Minimum Enclosing Circle with Few Extra Variables

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

1 **Indian Statistical Institute, Kolkata - 700108, India**
{minati_r,nandysc}@isical.ac.in
2 **Chennai Mathematical Institute, Chennai - 603103, India**
sasanka@cmi.ac.in

Abstract

Asano et al. [JoCG 2011] 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. We show that Megiddo's prune and search algorithm for computing the minimum radius circle enclosing the given points can be tailored to work in a read-only environment in $O(n^{1+\epsilon})$ time using $O(\log n)$ extra space, where ϵ is a positive constant less than 1. As a warm-up, we first solve the same problem in an *in-place* setup in linear time with $O(1)$ extra space.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Minimum enclosing circle, space-efficient algorithm, prune-and-search

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2012.510

1 Introduction

The minimum enclosing circle (MEC) for a set of points P 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 devices 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 is first proposed by Sylvester in the year 1857 [15], and it asks for the location of a single facility that minimizes the distance (in some chosen metric) of the farthest demand point from the facility. Thus the problem we are considering is the Euclidean version of the 1-center problem. Elzinga et al. with their work [8] 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 [14], we have an $O(n \log n)$ time algorithm for computing the MEC of the point set P. The best known result for computing the MEC is an $O(n)$ time algorithm proposed by Megiddo [10]. Later Welzl [16] 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 [11] that runs in $O(n(\log n)^3(\log \log n)^2)$ time using the parametric search [9]. Later,

32nd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012).
Editors: D. D'Souza, J. Radhakrishnan, and K. Telikepalli; pp. 510–521

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Megiddo and Zemel [12] proposed an $O(n \log n)$ time randomized algorithm for this problem that does not use parametric search. All of these algorithms use $O(n)$ extra work-space.

Recently, Asano et al. [2] proposed an $O(n^2)$ time and $O(1)$ extra-space algorithm for computing the vertices of the farthest point Voronoi diagram of 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. In the same paper they mentioned the possibility of finding the minimum enclosing circle in sub-quadratic time in read-only setup with sub-linear work-space as an open problem. We answer this question affirmatively as stated below.

Our results: In this paper, we propose an algorithm for computing the minimum enclosing circle of a given set of n points in a read-only array. The time and extra space required for this algorithm are $O(n^{1+\epsilon})$ and $O(\log n)$ respectively, where $\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1$. As a warm-up, we first propose an algorithm for the same problem in an in-place model where swapping elements in the array is permissible. This needs $O(n)$ time and $O(1)$ extra-space. This algorithm is invoked in our proposed algorithm in the read-only setup.

Related works: 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 [6]. If the array is read-only (i.e., swapping two values in the input array is prohibited) then the problem can be solved in $O(n^{1+\epsilon})$ time with $O(\frac{1}{\epsilon})$ space, where ϵ is a small (< 1) positive number to be fixed prior to the execution [13]. Chan [7] has shown that if S extra-bits are given in addition to the input array, then a lower bound on the expected time complexity of finding the median is $\Omega(n \log \log_S n)$. A lower bound on the deterministic time complexity for the same problem is $\Omega(n \log^* \frac{n}{S} + n \log_S n)$, where S extra-words are given as work space [7]. An important work in a different direction is the in-place algorithm for the linear programming problem with two variables, which can be solved in $O(n)$ time with $O(1)$ extra space, where n is the number of constraints [5]. This can be used to design a prune-and-search algorithm for finding the center of the minimum enclosing circle of the point set P where the center is constrained to lie on a given straight line [10].

Low-memory algorithms have many advantages compared to traditional algorithms [5, 6]. 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 faster. Readers are referred to [3, 4] for the in-place algorithms of several other geometric optimization problems. For the geometric algorithms in the read-only setup, see [1, 2].

2 Overview of Megiddo's algorithm

Let $P[0, \ldots, n-1]$ be an array containing n points. We now describe Megiddo's linear time algorithm for the MEC problem for the points in P. 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 deleted.

Algorithm 1: MEC(P)

Input: An array $P[1, \ldots, n]$ of points in $\mathbb{R}^2$.
Output: The center m^* of the minimum enclosing circle of the points in P.
while $|P| \geq 16$ **do**
  $P = PRUNE(P)$
(* Finally, when $|P| < 16$ *) compute the minimum enclosing circle in brute force manner.

