Prune-and-Search with Limited Work-space [*]

Minati De[1] Subhas C. Nandy[2] Sasanka Roy[3]

[1] The Technion– Israel Institute of Technology, Haifa 32000, Israel (The work was done while the author was at Indian Statistical Institute, Kolkata)
[2] Indian Statistical Institute, Kolkata 700108, India
[3] Chennai Mathematical Institute, Chennai - 603103, India

Abstract

Prune-and-search is an excellent algorithmic paradigm for solving various optimization problems. 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 which have several advantages compared to the traditional model of computation. Our technique can be applied to a large number of problems which accept prune-and-search. 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 n points in $\mathbb{R}^2$, and
- linear programming problems with two and three variables and n constraints.

In the in-place setting, all these problems can be solved in $O(n)$ time using $O(1)$ extra-space. In the read-only setup, the time and extra-space complexities of the proposed algorithms for all these problems are $O(n\ polylog(n))$ and $O(polylog(n))$, respectively.

1 Introduction

Prune-and-search is an excellent algorithmic paradigm. It was first invented by Megiddo in 1983 [17]. Since then it has been widely used 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.

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.

[*]The preliminary version of a part of this work appeared in the proceedings of 32nd Foundation of Software Technology & Theoretical Computer Science (FSTTCS, 2012).

1.1 Motivation for Space-efficient Models

Designing algorithmic tools with fast and limited size memory (e.g caches) but having capability of very fast processing of a massive high quality data is an active field of research [2, 3, 4, 5]. The problem becomes much more difficult if the input data is given in a read-only array, and very small amount of work-space is available in the system. 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 [3].

Space-economic algorithms have many advantages compared to traditional algorithms [7, 8]. As they use only a very small amount of extra-space during their execution, a larger part of the data can be kept in the faster memory. As a result, the algorithm becomes much more faster than the traditional algorithm.

There are tremendous applications of space-economic algorithms in data-streaming and data-mining problems. Here, huge amount of data is available for processing. So, apart from storing the input data, few work-space is available for the computation. In other areas also, the space-efficient algorithms are important in spite of the fact that computer hardware has become extremely cheap now-a-days. For example, increasing interest of using tiny portable devices (for example, sensors, GPS systems, mobile hand-sets, small robots, etc.) with multipurpose activity forces the manufacturers to make devices as tiny as possible. As a result of that those devices have small amount of memory as well. Here space-efficient algorithms are only the solution.

Readers are referred to [5, 6] for the in-place algorithms of several other geometric problems. For the geometric algorithms in the read-only setup, see [1, 2].

1.2 Considered Problems

The work in this paper is motivated from an open question of Asano et al. [2], which asks the possibility 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 with sub-linear work-space. In this paper, we consider the following problems which can be solved by prune-and-search technique.

Minimum enclosing circle: Given a set of n points $\mathbb{R}^2$ in a read-only array of size n, an algorithm for computing the minimum enclosing circle (MEC) of the point-set P. The complexity results of the proposed algorithms are (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, and (ii) $O(n \log^5 n)$ time using $O(\log^2 n)$ extra-space.

Linear programming: Solving linear programming problem with two (resp. three) variables and n constraints, where the constraints are given in a read-only array.

The complexity results of the proposed algorithms for the linear programming with two variables are: (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, and (ii) $O(n \log^4 n)$ time with $O(\log^2 n)$ extra-space.

For the linear programming with three variables, the complexity results are (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 $O(n \log^5 n)$ time with

$O(\log^2 n)$ extra-space.

Throughout the paper, we assume that random access of the input array elements is permitted, and each location of the work-space consists of $\log n$ bits.

As a warm-up, we first propose an $O(n)$ time in-place algorithm for the MEC problem using $O(1)$ extra-space, where apart from random accessing of the array elements, swapping elements in the array is permissible. For all other problems, similar linear time in-place algorithm can be designed with $O(1)$ extra-space.

1.3 Literature survey

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 π 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 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. It was first proposed by Sylvester [26] in the year 1857. Elzinga et al. with their work [11] paved the way for solving minimax problems 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 the 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 [23], 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 [17]. It uses a prune-and-search technique; the time and space complexities of this algorithm are both $O(n)$. Later Welzl [27] 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 best-known result is also by Megiddo [18] that runs in $O(n(\log n)^3(\log\log n)^2)$ time using the parametric search [16]. Later, Megiddo and Zemel [21] 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 [20]. All of these algorithms use $O(n)$ extra-space. Asano et al. [2] proposed an $O(n^2)$ time and $O(1)$ extra-space algorithm for computing the vertices of $FVD(P)$, where the points in the set P are given in a read-only array. Needless to say, the same time complexity holds for computing the minimum enclosing circle of the point set P. The authors of [2] proposed the possibility of computing minimum enclosing circle of a set of n points in $\mathbb{R}^2$ given in a read-only array in sub-quadratic time with sub-linear work-space as an open problem.

Needless to mention that linear programming problem can be solved in time polynomial in the number of constraints and number of variables [12, 13]. If the number of variables d is fixed, then the problem can be solved in $O(n)$ time, where n is the number of constraints [19]. The most useful one is a randomized algorithm proposed by Seidel [25]. If the number of variables d is fixed, then it works in $O(n)$ time. Bronnimann et al. [7] showed that the linear programming with two variables and n constraints can be solved in an in-place manner in $O(n)$ time with $O(1)$ extra-space. Chan and Chen [9] 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. To the best of our knowledge, no sub-quadratic time deterministic algorithm with sub-linear extra-space is available for low-dimensional linear programming in the literature. Although Chan and Chen's [9] linear time randomized algorithm works for the linear programming problem in fixed dimension using $O(\log n)$ extra-space in a read-only environment, but it is not very clear how to modify that algorithm to obtain same time and space complexity result for computing the minimum enclosing circle of a set of n points in R^2 given in a read-only array.

