an exposed vertex $w' = C[\ell] \in B^2_{ij}$ ($\ell \in \{\mu + \alpha + 1, \ldots, m\}$). If an augmenting path is found, the matching is augmented. The cardinality of both $A^1_{ij}$ and $A^2_{ij}$ are increased by one, and the cardinality of both $B^1_{ij}$ and $B^2_{ij}$ are decreased by one. Otherwise, $w$ becomes
an *useless exposed vertex* in the sense that no augmenting path is possible starting from this exposed vertex (see Theorem 10.5 of [Papadimitriou 2006]). We move \( w \) at the end of the list \( B_{ij}^1 \). We use a variable \( \gamma (\gamma \leq \mu) \) such that the nodes of \( B_{ij}^1 \) stored in \( C[\gamma], C[\gamma + 1], \ldots, C[\mu] \), are *useless exposed vertices*. Initially, \( \gamma \) is set with \( \mu + 1 \). The algorithm stops when there does not exist augmenting path starting from any exposed vertex in \( B_{ij}^1 \), i.e., \( \gamma = \alpha + 1 \). Thus, we need to execute \( \frac{m}{2} \) number of iterations, where \( m \) is the number of vertices in the graph. The procedures for finding an augmenting path from an exposed vertex and updating the matching are stated in the next subsection.

### 8.6.1.1 Computing an augmenting path from an exposed vertex

Let \( w \in B_{ij}^1 \) be an exposed vertex, which is stored in \( C[i], i = \alpha + 1 < \gamma \). Here, we show how to compute an augmenting path (if any) starting from \( w \).

First, we test whether there exists an exposed vertex \( v \in B_{ij}^2 \) such that \((w, v) \in E_{B_{ij}}\). If such a vertex exists (say at location \( C[j], \mu + \alpha < j \leq m \)), then move \( w \) and \( v \) in \( A_{ij}^1 \) and \( A_{ij}^2 \), respectively, by incrementing \( \alpha \) by 1 and executing \( \text{swap}(C[j], C[\mu + \alpha]) \) (see Figure 8.5(a)).

If no such vertex is found, then we search for an augmenting path of length greater than 1. So, we test whether \( w \) has an edge with any member of \( A_{ij}^2 \). If no such vertex is observed, then \( w \) is an *useless exposed vertex* (as mentioned earlier). We move \( w \) at the end of the list \( B_{ij}^1 \) by decrementing \( \gamma \) by 1, and executing \( \text{swap}(C[i], C[\gamma]) \) (see Figure 8.5(b)).

If a vertex \( u = C[\mu + \ell] \in A_{ij}^2 \) is observed such that \((w, u) \in E_{B_{ij}}\), then \( w \to u \to u' \) is an alternating path, where \( u' = C[\ell] \in A_{ij}^1 \), and \((u, u')\) is a matched edge\(^3\). We move \( u \) and \( u' \) at the beginning of the array \( A_{ij}^2 \) and \( A_{ij}^1 \), respectively. We use a scalar variable \( \beta \), initialized as 0. To store the matched edge \((u, u')\) on the alternating path, we increment \( \beta \) by 1, and store \( u(= C[\mu + \ell]) \) and \( u'(= C[\ell]) \ell > \beta \), in \( C[\mu + \beta] \) and \( C[\beta] \), respectively, by executing \( \text{swap}(C[\mu + \ell], C[\mu + \beta]) \) and \( \text{swap}(C[\ell], C[\beta]) \). Note that it is not a problem since both \((C[\beta], C[\mu + \beta])\) and \((C[\ell], C[\mu + \ell])\) are matched edges.

---

\(^3\)i.e., \( u' \) is the mate of \( u \) in the matching discovered till now.
8.6. Graphical clique for unit disk graph

Remark 8.1 The variable $\beta (\neq 0)$ signifies that $(w = C[i], A^2_{ij}[1], A^1_{ij}[1], A^2_{ij}[2], A^1_{ij}[2], \ldots, A^2_{ij}[\beta - 1], A^1_{ij}[\beta - 1], A^2_{ij}[\beta]A^1_{ij}[\beta])$ is an alternating path (see Figure 8.6).

For the vertex $u' = C[\beta]$, first, we try to finish the augmenting path by searching an exposed vertex $w^* \in B^2_{ij}$ such that $(u', w^*) \in E_{ij}$ as we did at the beginning. If such a vertex is found, then we have discovered an augmenting path $w \rightarrow u \rightarrow u' \rightarrow w^*$. If no such $w^*$ is found, then we search for a vertex $v \in A^2_{ij}[\beta + 1, \ldots, \alpha]$ such that $(u', v) \in E_{B_{ij}}$. If such a vertex $v$ is found, then we extend the alternating path with the matched edge $(v, v')$ by incrementing
β by 1, and storing v and v’ in C[µ + β] and C[β], respectively, as stated earlier. However, if no such vertex v is found, then from u’ = C[β] the alternating path can not be extended. In the following discussion, it will be cleared that in this case we can ignore u’.

At this moment, the matched vertices are of following two types:

**Definition 8.2 (Alternating matched vertex)** During the augmentation path finding from the exposed vertex w, a matched vertex is referred to as alternating matched vertex if it is either in C[1, 2, ..., β] or in C[µ + 1, µ + 2, ..., µ + β].

**Definition 8.3 (General matched vertex)** During the augmentation path finding from the exposed vertex w, a matched vertex is called general matched vertex if it has not yet become an alternating matched vertex.
Similarly, at this moment, matched edges are of two types: *alternating matched edge* and *general matched edge*. The following lemma plays a crucial role:

**Lemma 8.3** While finding an augmenting path from an exposed vertex \( w = C[\alpha + 1] \), if it is observed for the first time that from \( u' = C[\beta] \) there is no edge to any one of the exposed vertices and general matched vertices, then the existence of any augmenting path starting from \( w \) containing \( u' \) implies the existence of another augmenting path which does not contain \( u' \).

**Proof:**

Let \( p = (w, u_1, u_1', u_2, u_2', \ldots, u_i, u_i', u, u', u_j, u_j', u_{j+1}, u_{j+1}', \ldots, u_k, u_k', w^*) \) be an augmenting path starting from \( w \) and containing \( u' \) (see Figure 8.7 for illustration). As there is no edge from \( u' \) to any one of the general matched vertices and exposed vertices, so \( u_j, u_j' \) must be an alternating edge.

Without loss of generality, assume that \((u_j, u_j') = (A_{ij}[t], A_{ij}[t])\), where \(1 \leq t < \beta\). Thus, \((w, A_{ij}[1], A_{ij}[1], A_{ij}[2], A_{ij}[2], \ldots, A_{ij}[t-1], A_{ij}[t-1], u_j, u_j')\) is an alternating path (see Remark 8.1) and we have an augmenting path \( p = (w, A_{ij}[1], A_{ij}[1], A_{ij}[2], A_{ij}[2], \ldots A_{ij}[t-1], A_{ij}[t-1], u_j, u_j', u_{j+1}, u_{j+1}, \ldots, u_k, u_k', w^*) \) which does not contain \( u' \).

\[\square\]

![Figure 8.7: Demonstration for the proof of Lemma 8.3](image)

In this context, we define the following:

**Definition 8.4 (Useless matched vertex)** While finding an augmenting path from an exposed vertex \( w \in B_{ij}^1 \), if we observe that no edge is possible from \( u' = C[\beta] \) to any exposed vertices as well as general matched vertices, then the
vertex $u$ is considered as useless matched vertex for the rest of the execution of computing an augmenting path from the exposed vertex $w$.

Note that only an alternating matched vertex can become useless matched vertex.

We use a variable $\delta$, initialized as $\alpha + 1$. When we observe that from the matched vertex $u' = C[\beta]$, no edge is possible to any of the exposed vertex and general matched vertex, then we ignore the matched edge $(u, u') = C[\beta], C[\mu + \beta]$ by doing the following steps (see Figure 8.8 for an illustration):

- decrement $\delta$ by 1,
- swap $(C[\beta], C[\delta])$ and swap $(C[\mu + \beta], C[\mu + \delta])$, and
- decrement $\beta$ by 1.

Next, we try for extending the alternating path from the current alternating matched vertex $C[\beta]$ to any of the exposed vertices. If it fails, then we explore possibility of edges from $C[\beta]$ to any of the general matched vertices. If, it also fails then, we consider $C[\beta]$ as useless matched vertex and follow the above mentioned three steps. This action is justified by Lemma 8.3 and the following lemma.