Algorithm 2: PRUNE(P)

Input: An array $P[1,\ldots,n]$ of points in $\mathbb{R}^2$.
Output: The set of points P after pruning.
Step 1: Arbitrarily pair up the points in P. Let $(P[2i], P[2i+1]), i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor)$ be the aforesaid pairs;
Step 2: Let L_i denote the bisector of the pair of points $(P[2i], P[2i+1])$, and $\alpha(L_i)$ denote the angle of L_i with the x-axis. Compute the median μ of $\{\alpha(L_i), i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$;
Step 3: 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 $\lfloor \frac{n}{4} \rfloor$ 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 4: 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 $y = \mu x$;
Compute the center π of the constrained minimum enclosing circle whose center lies on $\mathcal{L}_H$ using Algorithm *Constrained_MEC*$(P, \mathcal{L}_H)$;
Step 5: (* Decide in which side of $\mathcal{L}_H$ the center π^* of the unconstrained MEC lies *);
Let Q be the set of points in P that are farthest from π;
if $|Q| = 1$ **then** π^* and the only point $p_i \in Q$ lie in the same side of $\mathcal{L}_H$;
if $|Q| \geq 2$ **then**
  if all the members of Q lie in the same side of $\mathcal{L}_H$, **then** π^* will also lie in that side of $\mathcal{L}_H$;
  otherwise (* we need to check whether the convex polygon formed by the points in Q contain π or not as follows *)
  Let Q_1 and Q_2 be two subsets of Q that lie in two different sides of $\mathcal{L}_H$ respectively;
  $Q_1 \cup Q_2 = Q$. Find two points $p_i, p_j \in Q_1$ that make maximum and minimum angles with $\mathcal{L}_H$ with center at π in anticlockwise direction. Now consider each point $q \in Q_2$ and test whether $\pi \in \Delta p_i q p_j$.
  Similarly, find $p_k, p_\ell \in Q_2$ that make maximum and minimum angles with $\mathcal{L}_H$ with center at π in clockwise direction. Consider each point $q' \in Q_1$ and test whether $\pi \in \Delta p_k q' p_\ell$;
  If any one of these triangles contain π, then the convex polygon Q contains π. Here the algorithm stops reporting $\pi^* = \pi$.
  Otherwise, either (p_i, p_k) or (p_j, p_ℓ) define the diagonal (farthest pair of points) in Q. Let q be the mid-point of the diagonal. Here, π^* and q will lie in the same side of $\mathcal{L}_H$;
Step 6: Let $M'_I = \{(L_i, L_j) \in M_I |$ a_{ij} and π^* 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;
Execute Algorithm *Constrained_MEC*$(P, \mathcal{L}_V)$ and decide in which side of $\mathcal{L}_V$ the point π^* lies as in Step 5;
From now onwards, we will denote $\mathcal{L}_H$ and $\mathcal{L}_V$ as horizontal and vertical lines respectively;
Without loss of generality, assume that π^* lies in the top-left quadrant;
Step 7: (* Pruning step *)
for *all the members* $(L_i, L_j) \in M_I$ *whose points of intersection* (a_{ij}) *lie in the bottom-right quadrant* **do**
  Let $\alpha(L_i) \leq \mu$ and L_i be defined by the pair of points $(P[2i], P[2i+1])$;
  Discard one of $P[2i]$ and $P[2i+1]$ which is top-left to other one from P;
for *all the members* $(L_i, L_j) \in M_P$ *whose* $y_{ij} \leq y_m$ **do**
  Let L_i be below $\mathcal{L}_H$, and L_i be defined by a pair of points $[P[2i], P[2i+1])$;
  Discard either $P[2i]$ or $P[2i+1]$ depending on which one lies above L_i;
Step 9: return P (* Now P denotes the set of points after pruning *).

Algorithm 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.
Step 1:
while $|P| \geq 3$ **do**
  Step 1.1: Arbitrarily pair up the points in P. Let $(P[2i], P[2i+1]), i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor$ be the aforesaid pairs;
  Step 1.2: Let ℓ_i denote the perpendicular bisector of the pair of points $(P[2i], P[2i+1])$, $i = 0, 1, \ldots \lfloor \frac{n}{2} \rfloor$. Let ℓ_i intersect L at a point q_i. and $Q = \{q_i, i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$;
  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**
    return $m^* = m$ (* center of the constrained minimum enclosing circle on the line L *)
  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: Without loss of generality, assume, that $m^* > m$. Then for each bisector line $\ell_{p,q}$ (defined by the point-pair $p, q \in P$) 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 members of P intersects L* **then**
  return the point of intersection as m^*;
else
  (* the perpendicular bisector of the members of P is parallel with L *)
  return m^* = projection of the farthest point of P on L.