In this context, it needs to be mentioned that the prune-and-search technique of Kirkpatrick and Seidel [14] for computing the convex hull of a planar set of points can be tailored in memory constrained environment to compute the convex hull of a sorted set of points stored in a read-only array in $O(n \log^5 n)$ time using $O(\log^2 n)$ extra-space [10]. The convex hull of a simple polygon with n vertices can be computed in a read-only setup in $O(\frac{n \log n}{\log p})$ time with $O(\frac{p \log n}{\log p})$ extra-space, or $O(\frac{n^2}{2^S})$ time with $O(S)$ extra-space, where $S \in o(\log n)$ [3]. It is noticed that this can be used for sorting a set of points sorted with respect to x-coordinates in a read-only array with the same time complexity [15].

1.4 Organization of the paper

The general way to implement prune-and-search algorithms in space-efficient manner is described in Section 2. Space-efficient computation of minimum enclosing circle and low dimensional linear programming are described in Section 3 and 4, respectively. Finally, the concluding remarks appear in Section 5.

2 General scheme for prune-and-search

In the standard scheme for prune-and-search algorithm, the search-space is reduced by pruning. In each iteration of the while-loop, a constant fraction $\frac{1}{f}$ $(f > 1)$ 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. The stepwise description of the method is given in Algorithm 1.

Time complexity of this general prune-and-search type algorithm is $T(n) = T((1 - \frac{1}{f})n) + E(n) + O(n)$, 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.

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

2.1 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) INTERMEDIATE-COMPUTATION can be implemented in an in-place manner, and (ii) after INTERMEDIATE-COMPUTATION, both 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: $T(n) = T((1 - \frac{1}{f})n) + E_I(n) + O(n)$, where $E_I(n)$ is the time needed for in-place version of the 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 in-place INTERMEDIATE-COMPUTATION apart from the array containing the input. If $ES_I(n) = O(1)$, then total space complexity is also $O(1)$. Thus, we have the following theorem:

Theorem 1 *Any prune-and-search algorithm having the standard scheme given in Algorithm 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) 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 INTERMEDIATE-COMPUTATION*, each paired objects $(s_i, s_j) \in PAIR$ can be identified correctly.*

2.2 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 implement it using $O(\log n)$ extra-space (i.e., $O(\log^2 n)$ bits).

Total number of iterations of the while-loop in Algorithm 1 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 $O(1)$ extra-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 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 at the end of the t-th iteration, then (p, q) will again form a valid pair at $(t+1)$-th iteration, i.e., $(p, q) \in PAIR_{t+1}$.

(iii) If (p, s) are paired at $(t+1)$-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 t-th iteration (i.e., $(r, s) \in PAIR_t$), and q & 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 total number of variables and 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 $(\mathbf{p}, \mathbf{q})$ 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: $\mathbf{STATUS_{t-1}}$ contains -1:** Here, the index of the active object q in the input array R is stored in $STATUS_{t-1}$.
  - **Case 1.2: $\mathbf{STATUS_{t-1}}$ 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 $\mathbf{p}, \mathbf{q}$ is pruned:** Here both the objects p, q become members of S_t and the pair is considered 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 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.*

Lemma 1 leads to the following theorem:

Theorem 2 *Any prune-and-search algorithm having the standard scheme given in Algorithm 1 can be implemented in a read-only environment using $O(s)$ extra-space, provided* Intermediate-Computation *is implementable in the* pairing scheme *using $O(s)$ extra-space, where $s \geq \log n$.*

Remark 1 *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 $STATUS_1, STATUS_2, \cdots, STATUS_{t-1}$.*

2.3 Selection in the pairing scheme

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

- **ALGO-MR:** Munro and Raman's [22] selection algorithm which takes $O(n^{1+\epsilon})$ time and $O(\frac{1}{\epsilon})$ extra-space, where ϵ is a small fixed positive constant ($\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1$).
- **ALGO-RR:** Raman and Ramnath's [24] selection algorithm, which is slightly faster than ALGO-MR. Using $O(\log n)$ extra-space, it runs in $O(n \log^2 n)$ time.

For the self-completeness, we first 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 1 using the pairing scheme stated in Section 2.2.

2.3.1 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 $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$, where procedure $\mathcal{A}_i$ finds the median by evoking the procedure $\mathcal{A}_{i-1}$ for $i \in \{1, 2, \ldots, k\}$. The procedures $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ are stated below.

Procedure $\mathcal{A}_0$: 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 $\mathcal{A}_1$: It divides the array P into blocks of size $\sqrt{n}$ and in each block it finds the median using Procedure $\mathcal{A}_0$. 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 μ. Thus the time complexity of this procedure is $O(n\sqrt{n} \log n)$.

Procedure $\mathcal{A}_2$: It divides the whole array into $n^{1/3}$ blocks each of size $n^{2/3}$, and computes the block median using the procedure $\mathcal{A}_1$. 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 $\mathcal{A}_k$ 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 $\mathcal{A}_{k-1}$, $\mathcal{A}_{k-2}$, $\ldots$, $\mathcal{A}_0$, the space complexity of this algorithm is $O(k)$.