**Lemma 8.4** While finding an augmenting path from an exposed vertex $w = C[\alpha + 1]$, if it is observed that from $u' = C[\beta]$ there is no edge to any one of the exposed vertices and any one of the general matched vertices, then the existence of any augmenting path starting from $w$ containing $u'$ implies the existence of another augmenting path which does not contain $u'$ and any of the useless matched vertices.

**Proof:**

Let $p = (w, u_1, u'_1, u_2, u'_2, \ldots, u_i, u'_i, u, u', u_j, u'_j, u_{j+1}, u'_{j+1}, \ldots, u_k, u'_k, w^*)$ be an augmenting path starting from $w$ and containing $u'$ (see Figure 8.9 for an illustration). Note that no edge to the exposed vertex $w^*$ is possible from any one of the useless matched vertices and any one of the alternating matched vertices. So, $(u_k, u'_k)$ must be a general matched edge. Let
$u_t, u'_t, u_{t+1}, u'_{t+1}, \ldots, u_k, u'_k$ be the largest sub-path of $p$ which ends at $w^*$ and contains only general matched vertices. Note that $t$ is strictly greater than $j$ because no edge is possible from $u'$ to the general matched vertex $u_t$. As no edge to a general matched vertex is possible from any useless matched vertex,
the only possibility is that \((u_{t-1}, u'_{t-1})\) is an alternating edge.

Without loss of generality, assume that \((u_{t-1}, u'_{t-1}) = (A^2_{ij}[\ell], A^1_{ij}[\ell])\), where \(1 \leq \ell < \beta\). Thus, we have another augmenting path \(p' = (w, A^2_{ij}[1], A^1_{ij}[1], A^2_{ij}[2], A^1_{ij}[2], \ldots, A^2_{ij}[\ell], A^1_{ij}[\ell], u_t, u'_t, \ldots, u_k, u'_k, w^*)\) which does not contain any useless matched vertex and \(w'\).

Note that whenever a useless matched vertex is observed, \(\beta\) is decremented. Finally, if \(\beta = 0\) is observed, we explore other neighbors of \(w\) in \(A^2_{ij}\). If we can complete an augmenting path, say \(w = C[i] \rightarrow A^2_{ij}[1] \rightarrow A^1_{ij}[1] \rightarrow A^2_{ij}[2] \rightarrow A^1_{ij}[2] \rightarrow \ldots \rightarrow A^2_{ij}[\beta - 1] \rightarrow A^1_{ij}[\beta - 1] \rightarrow A^2_{ij}[\beta] \rightarrow A^1_{ij}[\beta] \rightarrow w^* = C[j]\) \((\mu + \alpha < j \leq m)\) where \(w \in B^1_{ij}\) and \(w^* \in B^2_{ij}\), then the matching is updated (augmented) as follows:

Execute \(\text{swap}(C[i], C[k])\) for \(k = 1, 2, \ldots, \beta\), and then execute \(\text{swap}(C[j], C[\mu + \alpha + 1])\). Finally, increment \(\alpha\) by 1 (see Figure 8.10).

If no general matched vertex is left and augmenting path is not produced, then \(w\) is considered to be a useless exposed vertex and it is moved at the end of the array \(B^1_{ij}\) as explained earlier.

8.6.1.2 Complexity of computing the bipartite matching

Lemma 8.5 The maximum matching of the bipartite graph \(G_{B_{ij}} = (C^1_{ij}, C^2_{ij}, E_{ij})\) can be computed in an in-place manner in \(O(m^3)\) time using \(O(1)\) extra-space, where \(m\) is the number of vertices in \(G_{B_{ij}}\).

Proof: Computing an augmenting path from an exposed vertex \(w\) needs at most \(O(m^2)\) time because of the following reasons:

- Initially, each matched edge is a general matched edge.
- Every general matched edge becomes at most once alternating matched edge.
- Each transition from a general matched edge to an alternating matched edge needs at most \(O(m)\) time, because for this we may have to test all the exposed and general matched vertices of the set \(C^2_{ij}\).
Figure 8.10: (a) Augmenting path, and (b) updating the matching

- Every alternating matched vertex becomes useless matched vertex at most once.
- Each transition from an alternating matched vertex to a useless matched vertex needs at most $O(m)$ time, because for this we have to test all the exposed and general matched vertices of the set $C^2_{ij}$.
- Updating the existing matching after finding the augmenting path takes $O(m)$ time in the worst case.
The time complexity follows from the fact that the matching is augmented at most $O(m)$ times. As we use only constant number of variables apart from the input array, the space complexity follows.

\[\Box\]

### 8.6.2 Complexity analysis

**Theorem 8.5** The largest (graphical) clique in the intersection graph $G$ of a given set $C$ of $n$ unit disks in $\mathbb{R}^2$, can be computed in $O(n^2 + m(n + K^3))$ time using $O(1)$ extra-space, where $m$ is the number of edges in $G$ and $K$ is the size of the largest clique in $G$.

**Proof:** The members of the array $C$ gets permuted after each evocation of the maximum matching algorithm. So,

(i) the time needed for choosing each $C_i \in C$ in increasing lexicographic order of the $(x, y)$-coordinates of the members of $C$ needs a linear scan among the members in $C$.

(ii) Similarly, for a $C_i$, choosing each $C_j$ so that $C_i \cap C_j \neq \emptyset$ needs a linear scan among the members in $C$.

(ii) Prior to the execution of bipartite matching algorithm for a pair $(C_i, C_j)$, $C_i \cap C_j \neq \emptyset$, accumulation of all the centers of the disks in $C$ that lie in $R_{ij}$ at the beginning of the array $C$ needs another linear scan among the members in $C$.

The time complexity of in-place computation of maximum matching in the graph $G_{B_{ij}}$ is $O(|C_{ij}^1 \cup C_{ij}^2|^3)$ (see Lemma 8.5). As all the disks in $C_{ij}^1$ (resp. $C_{ij}^2$) form the clique, the size of the maximum clique $K$ in the unit disk graph $G$ must be at least $\frac{|C_{ij}^1 \cup C_{ij}^2|}{2}$. In other words, $|C_{ij}^1 \cup C_{ij}^2| \leq 2K$. So, we can say that the time complexity of the maximum matching in the graph $G_{B_{ij}}$ for any pair of intersecting disks $(C_i, C_j)$ is $O(K^3)$.

The first term of the complexity result corresponds to the total time needed for Step (i) for all the members in $C$, which is $O(n^2)$.  

The second term of the complexity result follows from the fact that $m$ is the number of edges in the graph $G$, and the total time needed for Steps (ii), (iii) and the time required for computing the maximum matching for all the intersecting pair of disks $(C_i, C_j) \in \mathcal{C}$, which is $O(m(n + K^3))$.

\[ \square \]

8.7 Maximum independent set

The following needs to be mentioned regarding the in-place computation of the maximum independent set of some geometric intersection graphs.

For interval graph, we can compute the maximum independent set by adopting the same technique for computing the maximum clique of interval graph. Here the sub-arrays $D$, $H$ and $L$ are maintained at the beginning, middle and end of the input array $I$ using two index variables $\lambda$ and $\mu$. While scanning the intervals from left, after getting a few left-end-points (from $L$), when a right-end-point (from $H$) is observed, that interval is reported as a member of the independent set, all the elements of $H$ are moved to $D$ by setting $\lambda = \mu - 1$. The process continues until all the intervals in $L$ are considered. The time and extra-space complexities will remain same as that of Theorem 8.1.

For rectangle intersection graph of fixed height rectangles, the 2-factor approximation algorithm of Agarwal et al. [Agarwal 1998b] for finding the maximum independent set of the rectangles in $\mathcal{R}$ can easily be made in-place using the in-place method of computing the maximum independent set of a set of intervals on a real line, as stated above. The running time and extra-space requirement of the algorithm will be $O(n \log^2 n)$ and $O(1)$ respectively.