The correctness of the algorithm is given in [10]. An iteration of the procedure *PRUNE* 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 procedure *PRUNE* at most $O(\log n)$ times, and its total running time is $O(n)$ time using $O(n)$ extra space.

3 In-place implementation of MEC

In this section, we will show that Megiddo's algorithm (stated in the Section 2) can be made in-place with the same time complexity. It is to be noted that we may succeed in making all the steps 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 4 of the procedure *PRUNE*) in-place (as *2D linear programming* can be solved in an in-place manner in linear time [5]), but one will have to assure that after this, one will be able to figure out the chosen pair of bisectors satisfying the condition mentioned in Step 3 of the procedure *PRUNE*, as this will be required in the Step 6 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 [6].

In Step 2 of the procedure *PRUNE* we can compute the median angle μ in an in-place manner. Note that we do not have to store the $\{L_i, i = 0, 1, \ldots \lfloor \frac{n}{2} \rfloor\}$ as one can compute them on demand with the knowledge of $(P[2i], P[2i+1])$.

Step 3 of the procedure *PRUNE* can be made in-place in $O(n)$ time and $O(1)$ extra space as

follows: identify $\lfloor \frac{n}{4} \rfloor$ 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 $(P[2i], P[2i+1], P[2j], P[2j+1])$ 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 3 of Procedure *PRUNE*) on the fly. So, we maintain the following *invariant*

▶ Invariant 1. (i) During the execution of Steps 3-6 of the procedure *PRUNE*, the pair of points (p, q) of P defining L_i (their perpendicular bisector), for each $i = 0, 1, \ldots \lfloor \frac{n}{2} \rfloor$ will remain in consecutive locations of the input array P.

(ii) During the execution of Steps 4-6 of the procedure *PRUNE*, the tuple of points (p, q, r, s) of P, defining the y_{ij}-value for two bisectors (L_i, L_j) ((p, q) defining L_i and (r, s) defining L_j) that satisfy $\alpha(L_i) \leq \mu$ and $\alpha(L_j) \geq \mu$, 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. In each iteration of the Algorithm MEC, after the (pruning) Step 7 of the procedure *PRUNE*, the deleted points are moved at the end of the array, and ν is updated to the number of non-deleted points. We have already shown that Steps 1-3 can be made in-place. In the next subsection, we show that Steps 4-6 can also be made in-place satisfying *invariant* 1 (see Lemma 3). Thus, we have the following result.

▶ **Theorem 1.** *Minimum enclosing circle of 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 work-space.*

3.1 In-place implementation of *Constrained_MEC*

In a particular iteration 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 4 of the procedure *PRUNE*, 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. 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 *Invariant* 1 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 [6], 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 $\ell_i = (P[2i], P[2i+1])$ 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 Invariant 1 will not remain valid.

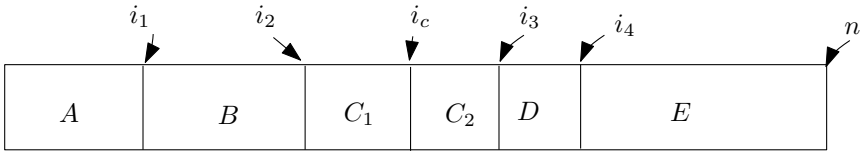

Figure 1 Block Partition of the Array P

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 block 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 *Invariant* 1.

▶ Invariant 2. The tuples with zero, one, two, three and four valid point(s) will be in the block A, B, C, D and E, respectively as mentioned above.