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

Result 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 $O(n^{1+\epsilon})$ time with $O(\frac{1}{\epsilon})$ extra-space, where $2\sqrt{\frac{\log \log n}{\log n}} \leq \epsilon < 1$.*

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 1 and Remark 1.). As procedure $\mathcal{A}_0$ needs to scan all the valid elements n times to find the median, it needs $O(n)$ number of enumerations. So, $\mathcal{A}_0$ will take $O(tn^2)$ time and

$O(t)$ extra-space (as space can be reused). Similarly, $\mathcal{A}_1$ takes $O(tn^{1+\frac{1}{2}} \log n)$ time and $O(t)$ extra-space. Finally, $\mathcal{A}_k$ 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 2 *In the t-th iteration of the while-loop, using the pairing scheme, the median can be found using ALGO-MR in $O(n^{1+\epsilon})$ time with $O(\frac{1}{\epsilon}t)$ extra-space, where ϵ is an user defined constant satisfying $2\sqrt{\frac{\log \log n}{\log n}} \leq \epsilon < 1$.*

2.3.2 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 for the recursion. Hence, we have the following result:

Result 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 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 2 and 3, we have the following:

Lemma 4 *In the t-th iteration of the while-loop, using the pairing scheme, the median can be found in*
(i) $O(tn^{1+\frac{1}{k+1}} \log^k n)$ time with $O(tk)$ extra-space, where k is a fixed natural number, or (ii) $O(tn \log^2 n)$ time with $O(t \log n)$ extra-space.

3 Minimum Enclosing Circle

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 pairwise distance of each pair of points in P is distinct. However, the assumptions can be relaxed by incorporating a few checks in the algorithm. First, we explain Megiddo's linear time algorithm [17] 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 execution all the points are available in the array P, possibly in a different permutation. Finally, we present an algorithm where the array P containing the input points is read-only.

3.1 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 [17] for the MEC problem for the points in P in Algorithm 2. In each iteration of the while-loop, it invokes INTERMEDIATE-COMPUTATION-for-MEC which is described in detail in Algorithm 3. This in turn calls the procedure CONSTRAINED-MEC, given in Algorithm 4.

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

The correctness of the algorithm is given in [17]. 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.

3.2 In-place implementation of MEC

In this section, we will show that Megiddo's algorithm (stated in Section 3.1) can be made in-place with the same time complexity. It is to be noted that this algorithm follows the standard

Algorithm 2: 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 $P$.
while $|P| \geq 16$ do
    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;
    INTERMEDIATE-COMPUTATION-for-MEC; (* It returns a quadrant $Quad$ defined by two perpendicular lines $\mathcal{L}_H$ and
    $\mathcal{L}_V$ *)
    (* Pruning step *)
    forall the pair of points $(P[2i], P[2i+1]) \in PAIR$ do
        if The bisector line $L_i$ defined by the pair $(P[2i], P[2i+1])$ does not intersect the quadrant Quad
        then
            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 $L_i$;
(* Finally, when $|P| < 16$ *) compute the minimum enclosing circle in brute force manner.
```

Algorithm 3: INTERMEDIATE-COMPUTATION-for-MEC(P)