It is easy to see that the 5-factor approximation algorithm of Marathe et al. [Marathe 1995] for computing the maximum independent set problem of disks with arbitrary radii can easily be made in-place by choosing the smallest disk, including it in the independent set, removing (moving them at the end of the array) it and all the disks that overlap on it, and then repeating the same process until all the disks are removed. Similarly, for intersection graph of unit disks, we can get an in-place algorithm for getting a 3-factor approximation result for the maximum independent set problem of unit disk
graph [Marathe 1995] by choosing the disk having center with minimum \(x\)-coordinate, including it in the independent set, removing it and all the disks that overlap on it, and then repeating the same process until all the disks are removed. As each iteration of both the algorithms need \(O(n)\) time, they run in \(O(n^2)\) time using \(O(1)\) extra-space.

### 8.8 Conclusion

In this chapter, we have proposed in-place algorithms for computing the maximum clique of some geometric intersection graphs. For each problem, the input objects are given in an array with minimum amount of space. At the end of the execution, the elements in the array may change their positions, but all the objects are available in the array in some different permutation. For all the problems we have used only \(O(1)\) extra locations apart from the input array.

Our proposed in-place algorithm for bipartite matching problem is of independent interest. To the best of our knowledge there does not exist any space-efficient algorithm for computing the maximum matching in a general bipartite graph in the literature. Our proposed algorithm can be used in the following two environments.

1. Given a bipartite graph \(G = (V_1, V_2, E)\) whose edges are given in the form of a bit-matrix of size \(|V_1| \times |V_2|\) in read-only memory, we can get the maximum matching in time \(O(n^3)\) time using \(O(n)\) extra-space (or equivalently \(O(n \log n)\) extra bits), where \(n = |V_1| + |V_2|\). The reason is that we can test the existence of an edge between a pair of vertices \(u \in V_1\) and \(v \in V_2\) in \(O(1)\) time from the matrix \(E\).

2. The in-place method for computing the maximum matching of the bipartite graph may be used to compute the maximum matching and maximum independent set among a set of axis-parallel line segments in \(\mathbb{R}^2\) [Imai 1983] with \(O(1)\) extra storage.
Chapter 9

Minimum Discrete Piercing Set
for Unit Disks

Contents

9.1 Introduction ................................................. 137
9.2 Approximation algorithms ................................. 138
  9.2.1 Simple $t$-approximation algorithms ............... 139
  9.2.2 Further improving the approximation factor ....... 144
9.3 Polynomial time approximation scheme ................. 153
  9.3.1 Improvement in time complexity .................. 156
9.4 Conclusion .................................................... 157

9.1 Introduction

The piercing set for a set of geometric objects $S$ in $\mathbb{R}^2$ is a set of points $Q$ such that each object in $S$ contains at least one point of $Q$. The optimization problem is to compute a piercing set for $S$ of minimum size. In this chapter, we consider a variation of the piercing set problem for unit disks, called the discrete piercing set problem. Here, a set of points $P$ is given; the objective is to choose minimum number of points from $P$ to pierce all the disks of unit radius centered at the points of $P$. This problem is also known as minimum dominating set problem in the literature, where dominating set of a graph $G = (V, E)$ is defined as a set of vertices $D \subseteq V$ such that all the vertices of $G$ which are not in $D$ must be adjacent to some vertex in $D$. If we consider
a graph with the points of $P$ as vertices and two vertices have an edge if and only if their distance is at most 1, then the vertices corresponding to the points in the (minimum) discrete piercing set corresponds to the (minimum) dominating set of the graph.

As mentioned in Section 2.3.2, this problem is known to be NP-hard [Clark 1990] and a 5-approximation algorithm is known [Carmi 2008]. Our present work is directed towards finding an efficient approximation algorithm. We assume that square-root and ceiling functions are unit cost operations. In Section 9.2, we propose three similar algorithms for this problem. The first one produces a 12-approximation result in $O(n \log n)$ time. The second one produces a 4-approximation solution in $O(n^8)$ time, and the last one produces a 3-approximation solution in $O(n^{15})$ time.

We show that each of these algorithms can be made in-place with $O(1)$ extra-space keeping their time complexities invariant. This is of great advantage in the sense that the algorithms for different optimization problems in unit disk graph have wide applications in sensor network where the necessary softwares are embedded in tiny sensors, and the constraint in memory size is very much important due to the size of the sensors.

In Section 9.3, we propose a polynomial time approximation scheme (PTAS) for the problem which runs in $n^{O(\varepsilon)}$ time.

### 9.2 Approximation algorithms

**Definition 9.1 (Approximation algorithm [Cormen 2009])** Let $P$ be a minimization (resp. maximization) problem. An algorithm $A$ is said to be an $\alpha$-approximation algorithm ($\alpha > 1$) for $P$ if and only if for any instance $X$ of $P$, $A(X)$ runs in time polynomial in $|X|$ (size of $X$) and delivers a feasible solution $SOL(X)$, such that $|SOL(X)| \leq \alpha \times |OPT(X)|$ (resp. $|SOL(X)| \geq \frac{|OPT(X)|}{\alpha}$). Here, $OPT(X)$ denotes the optimum solution of the problem $P$ for the given instance $X$.

We are given a set of points $P$, where each point corresponds to a unit disk centered at that point. The objective of the optimum algorithm for the discrete piercing set problem is to choose minimum number of points of $P$ such
that each disk contains at least one chosen point. In this section, we propose constant factor approximation algorithms for this problem.

9.2.1 Simple $t$-approximation algorithms

In this section, we propose two approximations algorithms for the considered problem. The basic idea of these two algorithms are as follows:

- Partition the plane into basic cells such that all the unit disks centered in a cell can be pierced by choosing any point from that cell.
- Use one point from each cell to pierce all the unit disks.
- Show that a piercing point from a cell can pierce unit disks from at most $t$ cells.
- In the optimum solution, one piercing point can pierce unit disks centered in at most $t$ cells; but we use at most $t$ piercing points for that; thus our algorithm produces a $t$-approximation result.

Using the above argument, we show how to obtain 12-approximation algorithm for the problem. As a warm-up, first, we describe 14-approximation algorithm.

9.2.1.1 A simple 14-approximation algorithm

Consider a partitioning of the plane into a grid whose each cell is of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$. Since the maximum distance between any two points in a grid cell is at most 1, we can pierce all the disks centered in any cell $\chi$ by choosing any one member $p \in P$ lying in that cell. We show that a point $p \in P$ inside a cell may pierce (some or all) unit disks centered in at most fourteen cells.

**Observation 9.1** A point inside a cell can not pierce unit disks centered in more than fourteen cells simultaneously.

**Proof:** Consider the grid structure as shown in Figure 9.1. The length of each side of a cell is $\frac{1}{\sqrt{2}}$. The cells are numbered as 1, 2, ..., 25. The cell 13 is split into four parts, namely $A$, $B$, $C$ and $D$. Let $p$ be a point inside the cell
13. Without loss of generality, assume that \( p \) is in the sub-cell \( A \). Observe that \( p \) may pierce (some or all) unit disks centered in only fifteen cells, numbered 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18 and 19. We can further tighten the observation by showing that \( p \) can not pierce unit disks centered in cell numbered 4 and 16 simultaneously as follows (see Figure 9.1).

Let \( u \) and \( v \) be any two points from the cells 4 and 16, respectively. Thus, \( \text{dist}(u, v) \) is at least 2, where \( \text{dist}(u, v) \) denotes the Euclidean distance between the pair of points \( u \) and \( v \). Therefore, \( \text{dist}(u, p) + \text{dist}(p, v) > 2 \). This implies that at least one of \( \text{dist}(u, p) \) and \( \text{dist}(p, v) \) is greater than 1. Thus, the point \( p \) can not pierce unit disks centered in cell 4 and cell 16 simultaneously. Similar observation holds for any point \( p \in P \) lying in sub-cells \( B, C \) and \( D \). Thus, we have the result.

In our approximation algorithm, we select one point from each non-empty cell. The step-wise description of the proposed method is given in Algorithm 9.1.

**Theorem 9.1** The proposed algorithm is a 14-approximation, and its running time is \( O(n \log k) \), where \( k \) is the size of the optimum solution. Apart from the array containing the input points, the extra-space required for this algorithm is \( O(k) \).

**Proof:** Consider a piercing point in the optimum solution. By Observation 9.1, it can pierce unit disks centered in at most fourteen cells. But, we have chosen at most fourteen different points to pierce those unit disks. Thus, the approximation factor follows.
Algorithm 9.1: MDS_14-FACTOR($P$)

**Input:** Set $P$ of points in a 2-dimensional plane.

**Output:** A set $P^* \subseteq P$ such that the unit disks centered at points in $P^*$ cover all the points in $P$.

1. Consider a partitioning of the plane into a grid whose each cell is of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$.