Now, we need (i) to form the bisector lines $\{\ell_i, i = 1, 2, \ldots \lfloor \frac{n}{2} \rfloor\}$, 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 [6]. If we form these bisector lines with two consecutive valid points in the array P, then the Invariant 1 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: 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 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 $\mathcal{L}_i$ 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 [6], 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 both the Invariants 1 and 2 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 tuple $(p, q, r, s), (p', q', r', s') \in D$, we test the bisector lines $\ell = (q, q') \in \mathcal{L}_4$ and $\ell' = (r, s) \in \mathcal{L}_4$ and $\ell'' = (r', s') \in \mathcal{L}_5$ with $[m', m'']$. This may cause deletion of one or two points from (p, q, r, s) (resp. (p', q', r', s')). If for the tuple (p, q, r, s),

- none of the points becomes *invalid*, it will remain in the set D;
- if only q becomes invalid, it 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*, it 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*, it is moved to B with necessary adjustment of i_2, i_3;
- if q and s both become *invalid*, then it 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 movement can be done with one or two tuple-swap operation. Surely at most two swap operation inside the tuple may be required to satisfy Invariant 2.

3.1.0.1 Correctness and complexity results

▶ **Lemma 2.** *The above pruning steps ensure Invariants 1 and 2 and at least* $\frac{n}{4}$ *points become* invalid *after each iteration, where* n *is the number of valid points in* P *at the beginning of the 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 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 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 *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 2. It also helps in forming the bisector lines, and pruning of points maintaining Invariant 1. The second part of Lemma 2 justifies the following result.

▶ **Lemma 3.** *The Constrained_MEC can be computed in an in-place manner in $O(n)$ time with constant amount of extra space.*

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 $\mathbb{R}^2$ with $O(\log n)$ 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 2. As we are not allowed to move the deleted elements to one end, the main challenge in read-only memory for implementing Megiddo's algorithm is in detecting the *valid* points after pruning.

Long ago Munro and Raman [13] gave a space-time trade-off for median finding algorithms in read-only memory. Though we can not use their algorithm directly for median finding in our setup, we will use a similar idea. For ease of understanding, we will briefly describe the median finding algorithm of [13]. Next we will describe our approach for computing the minimum enclosing circle for the points in the array P.

4.1 Munro and Raman's median finding algorithm

Given a set of n points in $\mathbb{R}$ in a read-only array P, the algorithm of [13] 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 marked as *invalid*. This does not need any mark bit; only one needs 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 marked *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 $n^{1+\frac{1}{3}} \log^2 n$.

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 $\mathcal{A}_{k-1}$, $\mathcal{A}_{k-2}$, ..., $\mathcal{A}_0$, 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 $n^\epsilon = \log^{\frac{1}{\epsilon}} n$, then ϵ will be $\sqrt{\frac{\log \log n}{\log n}}$, and this will give 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 memory, the median can be found in $O(n^{1+\epsilon})$ time with $O(\frac{1}{\epsilon})$ extra-space, where $2\sqrt{\frac{\log \log n}{\log n}} \leq \epsilon < 1$.

4.2 Algorithm MEC in read-only setup

Given a set of n points in $\mathbb{R}^2$ in a read-only array P of size n, our objective is to compute the minimum enclosing circle of the points in P using $O(\log n)$ extra space. We first show how one can compute *Constrained_MEC* when the input array is read-only using $O(\log n)$ extra variables. Next, we use this algorithm along with another $O(\log n)$ space to compute the center of the unconstrained minimum enclosing circle.

4.2.1 *Constrained_MEC* in read-only setup

We first note that at each iteration of the procedure *Constrained_MEC* at least $\frac{1}{4}|P|$ points in P are pruned (marked *invalid*). Thus, the number of iterations executed in the procedure *Constrained_MEC* is at most $O(\log |P|)$.

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 i-th iteration, we store m at $M[i]$; $D[i]$ will contain 0 or 1 depending on whether $m^* > m$ or $m^* < m$.

We now explain the i-th iteration assuming that $(i-1)$ iterations are over. Here we need to pair-up points in P in such a way that all the *invalid* elements up to the $(i-1)$-th iteration can be ignored correctly. We use one more array $IndexP$ of size $\log |P|$. At the beginning of this iteration all the elements in this array are initialized with -1.

Note that, we have no space to store the mark bit for the *invalid* points in the array P. Thus, we use the *compute in lieu of store* paradigm, or in other words, we check whether a point is *valid* at the i-th iteration, by testing its validity in all the $t = 1, 2, \ldots, i-1$ *levels* (previous iterations).