```
Input: An array $P[1,\ldots,n]$ of points in $\mathbb{R}^2$.
Output: The triplet $(\mathcal{L}_H, \mathcal{L}_V, Quad)$, where $\mathcal{L}_H$ and $\mathcal{L}_V$ are a pair of mutually perpendicular lines and $Quad$ is one
        among the four quadrants defined by $\mathcal{L}_H$ and $\mathcal{L}_V$
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 $\{\alpha(L_i), i = 1, 2, \ldots, \lfloor \frac{|P|}{2} \rfloor\}$;
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 $\lfloor \frac{|P|}{4} \rfloor$ pairs of lines;
We split $M$ into two subsets $M_P$ and $M_I$, where
$M_P = \{(L_i, L_j) | \alpha(L_i) = \alpha(L_j) = \mu\}$ (* parallel line-pairs *) and
$M_I = \{(L_i, L_j) | \alpha(L_i) \neq \alpha(L_j)\}$ (* intersecting line-pairs *);
for each pair $(L_i, L_j) \in M_P$ do
    compute $y_{ij} = \frac{d_i + d_j}{2}$, where $d_i$ = distance of $L_i$ from the line $y = \mu x$
for each pair $(L_i, L_j) \in M_I$ do
    Let $a_{ij}$ = point of intersection of $L_i$ & $L_j$, and $b_{ij}$ = projection of $a_{ij}$ on $y = \mu x$. Compute
    $y_{ij}$ = signed distance of the pair of points $(a_{ij}, b_{ij})$, and
    $x_{ij}$ = signed distance of $b_{ij}$ from the origin;
Next, compute the median $y_m$ of the $y_{ij}$ values corresponding to all the pairs in $M$;
Step 3: Consider the line $\mathcal{L}_H : y = \mu x + y_m \sqrt{\mu^2 + 1}$, which is parallel to $y = \mu x$ and at a distance $y_m$ from $y = \mu x$;
Execute the Algorithm CONSTRAINED-MEC($P, \mathcal{L}_H$) to decide on which side of the line $\mathcal{L}_H$ the center $\pi^*$ of the
minimum enclosing circle lies;
Step 4: Let $M_I' = \{(L_i, L_j) \in M_I$ such that $a_{ij}$ and $\pi^*$ lie in the different sides of the line $\mathcal{L}_H$ $\}$ ;
Compute the median $x_m$ of $x_{ij}$-values for the line-pairs in $M_I'$. Define a line $\mathcal{L}_V$ perpendicular to $y = \mu x$ and passing
through a point on $y = \mu x$ at a distance $x_m$ from the origin;
Step 5: Execute Algorithm CONSTRAINED-MEC($P, \mathcal{L}_V$) and decide in which side of $\mathcal{L}_V$ the point $\pi^*$ lies ;
Step 6: Report $(\mathcal{L}_H, \mathcal{L}_V, Quad)$; Note that $Quad$ can be encoded by 2 bits.
```

scheme for the Prune-and-Search mentioned in Algorithm 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 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 $(p, q) \in PAIR$ can be identified correctly.

The in-place INTERMEDIATE-COMPUTATION-for-MEC maintains the following invariant which satisfies the condition (ii).

Algorithm 4: 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.
Step 1: $P' = P$;
while $|P'| \geq 3$ do
    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;
    Step 1.2: Let $\ell_i$ denote the perpendicular bisector of the pair of points $(P[2i-1], P[2i]) \in PAIR'$,
    $i = 1, 2, \ldots \lfloor\frac{|P'|}{2}\rfloor$.
    Let $\ell_i$ intersect $L$ at a point $q_i$, and $Q = \{q_i, i = 1, 2, \ldots, \lfloor\frac{|P'|}{2}\rfloor\}$;
    Step 1.3: Compute the median $m$ of the $y$-coordinate of the members of $Q$;
    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: *)
    Identify the point(s) $F \subset P'$ that is/are farthest from $m$;
    if the projection of all the members in $F$ on $L$ are in different sides of $m$ then
        $m^* = m$ (* center of the constrained minimum enclosing circle on the line $L$ *);
        Break;
        else
            (* i.e., the projection of all the members in $F$ on $L$ are in the same side (above or below) of $m$ *) $m^*$ lies
            in that side of $m$ on the line $L$
    Step 1.5:(* Pruning Step *) Without loss of generality, assume, that $m^* > m$. Then for each bisector 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 $\ell_{p,q}$;
Step 2: (* the case when $|P'| = 2$, *)
if the perpendicular bisector of the two members of $P'$ intersects $L$ then
    set $m^*$ as the point of intersection ;
else
    (* the perpendicular bisector of the members of $P'$ is parallel with $L$ *)
    $m^*$ = projection of the farthest point of $P'$ on $L$.
Step 3: (* Decide in which side of $L$ the center $\pi^*$ of the unconstrained MEC lies *);
Let $F$ be the set of points in $P'$ that are farthest from $m^*$;
if $|F| = 1$ then $\pi^*$ and the only point $p_i \in F$ lie in the same side of $L$;
if $|F| \geq 2$ then
    if all the members of $F$ lie in the same side of $L$, then $\pi^*$ will also lie in that side of $L$;
    otherwise (* we need to check whether the convex polygon formed by the points in $F$ contain $m^*$ or not as
    follows *)
    Let $F_1$ and $F_2$ be two subsets of $F$ that lie in two different sides of $L$ respectively; $F_1 \cup F_2 = F$.
    Let $m'$ be any point on $L$ that is below $m^*$.
    Find two points $p_i, p_j \in F_1$ such that $\angle m'm^*p_i = \max\{\angle m'm^*p | p \in F_1\}$ and $\angle m'm^*p_j = \min\{\angle m'm^*p | p \in F_1\}$.
    Now consider each point $q \in F_2$ and test whether $\pi \in \Delta p_i q p_j$.
    Similarly, find $p_k, p_\ell \in F_2$ such that $\angle m'm^*p_k = \max\{\angle m'm^*p | p \in F_1\}$ and $\angle m'm^*p_\ell = \min\{\angle m'm^*p | p \in F_1\}$.
    Consider each point $q' \in F_1$ and test whether $m^* \in \Delta p_k q' p_\ell$;
    If any one of these triangles contain $m^*$, then the convex polygon defined by the points in $F$ contains $m^*$. Here,
    the algorithm stops reporting $\pi^* = m^*$.
    Otherwise, either $(p_i, p_k)$ or $(p_j, p_\ell)$ define the diagonal (farthest pair of points) in $F$. Let $q$ be the mid-point of
    the diagonal. Here, $\pi^*$ and $q$ will lie in the same side of $L$;
```

Invariant 2 *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 $(p, q) \in PAIR$ defining L_i (their perpendicular bisector), will remain in consecutive locations of the input array P, for each $i = 1, 2, \ldots \lfloor\frac{|P|}{2}\rfloor$, 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 [8].

In Step 1 of the procedure INTERMEDIATE-COMPUTATION-for-MEC, we can compute the median angle μ in an in-place manner. Note that we need not have to store $\{L_i, i = 1, 2, \ldots \lfloor \frac{|P|}{2} \rfloor\}$ 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) $(\alpha(L_i) \leq \mu$ 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 3 *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 ν 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 ν 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 2 and 3 (see Lemma 6 in the Subsection 3.2.1). Thus, we have the following result.

Theorem 3 *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.*

3.2.1 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 π^* of the *unconstrained* MEC lies. Without loss of generality, let us assume that L is a vertical line. A straight forward way to implement this procedure in an in-place manner without maintaining Invariants 2 and 3 is as follows.