2. /* A grid cell $(\alpha, \beta)$ is said to be less than another grid cell $(\gamma, \delta)$ if and only if either $\alpha < \gamma$ or $\alpha = \gamma$ and $\beta < \delta$ */

3. We maintain a height balanced binary tree $T$ for storing the non-empty grid cells. Each element of $T$ is a tuple $(\alpha, \beta)$ indicating the indices of a non-empty cell. It is attached with any point $p_i \in P$ that lies in that cell (as the piercing point).

4. Set $P^* \leftarrow \emptyset$.

5. for each point $p_i = (x_i, y_i) \in P$ do

6. compute the indices of the grid cell $\alpha = \lceil \frac{x_i}{1/\sqrt{2}} \rceil$ and $\beta = \lceil \frac{y_i}{1/\sqrt{2}} \rceil$.

7. if the tuple $(\alpha, \beta)$ is not in $T$ then

8. store it in $T$ and attach $p_i$ with it (* Otherwise (i.e., if $(\alpha, \beta)$ is in $T$), we have nothing to do. *)

9. for each node $v$ of $T$ do

10. Let $p$ be the point attached to the node $v$. Set $P^* = P^* \cup \{p\}$

11. Return $P^*$;

In order to justify the time complexity, we shall not construct the grid explicitly. We maintain a height balanced binary tree $T$ for storing the non-empty grid cells. The processing of each point requires only the checking of the corresponding grid cell in $T$. After processing all the points in $P$, we need to visit $T$ for reporting the piercing points. Since the size of $T$ is at most $14k$, the time and space complexity results follow.

Making the algorithm in-place

The algorithm can easily be made in-place as follows. Each cell is associated with its identity number $(\alpha, \beta)$. A point $p = (x, y) \in P$ belongs to the cell $([\frac{x}{1/\sqrt{2}}], [\frac{y}{1/\sqrt{2}}])$. Sort the points in $P$ with respect to the cell in which it falls. While comparing a pair of points $p_i, p_j$, compute the cell identity number in which they fall and comparison of these two cell numbers are done according to the lexicographic order of their corresponding cell numbers (see line 2 of
Algorithm 9.1. Note that the cell number of the points are never stored; these are computed on demand. After sorting, the points in the same cell appear consecutively in the array $P$. Finally, another scan in the array $P$ is required to report one point (in $P$) from each non-empty cell. Thus, the total time required is $O(n \log n)$. Since the size of the optimum solution, $k$, can be $O(n)$ in the worst case, the asymptotic running time of this in-place algorithm is same as that of the earlier one.

9.2.1.2 Improving the approximation factor to 12

Consider the partitioning of the plane using a regular hexagonal lattice as shown in Figure 9.2. Here, each cell is a regular hexagon whose largest diagonal is of unit length. So, all the unit disks centered in a particular cell $\chi$ can be pierced by choosing any point from that cell $\chi$. Our algorithm will take a point $p \in P$ from each non-empty cell and report them as a piercing set. Now, we will prove the following observation:

**Observation 9.2** A point inside a cell can not pierce unit disks centered at more than twelve cells simultaneously.

**Proof:** Consider the partitioning of the plane as shown in Figure 9.2. Here, the cells are numbered as 1, 2, ..., 27. The cell 14 is further divided into 6 sub-cells named as $A, B, C, D, E$ and $F$. Let $p$ be a point inside the cell 14. Without loss of generality, assume that $p$ is in the sub-cell $A$. Now, observe that $p$ can pierce unit disks centered in the fourteen cells numbered as 2, 3, 7, 8, 9, 12, 13, 14, 15, 18, 19, 20, 24 and 25. We will show that $p$ can not pierce unit disks centered in cells 2 and 24 simultaneously. For any two point $u$ and $v$ in cell 2 and 24 respectively, the distance $\text{dist}(u, v) \geq 2$. From triangular inequality, we have $\text{dist}(u, p) + \text{dist}(p, v) \geq \text{dist}(u, v) > 2$. This implies that at least one of $\text{dist}(u, p)$ and $\text{dist}(p, v)$ is greater than 1. Thus, the point $p$ can not pierce unit disks centered at $u$ and $v$ simultaneously. Using similar argument it can be shown that $p$ can not pierce unit disks centered at pair of cells $(3, 24), (2, 25)$ and $(3, 25)$ simultaneously. Therefore, $p$ can not pierce unit disks centered at more than twelve cells simultaneously. Hence we have the result.
Now, we will describe how to implement this algorithm in an in-place manner. As earlier, here too, each cell is associated with its identity number \((\alpha, \beta)\). As in the previous algorithm, we will first sort all the points according to the cells in which they belong and select one point from each non-empty cell. The only difference is in deciding for a point \(p = (x, y) \in P\), the cell in which it belongs.

This can be done by using the following idea. We will virtually assume that the cells are rectangle as shown in Figure 9.3. Each rectangle is of size \(\sqrt{3}/2 \times 3/4\) excepting the starting and ending rectangles of even rows, which are of size \(\sqrt{3}/4 \times 3/4\). For a point \(p = (x, y)\), we will first decide in which rectangle it belongs. Suppose, it is in the \(j\)-th rectangle in the \(i\)-th row. Next, we check whether it is in any of the two triangular pockets of that rectangle. If not, then we assign the point in the cell \((i, j)\). Otherwise, if it is in the left triangle in the \((i, j)\)-th rectangle and \(i\) is even (resp. odd), we assign \(p\) in cell \((i - 1, j - 1)\) (resp. \((i - 1, j)\)). Similarly, if \(p\) is in the right triangle in the \((i, j)\)-th rectangle and \(i\) is even (resp. odd), we assign \(p\) in cell \((i - 1, j)\) (resp. \((i - 1, j + 1)\)).

The details of this cell identification for a point is given as pseudo-code in
Algorithm 9.2. From the pseudo-code, it is clear that this can be done using only constant number of variables in $O(1)$ time. Thus, we have the following theorem:

**Theorem 9.2** The proposed in-place algorithm is 12-approximation; it takes $O(n \log n)$ time and uses only $O(1)$ extra-space.

**Proof:** The approximation factor follows from the Observation 9.2. As we need to sort the points according to their cell identity number and each cell identification takes $O(1)$ time and $O(1)$ extra-space, the running time of the algorithm is $O(n \log n)$ and it uses only $O(1)$ extra-space. 

### 9.2.2 Further improving the approximation factor

In this section, we follow a different scheme for designing approximation algorithms for the considered problem. The general overview of these algorithms is as follows.

- Partition the entire plane into (polygonal) cells and color the cells using $t$ colors, say $c_1, c_2, \ldots, c_t$, such that no unit disk intersects two or more cells of the same color.
Approximation algorithms

Algorithm 9.2: Cell-Identify(p)

**Input:** A point \( p = (x, y) \) ∈ \( P \)

**Output:** The Cell Identity number in which the point \( p \) belongs to

1. \( h = \frac{3}{4}; \ d' = \sqrt{\frac{3}{4}}; \)
2. \( c'_y = \left\lfloor \frac{y}{h} \right\rfloor; \ c'_x = \left\lceil \frac{x}{d'} \right\rceil; \)
3. if \( c'_y \) is even then
   4. \( r = (d'c'_x, hc'_y + \frac{1}{4}); \ s = (d'(c'_x + 1), hc'_y); \)
   else
      6. \( r = (d'c'_x, hc'_y); \ s = (d'(c'_x + 1), h(c'_x + \frac{1}{4}); \)
   if the point \( p \) is above the line passing through the points \( r \) and \( s \) then
      8. \[ c_y = c'_y - 1; \ c_x = \left\lceil \frac{c'_x}{2} \right\rceil; \]
   else
      10. \[ c_y = c'_y; \ c_x = \left\lceil \frac{c'_x}{2} \right\rceil + 1; \]
   else
      12. if \( c'_x \) is even then
         14. \( r = (d'c'_x, hc'_y); \ s = (d'(c'_x + 1), hc'_x + \frac{1}{4}); \)
      else
         16. \( r = (d'c'_x, hc'_y + \frac{1}{4}); \ s = (d'(c'_x + 1), hc'_y); \)
      if the point \( p \) is above the line passing through the points \( r \) and \( s \) then
         18. \[ c_y = c'_y - 1; \ c_x = \left\lceil \frac{c'_x}{2} \right\rceil + 1; \]
      else
         20. \[ c_y = c'_y; \ c_x = \left\lceil \frac{c'_x}{2} \right\rceil; \]
21. Return \((c_x, c_y)\)
- Find a polynomial time exact algorithm $A$ which will take a cell $\chi$ as input and compute minimal subset of points in $P$ to pierce all the unit disks centered in $\chi$.