We start scanning the input array P from the left, and identify the points that are tested as *valid* in the t-th level for all $t = 1, 2, \ldots, i-1$. 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 $(i-1)$-th iteration (p, q) form a pair, and p is observed as *invalid*. While

executing the i-th iteration, we again need to check whether p was *valid* in the $i-1$-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 i-th iteration, (p, q) should be paired at the $(i-1)$-th level.

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

▶ Invariant 3. If (i) two points $p, q \in P$ form a point-pair at the t-th level in the j-th iteration, and (ii) both of them remain *valid* up to k-th iteration where $k > j$, then p, q will also form a point-pair at the t-th level of the the k-th iteration.

Pairing scheme: We consider the point-pairs $(P[2\alpha], P[2\alpha+1])$, $\alpha = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor$ 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 remains *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 $P[2\alpha]$ and $P[2\alpha+1]$ 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 i-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 ($< i-1$). In Case (i), the point pair $(P[2\alpha], P[2\alpha+1])$ participates in computing the median value m_i. In case (ii), suppose $P[2\alpha]$ remains *valid* and $P[2\alpha+1]$ becomes *invalid*. 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 or $2\alpha+1$ in $IndexP[j]$ depending on whether $P[2\alpha]$ or $P[2\alpha+1]$ remains *valid* at level 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 i-th level or one of them is marked *invalid* in some level between j and i. Both the situations are handled in a manner similar to Cases (i) and (ii) as stated above.

▶ **Lemma 4.** *Invariant 3 is maintained throughout the execution.*

Proof. Follows from the fact that the tests for the points in P at different levels $t = 1, 2, \ldots, i-2$ at both the $(i-1)$-th and i-th iterations are the same. At the $(i-1)$-th level of the $(i-1)$-th iteration, we compute m_{i-1} and D_{i-1} with the *valid* points. At the $(i-1)$-th level of the i-th iteration, we prune points that were tested *valid* at the $(i-1)$-th iteration using M_{i-1} and D_{i-1}. ◀

▶ Observation 1. At the end of the i-th iteration,

(i) Some cells of the $IndexP$ array may contain valid indices ($\neq -1$).
(ii) In particular, $IndexP[i-1]$ will either contain -1 or it will contain the index of some point β in P that has participated in computing m_{i-1} (i.e., remained *valid* up to level $i-1$).
(iii) If in this iteration $IndexP[i-1] = \beta$ (where β may be a valid index or -1), then at the end of all subsequent iterations j ($> i$) it will be observed that $IndexP[i-1] = \beta$.

Proof. Part (i) follows from the pairing scheme. Parts (ii) & (iii) follow from Lemma 4. ◀

▶ **Lemma 5.** *In the i-th iteration, the amortized time complexity for finding all valid pairs is $O(ni)$.*

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

The main task in the i-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. We essentially use the median finding algorithm in [13] for this purpose. Notice that, in order to get each intersection point, we need to get a *valid* pair of points, which takes $O(i)$ time (see Lemma 5). Assuming L to be horizontal, the time required for finding the leftmost intersection point on L is $O(ni)$. Similarly, computing the second left-most intersection point needs another $O(ni)$ time. Proceeding similarly, the time complexity of the procedure $\mathcal{A}_0$ of [13] is $O(n^2i^2)$. Similarly, $\mathcal{A}_1$ takes $O(i^2n^{1+\frac{1}{2}}\log n)$ time, and so on. Finally, $\mathcal{A}_k$ takes $O(i^2n^{(1+\frac{1}{k+1})}\log^k n)$ time. Since we have chosen $k = \sqrt{\frac{\log n}{\log\log n}} < \log n$ for the median finding algorithm of [13], we need $O(\log n)$ space in total. Thus, we have the following result:

▶ **Lemma 6.** *The time complexity of the i-th iteration of Constrained_MEC is $O(i^2n^{(1+\frac{1}{k+1})}\log^k n)$, where $1 \leq k \leq \sqrt{\frac{\log n}{\log\log n}}$. The extra space required is $O(\log n)$.*

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 $O(\log n)$. We can further prune the points in the $IndexP$ array using the in-place algorithm for *Constrained_MEC* proposed in Section 3.1. Thus, we have the following result:

▶ **Lemma 7.** *The time complexity of Constrained_MEC is $O(n^{(1+\frac{1}{k+1})}\log^{k+3} n)$, where where $1 \leq k \leq \sqrt{\frac{\log n}{\log\log n}}$. Apart from the input array, it requires $O(\log n)$ extra space.*

Proof. By Lemma 6, the time complexity of the i-th iteration is $O(i^2n^{(1+\frac{1}{k+1})}\log^k n)$, where $i = 1, 2, \ldots, \log n$. Thus, the total time complexity of all the $O(\log n)$ iterations is $O(n^{(1+\frac{1}{k+1})}\log^{k+3} n)$. The extra time required by the in-place algorithm for considering all the entries in the array $IndexP$ is $O(\log n)$ (see Lemma 3), and it is subsumed by the time complexity of the iterative algorithm executed earlier.

The space complexity follows from the fact that the same set of arrays M, D, $IndexP$ and the stack for finding the median can be used for all the $\log n$ iterations, and each one is of size at most $O(\log n)$. ◀

4.2.2 Unconstrained MEC in a read-only setup

As earlier, we use the read-only variation of the *Constrained_MEC* algorithm (described in Subsection 4.2.1) for solving the unconstrained minimum enclosing circle problem. Here we need to maintain three more arrays $\mathcal{M}$, $\mathcal{D}$ and $\mathcal{I}$, each of size $O(\log n)$. $\mathcal{M}[i]$ contains the point of intersection of the vertical and horizontal lines used for pruning points at level i of the algorithm MEC; $\mathcal{D}[i]$ (a two bit space) indicates the quadrant in which the center of the MEC lies. The array $\mathcal{I}$ plays the role of the array $IndexP$ used for *Constrained_MEC*. It is shared by all the iterations of the algorithm.

While checking a point to be *valid* in any iteration of the procedure *Constrained_MEC* at the i-th iteration of the *MEC* algorithm, we first need to check whether it is pruned in any previous iteration of the algorithm *MEC*.

▶ **Theorem 8.** *The minimum enclosing circle of a set of n points in $\mathbb{R}^2$ given in a read-only array can be found in $O(n^{1+\epsilon})$ time and $O(\log n)$ space, where $\sqrt{\frac{\log\log n}{\log n}} < \epsilon < 1$.*

Proof. In the i-th iteration of the algorithm MEC, the time required for Steps 1-3 of the procedure $PRUNE$ is $O(i^2 n^{(1+\frac{1}{k+1})} \log^k n)$ (see the justifications of Lemma 6). In Step 4, the procedure *constrained_MEC* needs $O(n^{(1+\frac{1}{k+1})} \log^{k+3} n)$ time (see Lemma 7). Since the algorithm MEC consists of at most $O(\log n)$ iterations of the procedure $PRUNE$, the overall time complexity of the algorithm is $O(n^{(1+\frac{1}{k+1})} \log^{k+4} n)$. Substituting $\frac{\epsilon}{2} = \frac{1}{k+1}$ and then $n^{\frac{\epsilon}{2}} \geq \log^{4+\frac{1}{\epsilon}} n$, we have time complexity $O(n^{1+\epsilon})$, where ϵ satisfies $\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1$. ◀

5 Conclusion

In this paper, we propose a general prune-and-search technique in read-only memory which can be applied in other problems as well. 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 time-space complexity of MEC in a read-only setting.

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 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.
4 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.
5 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.
6 S. Carlsson and M. Sundström. Linear-time in-place selection in less than 3n comparisons. In *ISAAC*, pages 244–253, 1995.
7 T. M. Chan. Comparison-based time-space lower bounds for selection. In *SODA*, pages 140–149, 2009.
8 D. J. Elzinga and D. W. Hearn. Geometrical solutions for some minimax location problems. *Transportation Science*, 6(4):379–394, 1972.
9 N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.
10 N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM J. Comput.*, 12(4):759–776, 1983.
11 N. Megiddo. The weighted euclidean 1-center problem. *Mathematics of Operations Research*, 8(4):498–504, 1983.
12 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.
13 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.
14 M. I. Shamos and D. Hoey. Closest-point problems. In *FOCS*, pages 151–162, 1975.
15 J. J. Sylvester. A question in the geometry of situation. *Quarterly Journal of Mathemaitcs*, 1:79, 1857.
16 E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *Results and New Trends in Computer Science*, pages 359–370. Springer-Verlag, 1991.