Find the median point m on the line L among the points of intersection of the lines ℓ_i and L for $i = 1, 2, \ldots, \frac{n}{2}$ in an in-place manner using the algorithm given in [8], where the points of

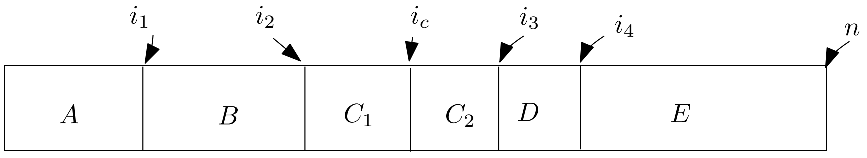

Figure 1: Block partition of the array P

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 ℓ_i. As we have $\frac{n}{4}$ intersection points below m, we can delete (i.e., move at the end of the array) $\frac{n}{4}$ 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 2, 3 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 1). Block A consists of those tuple (p, q, r, s) whose four 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 Constrained-MEC, we maintain the following invariant along with the Invariants 2 and 3.

Invariant 4 *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 $\{\ell_i, i = 1, 2, \dots \lfloor \frac{n}{2} \rfloor\}$, 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 [8]. If we form these bisector lines with two consecutive valid points in the array P, then the Invariants 2, 3 may not be maintained since (i) during the median finding ℓ_i's need to be swapped, and (ii) the points in a tuple may contribute to different ℓ_i's.

Here, three important things need to be mentioned:

Formation of ℓ_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 ℓ_i in Step 1 of the algorithm CONSTRAINED-MEC. Thus, we will have $\lfloor \frac{1}{2}(\frac{i_2-i_1}{4}) \rfloor$ bisectors. Let's denote these set of bisectors by $\mathcal{L}_1$.

Similarly, C_1 and C_2 will produce $\frac{i_c-i_2}{4}$ and $\frac{i_3-i_c}{4}$ bisector lines respectively, and these are denoted as $\mathcal{L}_2$ and $\mathcal{L}_3$ respectively.

In block D, each tuple (p,q,r,s) contains three *valid* points and the *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 $\mathcal{L}_4$, 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 *valid* point-pair (q,q'). Thus we get $\lfloor \frac{1}{2}(\frac{i_4-i_3}{4}) \rfloor$ such bisectors, and name this set $\mathcal{L}_5$.

From each tuple (p,q,r,s) in block E, we get two bisectors. Here, we form two sets of bisectors, namely $\mathcal{L}_6$ and $\mathcal{L}_7$. $\mathcal{L}_6$ is formed with (p,q) of each tuple in block E, and $\mathcal{L}_7$ is formed with (r,s) of each tuple in block E. Each of these sets contains $\lfloor \frac{n-i_4}{4} \rfloor$ bisectors.

Thus, we have seven sets of bisectors, namely $\mathcal{L}_i, i=1,2,\ldots,7$.

Computing Median: We compute the median of the points of intersection of the lines in each set of bisector lines $\mathcal{L}_i, i=1,2,\ldots,7$ 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 [8], if a pair of lines $\ell_i, \ell_j \in \mathcal{L}_k$ are swapped then the corresponding entire tuple(s) are swapped. Thus, the tuples are not broken for computing the median and all the Invariants 2, 3 and 4 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 $\cup_{i=1}^{7}\mathcal{L}_i$ with L. If a bisector line $\ell = (p,q) \in \cup_{i=1}^{7}\mathcal{L}_i$ intersects L in the interval $[m', m'']$, then none of p, q becomes *invalid*; otherwise, one of the points p or q becomes *invalid* as mentioned in Step 4 of the Procedure 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 swap 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 *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 *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 *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 *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 4.

Correctness and complexity results

Lemma 5 *The above Pruning Step ensures that Invariants 2, 3 and 4 are maintained, and at least $\frac{n'}{4}$ 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 $\mathcal{L}_i$ 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 $\mathcal{L}_i$ 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 4. It also helps in forming the bisector lines, and pruning of points maintaining Invariants 2 and 3. The second part of Lemma 5 justifies the following result.

Lemma 6 *The* Constrained-*MEC can be computed in an in-place manner in $O(n)$ time with constant amount of extra-space maintaining Invariants 2 and 3.*

3.3 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 3.1 which has the standard prune-and-search scheme as stated in Algorithm 1. According to Theorem 2, this algorithm is implementable in read-only memory, provided the procedure Intermediate-Computation-for-MEC is implementable in the *pairing scheme*.

Median finding (or Selection) and Constrained-MEC are the main steps in Intermediate-Computation-for-MEC. We already described how much time and space are required to find the median using the pairing scheme in Section 2.3. First, we show how one can compute Constrained-MEC using few extra variables when the input array is read-only. Next, we will show how to compute the MEC using this.

3.3.1 Constrained-MEC in read-only setup

Note that Constrained-MEC (given in Algorithm 4) also has the standard prune-and-search scheme stated in Algorithm 1.

At each iteration of the procedure Constrained-MEC, at least $\frac{1}{4}|P|$ points in P are pruned (marked *invalid*). Thus, the total number of iterations of the while-loop in the procedure 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 $t \in \{1, 2, \dots, K\}$ (see Section 2.2).

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 *invalid* elements up to the $(t-1)$-th iteration can be ignored correctly. We use one more array $IndexP$ 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 $IndexP$ works as the array $STATUS$ (see Section 2.2).