- Let $OPT_i$ be the set of piercing points obtained by the algorithm $A$ for all the cells colored by $c_i$. Due to the partitioning property, $|OPT_i| \leq |OPT|$, where $OPT$ is the optimum solution for the problem. Thus $\sum_i |OPT_i| \leq t|OPT|$, and we have $t$-approximation algorithm.

### 9.2.2.1 A $4$-approximation algorithm

For $4$-approximation algorithm, we partition the plane into a grid whose each cell is of size $\frac{3}{\sqrt{2}} \times \frac{3}{\sqrt{2}}$ as shown in Figure 9.4(a). Observe that we can color the cells using $4$ colors such that no unit disk can intersect two or more cells of same color.

First, we analyze the time complexity of computing the minimum number of points in $P$ required to pierce all the unit disks centered in a cell $\chi$ of size $\frac{3}{\sqrt{2}} \times \frac{3}{\sqrt{2}}$. We use $P_\chi$ to denote the set of points inside the cell $\chi$. We split $\chi$

![Figure 9.4: Proof of Lemma 9.1](image)

into 9 sub-cells each of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$ (see Figure 9.4(b)). In order to get the minimum cardinality subset of $P$ for piercing the unit disks centered at the points in $P_\chi$, we need to identify the minimum number of points in $P$ such that the unit disks centered at those points can cover all the points in $P_\chi$. Note that we need at most 9 disks to cover all the points in $P_\chi$. The reasons
approximation algorithms

are (i) the disk centered at any point inside a sub-cell covers all the points inside that sub-cell, and (ii) each non-empty sub-cell of $\chi$ can contribute one such point.

Consider the unit width shaded region around the cell $\chi$ (see Figure 9.4(c)). Let $Q_\chi$ be all the points in $P$ lying inside $\chi$ and this shaded region. The points in $Q_\chi$ can pierce the unit disks centered in the cell $\chi$. We choose every point of $p \in Q_\chi$, and check whether all the unit disks centered in $P_\chi$ are pierced by $p$. If it fails for all the points in $Q_\chi$, then we choose each pair of points $p, q \in Q_\chi$ and test whether each unit disks centered in $P_\chi$ is pierced by at least one of the points $p$ and $q$. If it fails again for all possible pair of points, we choose each triple of points of $Q_\chi$ and so on. For all possible 7-tuples, we choose each tuple of 7 points from $Q_\chi$ and test whether each unit disk centered in $P_\chi$ is pierced by any of those 7 points. For each tuple, the checking needs $O(n)$ time. Thus, for checking all possible tuples of size at most 7, the total time needed is $O(n^8)$ in the worst case. If this also fails, then we check for a solution of size 8 by the following manner:

Consider all possible 7-tuples of points from $Q_\chi$. For each 7-tuple $S \subseteq Q_\chi$, do the following:

1. Find the points of $P_\chi$ whose corresponding units disks are not pierced by any of the points in $S$. Let $U \subseteq P_\chi$ be these set of points.

2. Compute the farthest point Voronoi diagram (FVD)\cite{deBerg2008} of $U$.

3. Consider each point $p$ in $Q_\chi \setminus S$ and check whether $p$ pierces all the unit disks centered at $U$ as follows: find the farthest point $f$ of $p$ among the members in $U$ by performing a point location in the FVD of $U$. If the $\text{dist}(p, f) < 1$, then $S \cup p$ is a valid piercing set for all the unit disks centered in $P_\chi$.

Computation of FVD needs $O(n \log n)$ time and point location for each point needs $O(\log n)$ time. Thus, the overall time complexity for obtaining a solution of size 8 (if exists) is $O(n^8 \log n)$.

If there exists no solution of size 8, we can get a solution of size 9 by choosing one point from each non-empty sub-cell of $\chi$. 
Thus, we have the following result:

**Lemma 9.1** The minimum piercing set of the unit disks centered at the points inside a cell $\chi$ of size $\frac{3}{\sqrt{2}} \times \frac{3}{\sqrt{2}}$ can be computed in $O(n^8 \log n)$ time.

**Theorem 9.3** A $4$-factor approximation algorithm for the minimum discrete piercing set problem for unit disks exists with time complexity $O(n^8 \log n)$.

**Proof:** Observe that a unit disk in the optimum solution of a cell $\chi$ does not cover any point of some other cell $\psi$ unless $\psi$ is one of the eight neighboring cells of $\chi$. In Figure 9.4, we give a coloring of the cells with $4$ colors $A$, $B$, $C$ and $D$ such that no unit disk intersect two cells of the same color.

Consider the cells colored by $A$. The set of points in the optimum solution of one cell colored with $A$ do not pierce any unit disk in any other cell colored with $A$ (since $\frac{3}{\sqrt{2}} > 2$). Thus, the optimum solution of the unit disks centered in all the cells colored with $A$ can be computed by choosing each cell colored by $A$ independently, and computing its optimum solution. Let us denote this solution by $OPT_A$. Surely $|OPT_A| \leq |OPT|$, where $OPT$ is the optimum solution for the point set $P$ on the plane. Similarly, $OPT_B$, $OPT_C$ and $OPT_D$ denote the optimum solution of the cells colored as $B$, $C$ and $D$. The approximation factor of our algorithm follows from the fact that $|OPT_A| + |OPT_B| + |OPT_C| + |OPT_D| \leq 4|OPT|$, and our reported solution is $OPT_A \cup OPT_B \cup OPT_C \cup OPT_D$.

The time complexity follows from the facts that (i) each point is considered for computing the optimum solution of at most $9$ cells$^1$, (ii) we compute the optimum solution for only non-empty cells, and (iii) the time complexity result stated in Lemma 9.1.

Making the algorithm in-place

This algorithm can easily be made in-place as follows. As in the earlier section, we sort the points in $P$ according to the cell (identification) number in which they belong. Next, we consider each non-empty cell $\chi$ in an ordered manner,

$^1$points in cell $\chi$ participate in the optimum solution for the cell $\chi$ and other $8$ cells adjacent to cell $\chi$. 

and compute the optimum set of points to pierce all the unit disks centered at $\chi$ as follows.

Note that the cells are placed in order in the array $P$, and the points in each cell are also contiguously placed in $P$. Since the position of $\chi$ is known, we also know the position of 9 cells contributing in the point set $Q_\chi$. We execute a linear pass in the array $P$ to identify the starting and ending positions of each cell of $Q_\chi$ in the array $P$. These starting and ending positions are stored in an array $A$ of size $2 \times 9$. We use another array of size 8 to generate all possible combinations of integers $\{1, 2, \ldots, k\}$, where $k = \sum_{j=1}^{9} (A[2, j] - A[1, j] - 1)$ = total number of points in these 9 cells. For each generated combination, we can identify the corresponding points $Q' \in P$ by using the array $A$. Again, a linear scan among the members in $P_\chi$ is needed to test whether the corresponding unit disks are pierced by the chosen points in $Q'$. As we do not know any in-place way to compute the farthest point Voronoi diagram, we substitute the steps mentioned earlier for getting a solution of size 8 as follows.

Consider all possible 7-tuples of points from $Q_\chi$. For each 7-tuple $S \subseteq Q_\chi$, do the following

1. Find the points of $P_\chi$ whose corresponding unit disks are not pierced by any of the points of $S$. Let $U_1, U_2, \ldots, U_9$ be these points in the 9 sub-cells. Thus $U = U_1 \cup U_2 \cup \ldots \cup U_9$. Some of the $U_i$'s may be empty. Move $U_i$ at the beginning of the corresponding sub-cell in an in-place manner.

2. Compute convex hulls $C_1, C_2, \ldots, C_9$ for point sets $U_1, U_2, \ldots, U_9$, respectively, in an in-place manner [Brönnimann 2004b, Vahrenhold 2012]. The hull vertices are stored at the beginning of the corresponding sub-cells.

3. Consider each point $p$ in $Q_\chi \setminus S$ and check whether $p$ pierce all the unit disks centered in $U$ as follows:
   
   - Find farthest point $f_i$ of $p$ in each non-empty $U_i$ by performing a binary search among the vertices in $C_i$. Let $d_i = \text{dist}(p, f_i)$, where $U_i$ is non-empty, otherwise $d_i = 0$, $i \in \{1, 2, \ldots, 9\}$.
   - if $\max\{d_1, d_2, \ldots, d_9\} \leq 1$, then $\{p\} \cup S$ is a solution.
This step takes $O(n^8 \log n)$ time and $O(1)$ extra-space since in-place convex hull algorithm needs $O(n \log n)$ time and $O(1)$ extra-space [Brönnimann 2004b, Vahrenhold 2012].

If all these efforts fail, then choosing one point from each of the 9 sub-cells of $\chi$ can also be performed by a linear scan in the portion of the array $P_{\chi}$. Thus, the overall time complexity remains $O(n^8 \log n)$ in the worst case.

9.2.2.2 Improving the approximation factor to 3

As in the earlier sections, here too we need to partition the plane into cells. We split the plane into horizontal strips of height $\frac{3}{\sqrt{2}}$. Each odd numbered strip is divided into equal sized cells of width $\frac{6}{\sqrt{2}}$. The horizontal width of the last cell may be less than $\frac{6}{\sqrt{2}}$, depending on the horizontal width of the region. Each even numbered strip is divided into cells such that the first cell is of width $\frac{3}{\sqrt{2}}$, and the other cells are of width $\frac{6}{\sqrt{2}}$, excepting the last cell as mentioned for odd numbered strips.

![Figure 9.5: Coloring the cells for 3-factor approximation algorithm](image)

In Figure 9.5, we give a coloring of the cells using 3 colors. Observe that such a coloring has the property that no point can pierce unit disks centered in two or more cells of the same color.

The maximum number of points required to pierce all the unit disks centered in a cell $\chi$ of size $\frac{3}{\sqrt{2}} \times \frac{6}{\sqrt{2}}$ is 18 as we can split $\chi$ into 18 sub-cells each of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$. As described in previous subsection, we can show that to compute the minimum number of points from $P$ to pierce all the unit disks centered in $\chi$ needs at most $O(n^{17} \log n)$ time and it can be implemented in an in-place
manner using $O(1)$ extra-space. Thus, we have the following result:

**Theorem 9.4** A 3-approximation in-place algorithm for the minimum discrete piercing set problem for unit disks exists with time complexity $O(n^{17} \log n)$.

**Improving the time complexity**

Here, we split the plane into horizontal strips of width 2 such that no point in $P$ lies on the boundary of a strip. Next, we split each strip of width 2 into cells of size $4 \times 2$ (similar to Figure 9.5). Here too, we can color all the cells using 3 colors such that no point can pierce unit disks centered in two or more cells of same color.

Consider a cell $\chi$ of size $4 \times 2$ in the Euclidean plane. Let $P_\chi$ be the set of points in $P$ lying inside $\chi$. We now give an upper bound on the minimum number of points in $P$ required for piercing the unit disks centered at the points of $P_\chi$. Note that

- the set of points $Q_\chi$ that pierce the unit disks centered at $P_\chi$ lie inside a rectangle $R'$ of size $6 \times 4$ whose each of the four sides is 1 unit away from the corresponding boundary of $\chi$, and

- if $O \subseteq P_\chi$ (resp. $O' \subseteq Q_\chi$) be the minimum cardinality subset of points in $P_\chi$ (resp. $Q_\chi$) required to pierce the unit disks centered at $P_\chi$, then $|O'| \leq |O|$.

Consider the graph $G = (V, E)$, where vertices of $V$ correspond to the points in $P_\chi$ and $E = \{(v_i, v_j) | v_i, v_j \in V \& disks of radius \frac{1}{2} centered at the points corresponding to v_i and v_j intersect\}$. Therefore, finding $O$, the minimum cardinality subset of points in $P_\chi$ for piercing the unit disks centered at the points in $P_\chi$ is equivalent to finding minimum dominating set $\gamma(G)$ of the graph $G$ (as discussed in Section 9.1). Independent set of a graph $G = (V, E)$ is a set of vertices $I (\subseteq V)$ such that no two vertices in $I$ are adjacent to each other. Maximum independent set $I(G)$ of a graph $G$ is an independent set of maximum size. Minimum dominating set and maximum independent set are closely related as stated in the following lemma.
Lemma 9.2 [Allam 1978] For an arbitrary graph $G$, $|\gamma(G)| \leq |I(G)|$, where $\gamma(G)$ and $I(G)$ are the minimum dominating set and maximum independent set respectively.

We use $\Gamma$ and $I$ to denote the set of disks corresponding to the vertices in $\gamma(G)$ and $I(G)$ respectively.

Let $R''$ be the rectangular region of size $5 \times 3$ whose each of the four sides is $\frac{1}{2}$ unit away from the corresponding boundary of $\chi$. If a set of disks $H$ of radius $\frac{1}{2}$ can be placed inside a rectangle $R''$ where no two of these disks are mutually overlapping, but may be tangent to each other, then we state that the set of disks $H$ can be packed inside the rectangle $R''$. By an optimal packing, we mean a packing of maximum size. Let $\Psi$ be the set of disks of radius $\frac{1}{2}$ of maximum cardinality (not necessarily centered at the points in $P_\chi$) that can be packed in the region $R''$. We now have the following result.

Lemma 9.3 $|I| \leq |\Psi|$.

**Proof:** Note that if a disk lies completely inside $R''$ then its center lies inside $\chi$ or on the boundary of $\chi$. For contradiction, let $|I| > |\Psi|$. Since the disks in $I$ are non-intersecting, all the members in $I$ can be packed in $R''$. Thus, we have a contradiction since $|\Psi|$ is the maximum number of disks of radius $\frac{1}{2}$ that can be packed in $R''$. $\square$

Lemma 9.2 and 9.3 along with the above discussions imply that

$$|O'| \leq |O| = |\gamma(G)| \leq |I(G)| \leq |\Psi|.$$ 

In [Casado 2001], it is proved that the size of the optimal packing of a $10 \times 10$ square with unit disks is 25. They also provided a pictorial illustration of the corresponding packing (see Figure 9.6). It shows that the size of the optimal packing of a $10 \times 6$ rectangle is 15. The reason is that if we could pack the $10 \times 6$ rectangle with more than 15 unit radius disks, then we could improve the size of the optimal packing of the $10 \times 10$ square. This contradicts the claim stated in [Casado 2001]. Thus, the number of non-overlapping disks of radius $\frac{1}{2}$ that can be put in a rectangle of size $5 \times 3$ can be at most 15 (see Figure 9.6). Therefore, we have the following result.
9.3 Polynomial time approximation scheme

Definition 9.2 (Polynomial time approximation scheme [Cormen 2009])

Let $P$ be a minimization (resp. maximization) problem. An algorithm $A$ is a polynomial time approximation scheme (PTAS) for $P$ if and only if for any instance $X$ of $P$ and for any (fixed) $\varepsilon > 0$, $A(X, \varepsilon)$ runs in time polynomial in $|X|$ and delivers a feasible solution $\text{SOL}(X, \varepsilon)$, such that $|\text{SOL}(x, \varepsilon)| \leq (1 + \varepsilon) \times |\text{OPT}(X)|$ (resp. $|\text{SOL}(X, \varepsilon)| \geq \frac{|\text{OPT}(X)|}{1 + \varepsilon}$). Here, $\text{OPT}(X)$ denotes the optimum solution of the problem $P$ for the given instance $X$.

Given a point set $P$ in $\mathbb{R}^2$ and a fixed integer $k \geq 1$, we design polynomial time approximation scheme (PTAS) for our discrete piercing set problem by using two nested applications of shifting strategy, proposed by Hochbaum and Mass [Hochbaum 1985].