Note that we have no space to store the mark bit for the *invalid* points in the array P. We use the *compute in lieu of store* paradigm, or in other words, we check whether a point is *valid* at the t-th iteration, by testing its validity in all the $i = 1, 2, \dots, t-1$ *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 $i = 1, 2, \dots, t-1$. As in the in-place version of the Constrained-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 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 *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 $(P[2\alpha-1], P[2\alpha])$, $\alpha = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$ in order. For each pair, we compute their bisector ℓ_α, 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 ℓ_α 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 ℓ_α 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 2α 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 7 *Invariant 5 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 $D[t-1]$. □

Observation 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 β 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 β may be a valid index or -1), then at the end of all subsequent iterations j ($> t$) it will be observed that $IndexP[t-1] = \beta$.

[1]Similar action is taken if $P[2\alpha-1]$ remains valid and $P[2\alpha]$ becomes invalid.

Proof: Part (i) follows from the pairing scheme. Parts (ii) & (iii) follow from Lemma 7. □

Lemma 8 *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 $t-2$. □

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 Section 2.3, we have shown how to compute the median in this pairing scheme. So, combining the Lemma 4, we have the following result:

Lemma 9 *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 3.2.1. Thus, we have the following result:

Lemma 10 *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 $O(\log n)$. □

3.3.2 (Unconstrained) MEC in the read-only setup

We use the read-only variation of the Constrained-MEC algorithm (described in Subsection 3.3.1) for solving the minimum enclosing circle problem.

Here, we need to maintain three more arrays $\mathcal{M}$, $\mathcal{D}$ and $\mathcal{I}$, each of size $O(\log n)$. $\mathcal{M}[t]$ contains two mutually perpendicular lines and $\mathcal{D}[t]$ (a two bit space) indicates the quadrant in which the center π^* of the unconstrained MEC lies, in the t-th iteration of the while-loop of the algorithm MEC. So, the arrays $\mathcal{M}$, $\mathcal{D}$ acts as the SPC and the array $\mathcal{I}$ plays the role of $STATUS$ as described in Section 2.2.

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.

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, total number of iterations of this *inner-loop* for MEC is $O(\log^2 n)$. From Lemma 9, one can see the following:

Lemma 11 *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 $\mathcal{M}$, $\mathcal{D}$ and $\mathcal{I}$ 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 9, the result follows. $\square$

Hence, we have the following result:

Theorem 4 *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) $O(n^{1+\frac{1}{k+1}} \log^{k+3} n)$ 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.*

4 Low dimensional linear programming

Linear programming is one of the most important tool in solving a large number of optimization problems. In this section, 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 [17]. 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.

4.1 Linear programming in $\mathbb{R}^2$

Here the optimization problem is as follows:

$$\min_{x_1,x_2} c_1x_1 + c_2x_2$$

subject to: $a'_ix_1 + b'_ix_2 \geq \beta_i,\ i \in I = \{1, 2, \ldots n\}.$

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

$$\begin{array}{ll} & \min_{x,y} y \\ \text{subject to:} & y \geq a_i x + b_i,\ i \in I_1, \\ & y \leq a_i x + b_i,\ i \in I_2, \\ & |I_1| + |I_2| \leq n. \end{array}$$

Algorithm 5: 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:
$\min_{x,y} y$, subject to
(i) $y \geq a_i x + b_i$, $i \in I_1$, (ii) $y \leq a_i x + b_i$, $i \in I_2$, where $|I_1| + |I_2| \leq n$.
STEP 2: Set $a = -\infty$ and $b = \infty$;
while $|I_1 \cup I_2| > 4$ do
    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 $x_{ij} = \frac{b_i - b_j}{a_j - a_i}$;
    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
        $s_g = \min a_i | i \in I_1, a_i x_m + b_i = g$; $S_g = \max a_i | i \in I_1, a_i x_m + b_i = g$;
        $s_h = \min a_i | i \in I_2, a_i x_m + b_i = h$; $S_h = \max a_i | i \in I_2, a_i x_m + b_i = h$;
    STEP 2.3.2: (* $g \leq h \Rightarrow x_m$ 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 $x_1 = x_m$ & $x_2 = \frac{g - c_1 x_1}{c_2}$; 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.
```

The detailed steps of Megiddo's-2D-LP algorithm is given as MEGIDDO'S-2D-LP(I, c_1, c_2) in Algorithm 5. The justification of all the steps are available in [17].

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 $K = O(\log n)$.

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

We will give step by step description of implementing 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 $y = c_1x_1 + c_2x_2$, 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.

Observe that rest of this algorithm follows standard prune-and-search scheme stated in Algorithm 1.

So, according to Lemma 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 2.3 can be implementable easily once we can recognize the valid pairs/constraints, and STEP 2.2 is the median computation step. So, combining Lemma 1 and 4, we have the following:

Theorem 5 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.*

4.2 Linear programming in $\mathbb{R}^3$

In the same paper [17], Megiddo proposed a linear time algorithm for linear programming problem with three variables. The problem is stated as follows:

$$\begin{array}{ll} & \min_{x_1,x_2,x_3} d_1x_1 + d_2x_2 + d_3x_3 \\ \text{subject to:} & a'_ix_1 + b'_ix_2 + c'_ix_3 \geq \beta_i,\ i \in I = \{1, 2, \ldots n\}. \end{array}$$

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

$$\begin{array}{ll} & \min_{x,y,z} z \\ \text{subject to:} & z \geq a_ix + b_iy + c_i,\ i \in I_1, \\ & z \leq a_ix + b_iy + c_i,\ i \in I_2, \\ & 0 \geq a_ix + b_iy + c_i,\ i \in I_3, \\ & |I_1| + |I_2| + |I_3| \leq n. \end{array}$$

The Algorithm MEGIDDO'S-3D-LP pairs-up constraints (C_k^i, C_k^j), where C_k^i, C_k^j are from same set $I_k, k \in \{1, 2, 3\}$. So, there are at most $\frac{n}{2}$ pairs. Let C_k^i (resp. C_k^j) corresponds to $a_ix+b_iy+c_i$ (resp. $a_jx+b_jy+c_j$). 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}: $a_ix+b_iy+c_i = a_jx+b_jy+c_j$ which divides the plane into two halves. Let $\mathcal{L}$ be the set of lines obtained in this way. We compute the median μ of the

Algorithm 6: MEGIDDO'S-3D-LP(I, c_1, c_2)