We first split the plane into horizontal strips of width 2 and then split the plane into vertical strips of width 2. These will be referred as elementary
horizontal strips and elementary vertical strips, respectively. In the first level, we apply shifting strategy in horizontal direction by executing $k$ iterations as follows. In the $i$-th iteration of this level, decompose the entire region into disjoint horizontal slabs such that the first slab is of width $2i$ and all other slabs are of width $2k$. We then consider only those slabs $H$ having at least one point, and compute the solution by applying the shifting strategy in the vertical direction by executing $k$ iterations at the second level. Its $j$-th iteration splits the slab $H$ into rectangular cells such that the first cell is of size $2j \times h$ and all other cells are of size $2k \times h$, where $h$ is the width of the slab $H$; $h = 2i$ for the first slab and $h = 2k$ for all other slabs. We use elementary horizontal strips to split the region into strips at the first level, and then the elementary vertical strips to get the cells at the second level. We use the notation $(i,j)$-th iteration to denote the $i$-th iteration at the first level and $j$-th iteration at the second level.

We now explain the method of computing the optimum solution in each cell of size $2k \times 2k$. For a smaller cell also the same method works by conceptually extending it to a size $2k \times 2k$. We partition a cell of size $2k \times 2k$ into $\left(\lceil 2\sqrt{2k} \rceil\right)^2$ sub-cells each of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$. Since any point inside a sub-cell $\rho$ of size $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$ can pierce all unit disk centered in that sub-cell, we can say that maximum number of disk required to pierce all the unit disks centered inside the cell of size $2k \times 2k$ is $\left(\lceil 2\sqrt{2k} \rceil\right)^2$. Therefore, we have to check at most $O(n_{\chi} \left(\lceil 2\sqrt{2k} \rceil\right)^2)$ combinations of the points for getting the optimum solution for piercing all unit disks centered in a cell $\chi$ of size $2k \times 2k$, where $n_\chi$ is the number of points in $P$ whose corresponding disk has a portion in the cell $\chi$ (see Figure 9.4(c)). Since the points in $P$ centered in a cell are disjoint from that of the other cells, and a point in $P$ can participate in computing the optimum solution of at most 9 cells, we have the following result:

**Lemma 9.5** The total time required for the $(i,j)$-th iteration of the algorithm is $O(n_{\chi}(\lceil 2\sqrt{2k} \rceil)^2)$.

In the $(i,j)$-th iteration, the feasible solution is the union of the optimum solution of each cell constructed in that iteration. Finally, the algorithm returns the minimum among these $k^2$ feasible solutions obtained in $k^2$ iterations.
Theorem 9.6 Given a set $P$ of $n$ points in $\mathbb{R}^2$ and an integer $k \geq 1$, the proposed algorithm delivers a discrete piercing set of the disks centered at all the points of $P$ in $O(k^2n([2\sqrt{k}])^2)$ time, whose size is at most $(1 + \frac{1}{k})^2 \times \text{OPT}$, where $\text{OPT}$ is the size of the optimum solution.

Proof: We use the same technique of [Hochbaum 1985] to analyze the approximation factor of our algorithm. Let $\text{OPT}$ be the optimum solution, $\text{OPT}_{ij} \subseteq \text{OPT}$ that intersect the boundary lines of all the cells considered in the $(i,j)$-th iteration. If $S_{ij}$ is the feasible solution obtained by our algorithm in the $(i,j)$-th iteration, then $S_{ij} \leq \text{OPT} + \text{OPT}_{ij}$. Summing over all $(i,j)$, we have

$$\sum_{i=1}^{k} \sum_{j=1}^{k} S_{ij} \leq k^2 \text{OPT} + \sum_{i=1}^{k} \sum_{j=1}^{k} \text{OPT}_{ij}.$$ 

Since no disk of unit radius is intersected by two elementary horizontal strips (resp. elementary vertical strips), and each horizontal (resp. vertical) boundary appears at most $k$ times throughout the algorithm, we have

$$\sum_{i=1}^{k} \sum_{j=1}^{k} \text{OPT}_{ij} \leq k\text{OPT} + k\text{OPT}.$$ 

Thus,

$$\sum_{i=1}^{k} \sum_{j=1}^{k} S_{ij} \leq k^2 \text{OPT} + 2k\text{OPT} = (k^2 + 2k)\text{OPT}.$$ 

Denoting by $S_{av}$ and $S_{min}$ the average and minimum values of $S_{ij}$ considering all the iterations, we have $S_{min} \leq S_{av} \leq (1 + \frac{1}{k})^2 \text{OPT}$.

The time complexity result follows from Lemma 9.5, and the fact that we need to execute $k^2$ number of iterations.

The proposed algorithm can be implemented in an in-place manner using an array of size $([2\sqrt{k}])^2$ for generating the different combinations of indices of the points inside a cell. It also needs three scalar locations $I, J, S$, where $S$ stores the minimum solution obtained so far, and $(I, J)$ stores the iteration number when the minimum solution is obtained.
9.3.1 Improvement in time complexity

We can improve the time complexity using a divide and conquer strategy as stated below using the following result.

**Lemma 9.6** Let \( \chi \) be a cell of size \( k \times k \), and \( L \) be a vertical line that properly intersects \( \chi \). The number of points required to pierce all the unit disks which intersect \( L \) and whose center is to the left (resp. right) of \( L \) is at most \( 5k \).

**Proof:** Consider a vertical strip of width 2 inside \( R \) whose right (resp. left) boundary is the line \( L \). Split it into squares of size \( \frac{3}{2} \times \frac{3}{2} \). All unit disks centered in such a square can be pierced by any point of \( P \) inside that square. The result follows from the fact that the total number of squares in each of these two strips is \( \lceil 3 \times 3 \times \frac{k}{2} \rceil \leq 5k \) as \( k \geq 1 \).

Let \( Q \) be the set of points of \( P \) inside a cell \( \chi \) of size \( k \times k \). Consider a vertical line \( V \) and a horizontal line \( H \) that split \( R \) into four sub-cells each of size \( \frac{k}{2} \times \frac{k}{2} \). Let \( Q_{VH} \) be the points in \( Q \) whose corresponding unit disks intersect the \( V \) and/or \( H \), and \( Q_{\ell a}, Q_{\ell b}, Q_{ra}, Q_{rb} \) be the set of points of \( Q \) in the four quadrants whose corresponding unit disks do not intersect \( V \) and \( H \).

Note that (i) the size of the optimum discrete piercing set for \( Q_{VH} \) is at most \( 20k \) (see Lemma 9.6), and any subset of \( Q_{VH} \) may appear in the optimum solution for \( R \), (ii) \( Q_i \cap Q_j = \emptyset \), \( i, j \in \{\ell a, \ell b, ra, rb\} \), and (iii) any unit disk centered in the optimum solution of any one of the four quadrants do not intersect \( V \) and \( H \). Thus, we can adopt the following divide and conquer strategy to compute the optimum solution inside \( \chi \).

Consider all possible subsets \( Q' \subseteq Q_{VH} \) of size at most \( 20k \). For each of these choices, we do the following in each quadrant \( i \in \{\ell a, \ell b, ra, rb\} \): delete all the points in \( Q_i \) whose corresponding unit disks are already pierced by any of the points of \( Q' \); let \( \hat{Q}_i \subseteq Q_i \) be the remaining set of points. Now compute the optimum solution \( Q'_i \subseteq Q_i \) for piercing the unit disks centered in \( \hat{Q}_i \) recursively using the same procedure. Next, check whether \( Q' \cup Q'_{\ell a} \cup Q'_{\ell b} \cup Q'_{ra} \cup Q'_{rb} \) pierce all the unit disks centered at \( Q \). Thus, the number of combinations of points to be chosen for testing for an optimum solution follows the recursion relation \( T(n, k) = 4 \times T(n, \frac{k}{2}) \times n^{20k} = n^{O(k)} \). This leads to the following result:
Theorem 9.7 Given a set $P$ of $n$ points in $\mathbb{R}^2$ and an integer $k \geq 1$, the proposed algorithm delivers a discrete piercing set of the disks centered at all the points of $P$ in $n^{O(k)}$ time, whose size is at most $(1 + \frac{1}{k})^2 \times OPT$, where $OPT$ is the size of the optimum solution.

It needs to be mentioned that we could not suggest a scheme of implementing this recursive algorithm in an in-place manner using sub-linear extra-space whose size is not a function of the input size $n$.

9.4 Conclusion

We propose constant factor approximation algorithms for a variation of the discrete piercing set problem for unit disks, where the points chosen for piercing the disks are from the set of center points of the disks given for piercing. The simplest one produces $12$-approximation result in $O(n \log n)$ time. We then improve the approximation factor to 4. Finally, we propose a 3-factor approximation algorithm. Thus assuming that square-root and ceiling operations are unit time computable, our algorithm is an improvement of the current best known result by a factor of $\frac{5}{3}$ [Carmi 2008]. The method of designing approximation algorithms used in this paper may be useful for some other problems. All these algorithms can be made in-place with $O(1)$ extra-space without any degradation in their asymptotic running time. The algorithms proposed in [Mustafa 2010, Gibson 2010] can be tailored to get a PTAS for our problem in $n^{O(\frac{1}{\varepsilon})}$ time. We show that we can improve the time complexity of the PTAS for our problem to $n^{O(\frac{1}{\varepsilon})}$. 
Finally, in this chapter, we summarize the main contribution of this thesis and state possible future directions of research.

10.1 Space-efficient tools

In this thesis, we propose some important space-efficient tools which can be used widely. These are:

10.1.1 In-place priority search tree

Priority search tree, a classic data structure for storing points in $\mathbb{R}^2$, is an extensively used data structure to solve different problems. We show that this data structure can be made in-place, i.e., given a set of points in $\mathbb{R}^2$ in
an array, we can construct this data structure in the given array itself in an
in-place manner without using any extra-space. However, the algorithm uses
$O(1)$ extra-space for the execution. We also show that using this in-place data
structure, one can answer all the standard query operations on priority search
tree in same time bound as in the original priority search tree [McCreight 1985]
but using only $O(1)$ extra-space. Using this in-place priority search tree, we
show how to enumerate all maximal empty axis-parallel rectangles amongst a
points in $\mathbb{R}^2$ in an in-place manner.

10.1.2 Space-efficient prune-and-search

Prune-and-search is an important paradigm to solve different problems. In
this thesis, we give a general prune-and-search scheme and show how to im-
plement it in both in-place and read-only model. For examples, we consider
the following important geometric optimization problems which accept prune-
and-search solutions.

**Minimum enclosing circle** Minimum enclosing circle for a set of points
$P$ in $\mathbb{R}^2$ is a circle of minimum radius that encloses all the points of $P$. Min-
imum enclosing circle has a vast application in different areas like planning
the location of sensor device, gas station, hospital, school etc.. The best
known traditional algorithm for this problem takes $O(n)$ time and $O(n)$ space
[Megiddo 1983b]. Asano et al [Asano 2011b] proposed a space-efficient algo-

ithm which runs in $O(n^2)$ time using $O(1)$ extra-space considering the input
as read-only. They stated the possibility of an algorithm with sub-quadratic
running time poly-logarithmic extra-space in read-only memory/environment
as open. Using our space-efficient prune-and-search scheme, one can com-
pute the minimum enclosing circle in (i) linear time using constant number
of extra-space in an in-place manner, and (ii) sub-quadratic time using only
poly-logarithmic extra-space when the input is given in a read-only memory.

**Convex hull** Convex hull is a basic structure which is extensively used in
computational geometry. We show that using our space-efficient prune-and-
search approach, one can compute convex hull for a set of $n$ sorted points in
$\mathbb{R}^2$ in sub-quadratic time and using poly-logarithmic extra-space. Here, the
input is considered in a read-only array.
10.2. Space-efficient algorithms for geometric intersection graphs

**Low-dimensional linear programming** Linear programming is an extensively used tool to solve optimization problems. Megiddo proposed linear time algorithm which takes linear amount of space for fixed dimensional linear programming problem [Megiddo 1984]. In the space-efficient context, Chan and Chen [Chan 2007] proposed an $O(n)$ time randomized algorithm which takes $O(\log n)$ extra-space assuming that the input is given in a read-only array. For two and three dimensional linear programming, we propose (i) $O(n)$ time $O(1)$ extra-space in-place deterministic algorithm, and (ii) sub-quadratic time poly-logarithmic extra-space deterministic algorithm assuming the input in a read-only array.

10.1.3 Maximum matching for bipartite graph

Maximum matching for general bipartite graph is a very important problem both for its huge practical and theoretical implications. Matching of a bipartite graph $G = (V_1, V_2, E)$ is a set of edges $E' (\subseteq E)$ such that no two edges of $E'$ have common vertices. Maximum matching is a matching of maximum size. Assuming that the connectivity between two vertices can be checked in $O(1)$ time, we propose an in-place algorithm for this problem. Our algorithm takes $O(n^2)$ time and $O(1)$ extra-space, where $n$ is the number of vertices of the graph. To the best of our knowledge, this is the first in-place algorithm for the problem.

10.2 Space-efficient algorithms for geometric intersection graphs

The geometric intersection graph $G = (V, E)$ for a set of geometric objects $S$ is a graph whose vertices $V$ correspond to the set of objects in $S$. Between a pair of vertices $v_i$ and $v_j$, there is an edge $(v_i, v_j)$ if the corresponding objects in $S$ intersect. The intersection of a pair of objects is defined depending on the problem specification. For example, sometimes proper containment is considered to be an intersection and sometimes it is not. In this thesis, we consider space-efficient implementation of the following optimization problems for geometric intersection graphs.
10.2.1 Maximum clique of geometric intersection graphs

Clique of a graph $G = (V, E)$ is defined as a set $C \subseteq V$ such that for all pair of members in $C$, there is an edge in $E$. Apart from theoretical interest, clique has huge practical applications. We study maximum clique finding problem for different geometric intersection graphs like of axis-parallel rectangles and disks in $\mathbb{R}^2$ in space-constrained environment. For finding the maximum clique of the intersection graphs of a set of axis-parallel rectangles of arbitrary sizes, our proposed in-place algorithm runs in $O(n^2 \log n)$ time. For the rectangle intersection graph of fixed height rectangles, we show that the time complexity can be slightly improved to $O(n \log n + nK)$, where $K$ is the size of the maximum clique, which may be $O(n)$ in the worst case. Thus, the asymptotic time complexity becomes $O(n^2)$. We consider two variations of the maximum clique problem for disk graphs, namely geometric clique and graphical clique. The time complexity of our algorithm for finding the largest geometric clique is $O(n^2 \log n)$, and it works for disks of arbitrary radii. For graphical clique, our proposed algorithm works for unit disks (i.e., of same radii) and the worst case time complexity of our algorithm is $O(n^2 + mK^3)$ where $m$ is the number of edges in the unit disk intersection graph, and $K$ is the size of the largest clique in that graph. It uses $O(n^3)$ time in-place algorithm for maximum matching of a bipartite graph where vertices of two sets are given in an array, and the existence of an edge between a pair of vertices can be checked on demand in $O(1)$ time. It needs to be mentioned that the $(time \times extra-space)$ is less than or equal to the best-known results for all the problems we have considered excepting the graphical clique of unit disk graph.

10.2.2 Minimum discrete piercing set for unit disks

A set of points $P$ in $\mathbb{R}^2$ is given. A unit disk is centered in each of the points in $P$. We have to choose minimum number of points from $P$ to pierce all the units disks. This problem is alternatively known as minimum dominating set for unit disk graph. The problem is NP-complete [Clark 1990]. We propose easy to implement space-efficient approximation algorithms for this problem. First, we show how one can obtain 12-approximation result in $O(n \log n)$ time using $O(1)$ extra-space in an in-place manner. Next, we propose two in-place
algorithms that produce 4-approximation and 3-approximation results, respectively. The time complexities are $O(n^8 \log n)$ and $O(n^{15} \log n)$, respectively, and both of these algorithms use $O(1)$ extra-space for execution. Finally, we propose a polynomial time approximation scheme for the problem which takes $n^{O(\frac{1}{\varepsilon})}$ time and produces $(1 + \varepsilon)^2$-approximation result, where $0 < \varepsilon < 1$.

10.3 Conclusion

For all of the considered problems, one can study whether further improvement of $time \times extra-space$ is possible. In other words, to prove the lower-bound of $time \times extra-space$, for each of the considered problems, is a challenging direction of future work.


Publications related to this thesis

Refereed Journals


(J2) M. De, G. Das, P. Carmi, and S. C. Nandy, Approximation algorithms for the discrete piercing set problem for unit disks, Accepted for publication in *Int. J. on Computational Geometry and Applications*.

Refereed Conference Proceedings


Informal Publications