```
Input: A set of $n$ constraints $a'_i x_1 + b'_i x_2 + c'_i x_3 \geq \beta_i$, for $i \in I = \{1, 2, \ldots n\}$,
Output: The value of $x_1, x_2, x_3$ which minimizes $c_1 x_1 + c_2 x_2 + c_3 x_3$
(* Uses standard prune-and-search scheme *)
STEP 1: Convert the given form into following:
$\min_{x,y,z} z$, subject to (i) $y \geq a_i x + b_i y + c_i$, $i \in I_1$, (ii) $y \leq a_i x + b_i y + c_i$, $i \in I_2$, (iii) $0 \geq a_i x + b_i y + c_i$, $i \in I_3$, where
$|I_1| + |I_2| + |I_3| \leq n$.
STEP 2:
while $|I_1 \cup I_2 \cup I_3| \geq 16$ do
    STEP 2.1: Arbitrarily pair-up the constraints $a_i x + b_i y + c_i$, $a_j x + b_j y + c_j$ where $i, j$ are from same set
    $I_k, k \in \{1, 2, 3\}$. Let $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ be the set of aforesaid pairs and $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3$.
    STEP 2.2: Let $\mathcal{L}_C = \{(i,j) \in \mathcal{L} | (a_i, b_i) \neq (a_j, b_j)\}$ and $\mathcal{L}_P = \{(i,j) \in \mathcal{L} | (a_i, b_i) = (a_j, b_j)\}$;
    Compute the median $\mu$ of the slopes $\alpha(L_{ij})$ of all the straight lines $L_{ij}$: $a_i x + b_i y + c_i = a_j x + b_j y + c_j$, $(i,j) \in \mathcal{L}_C$;
    STEP 2.3:
    Arbitrarily pair up $(L_{ij}, L_{i'j'})$ where $\alpha(L_{ij}) \leq \mu \leq \alpha(L_{i'j'})$ and $(i,j), (i',j') \in \mathcal{L}_C$. Let $M$ be the set of these $\lfloor \frac{n}{4} \rfloor$
    pairs of lines;
    Let $M_P = \{(L_i, L_j) \in M | \alpha(L_i) = \alpha(L_j) = \mu\}$ (* parallel line-pairs *) and $M_I = \{(L_i, L_j) \in M | \alpha(L_i) \neq \alpha(L_j)\}$ (*
    intersecting line-pairs *);
    for each pair $(L_i, L_j) \in M_P$ do
        compute $y_{ij} = \frac{d_i + d_j}{2}$, where $d_i$ = distance of $L_i$ from the line $y = \mu x$
    for each pair $(L_i, L_j) \in M_I$ do
        Let $a_{ij}$ = point of intersection of $L_i$ & $L_j$, and $b_{ij}$ = projection of $a_{ij}$ on $y = \mu x$. Compute $y_{ij}$ = signed
        distance of the pair of points $(a_{ij}, b_{ij})$, and $x_{ij}$ = signed distance of $b_{ij}$ from the origin;
    Next, compute the median $y_m$ of the $y_{ij}$ values corresponding to all the pairs in $M$;
    Step 2.4: Consider the line $L_H : y = \mu x + y_m \sqrt{\mu^2 + 1}$, which is parallel to $y = \mu x$ and at a distance $y_m$ from
    $y = \mu x$;
    Test on which half-plane defined by the line $L_H$ contains the optimum by evoking TESTING-LINE$(L_H)$
    Step 2.5: Let $M'_I = \{(L_i, L_j) \in M_I | \ a_{ij}$ & $\pi^*$ lie in different sides of $L_H\}$;
    Compute the median $x_m$ of $x_{ij}$-values for the line-pairs in $M'_I$. Define a line $L_V$ perpendicular to $y = \mu x$ and
    passing through a point on $y = \mu x$ at a distance $x_m$ from the origin;
    Execute the procedure TESTING-LINE$(L_V)$ and decide in which side of $L_V$ the optimum lies;
    We consider $L_H$ and $L_V$ as horizontal and vertical lines respectively; let $Quad$ is the quadrant defined by $L_H$ and
    $L_V$ in which the optimum lies;
    Step 2.6: (* Pruning step *)
    forall the $i, j \in \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3$ do
        if $(a_i, b_i) = (a_j, b_j)$ then
            Ignore one of the corresponding constraints depending on their $c$ value;
        else if the corresponding line $L_{ij}$ does not intersect the quadrant $Quad$ then
            Discard one of the constraints depending on their position with respect to $Quad$;
STEP 3: *(The case when $|I_1 \cup I_2 \cup I_3| \leq 16$)*
The problem can be solved directly by brute-force manner.
```

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 μ and the other one have slope greater than μ. Let Π be the set of these paired lines. Each of these pairs will intersect. We compute the intersection point a having mch4:PSedian y_m among the y-coordinates of these intersection points. Next, we execute the procedure TESTING-LINE (described in Subsection 4.2.1) with respect to the line $L_H : y = \mu x + y_m \sqrt{\mu^2 + 1}$ (having slope μ and passing through 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 Π which intersect on the other side of the optimum solution. Among these pairs, we compute the intersection point b 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 Π that intersect in the quadrant Q', diagonally opposite to Q. Let $(L_{ij}, L_{k\ell})$ be a paired line of Π 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 [17]). 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 detail description of the algorithm MEGIDDO'S-3D-LP is given in Algorithm 6.

4.2.1 TESTING-LINE

The procedure TESTING-LINE, described in [17], works as follows. It takes a straight line L in the x-y plane and decide 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 $\mathbb{R}^3$:

$$\begin{array}{ll} & \min_{x,y,z} z \\ \text{subject to:} & z \geq a_ix + b_iy + c_i,\ i \in I_1, \\ & z \leq a_ix + b_iy + c_i,\ i \in I_2, \\ & 0 \geq a_ix + b_iy + c_i,\ i \in I_3, \\ & |I_1| + |I_2| + |I_3| \leq n. \end{array}$$

Here, the constraints are only valid constraints when TESTING-LINE is evoked.

Define $f(x, y) = max\{\max\{a_ix+b_iy+c_i, i \in I_1\} - \min\{a_ix+b_iy+c_i, i \in I_2\}, \max\{a_ix+b_iy+c_i, i \in I_3\}\}$.

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:

$$\begin{array}{ll} & \min_{x,z} z \\ \text{subject to:} & z \geq a_ix + c_i,\ i \in I_1, \\ & z \leq a_ix + c_i,\ i \in I_2, \\ & 0 \geq a_ix + c_i,\ i \in I_3, \\ & |I_1| + |I_2| + |I_3| \leq n. \end{array}$$

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:

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

If $\eta < 0$ is reported, then Testing-Line decides $y > 0$. Otherwise, it performs the following linear programming:

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

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 [17].

4.2.2 Read-only linear programming in $\mathbb{R}^3$

Observe that Megiddo's-3D-LP, given in Algorithm 6, follows the standard prune-and-search scheme stated in Algorithm 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 2.2. 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 Section 2.3, we have already shown that median can be computed using the pairing scheme in the read-only environment (see Lemma 4). In Section 4.1 (see Theorem 5), 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 6 *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.*

5 Conclusion

We provide a standard prune-and-search scheme and show how to implement any algorithm which follows this standard scheme in space-efficient manner. We consider two fundamental problems: (i) minimum enclosing circle of a point set in $\mathbb{R}^2$, and (ii) linear programming problems with two and three variables. We believe that our technique can be widely applied to many other problems as well.

Acknowledgement

The authors acknowledge V. Raman for referring the better selection algorithm [24].

References

[1] T. Asano and B. Doerr. Memory-constrained algorithms for shortest path problem. In *CCCG*, 2011.

[2] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *JoCG*, 2(1):46–68, 2011.

[3] L. Barba, M. Korman, S. Langerman, R. I. Silveira, and K. Sadakane. Space-time trade-offs for stack-based algorithms. In *STACS*, pages 281–292, 2013.

[4] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. *Algorithmica*, 57(1):1–21, 2010.

[5] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom.*, 37(3):209–227, 2007.

[6] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Symp. on Comput. Geom.*, pages 239–246, 2004.

[7] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.*, 321(1):25–40, 2004.

[8] S. Carlsson and M. Sundström. Linear-time in-place selection in less than 3n comparisons. In *ISAAC*, pages 244–253, 1995.

[9] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007.

[10] M. De, S. C. Nandy, and S. Roy. Convex hull and linear programming in read-only setup with limited work-space. *CoRR*, abs/1212.5353, 2012.

[11] D. J. Elzinga and D. W. Hearn. Geometrical solutions for some minimax location problems. *Transportation Science*, 6(4):379–394, 1972.

[12] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *STOC*, pages 302–311, 1984.

[13] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, 20:191–194, 1979.

[14] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.

[15] M. Korman. Private communication.

[16] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[17] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM J. Comput.*, 12(4):759–776, 1983.

[18] N. Megiddo. The weighted euclidean 1-center problem. *Mathematics of Operations Research*, 8(4):498–504, 1983.

[19] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31(1):114–127, 1984.

[20] N. Megiddo. On the ball spanned by balls. *Discrete Computational Geometry*, 4:605–610, 1989.

[21] N. Megiddo and E. Zemel. An o(n log n) randomizing algorithm for the weighted euclidean 1-center problem. *J. Algorithms*, 7(3):358–368, 1986.

[22] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996.

[23] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* 1990.

[24] V. Raman and S. Ramnath. Improved upper bounds for time-space trade-offs for selection. *Nord. J. Comput.*, 6(2):162–180, 1999.

[25] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[26] J. J. Sylvester. A question in the geometry of situation. *Quarterly Journal of Mathemaitcs*, 1:79, 1857.

[27] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *Results and New Trends in Computer Science*, pages 359–370. Springer-Verlag, 1991.