Enoch Lau
SID 200415765

Computational Geometry: An Odyssey

COMP4045 Assignments 3 & 4 in only 26 pages

Contents

Note about notation: In algorithms, ▷ will be used to denote a line containing comments.

1 Question 1: Point Queries in Polygons

1.1 a)

Theorem 1. *Given a convex polygon P as an array of n vertices in sorted order along the boundary, a test of whether a query point q lies inside P can be done in $O(\log n)$ time.*

Proof. Without loss of generality, let us assume that the points are in anticlockwise order (if not, we can simply access the elements of the array in reverse order). Let the points in the array be $p_1, p_2, \ldots, p_n$. Let the first edge p_1p_2 be the *reference edge*; we will measure all angles with respect to this edge (see Figure 1). We will use *sector* in the following way: take the diagonals from p_1 to two adjacent vertices, extend them away from p_1 into half-infinite lines, and the region between the two lines is a sector. Note that because P is convex, it is possible to draw lines to very other vertex from p_1 without intersecting the exterior of P.

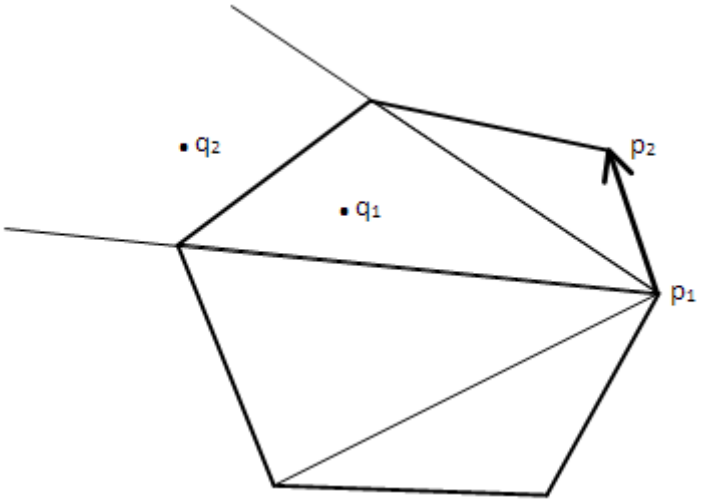

Figure 1: The edge with the arrow p_1p_2 is the *reference edge*. Both q_1 and q_2 lie in the same *sector*. However, only q_1 lies in the polygon.

In general, what we seek to do is to perform a binary search for the sector that q lies in, and then test whether q lies in the triangle bounded by the edge opposite from p_1 and the two rays defining the sector. In the following discussion, when we say *the angle that a point s makes with the reference edge*, we mean the angle between the vectors $\overrightarrow{p_1p_2}$ and $\overrightarrow{p_1s}$, which we can find by using the dot product:

$$\theta = \arccos\left(\frac{\overrightarrow{p_1p_2} \cdot \overrightarrow{p_1s}}{|p_1p_2||p_1s|}\right)$$

Firstly, we test if $q = p_1$; if true, then we report that it is in the polygon immediately, and if false, then we can continue.

To perform the binary search, we first calculate the angle that q makes with the reference edge; call this angle θ_q. Note that because P is convex, if θ is the angle that a vertex $v \neq p_1$ makes with the reference edge, then $0 \leq \theta < \pi$; if there were a value of $\theta \geq \pi$, then the interior angle at p_1 would be greater or equal to π, which contradicts the assumption that P is convex (assuming that there are no collinear edges). Furthermore, because we have been given the vertices in sorted order around P,

the angle that successive vertices make with the reference edge must be non-decreasing; otherwise, the edges will "backtrack" and it will be possible to draw a ray from p_1 that intersects the interior of the polygon twice (Figure 2). Therefore, the mapping from the elements of the array of points to angles relative to the reference edge results in a non-decreasing sequence of values $(\theta_2, \theta_3, \ldots, \theta_n)$; it is therefore possible to perform a binary search for largest $\theta_i \leq \theta_q$. There are several possibilities:

- $\theta_q < 0$ or $\theta_q > \theta_n$: the point lies outside of the possible range of angle values, and q cannot lie inside P.
- $\theta_q = \theta_i$ for some i: the point lies on a half-infinite ray from p_1 to some other point. In this case, q lies inside P iff $\left|p_1 q\right| \leq \left|p_1 p_i\right|$.
- $\theta_i < \theta_q < \theta_{i+1}$ for some i: the point lies in the sector bounded by the half-infinite rays from p_1 to p_i and p_{i+1}. If q lies inside P, then q must lie to the left of $\overrightarrow{p_i p_{i+1}}$, so that it is possible to draw a line from p to q without going over the edge of $\overrightarrow{p_i p_{i+1}}$. To do this, we can use the cross product; q lies inside P iff $\frac{\overrightarrow{p_i p_{i+1}} \times \overrightarrow{p_i q}}{\hat{\mathbf{n}}} \geq 0$, where $\hat{\mathbf{n}}$ is the unit vector pointing out of the page (assuming a right-handed coordinate system).

Figure 2: The two cases that are not permitted because the polygon is convex; as a result, the angles are taken in non-decreasing order around p_1.

The preprocessing steps are all $O(1)$ operations. The binary search takes $O(\log n)$; note that we only need to calculate the mapping from points to angles for the points that we compare against in the binary search to avoid the $O(n)$ cost of converting all the points. The post-processing after we have found the value of θ_i is $O(1)$. In total, the query of whether q lies in P is $O(\log n)$. □

1.2 b)

Theorem 2. *Given a y-monotone polygon P as an array of n vertices in sorted order along the boundary, a test of whether a query point q lies inside P can be done in $O(\log n)$ time.*

Proof. Without loss of generality, we will assume that the vertices are given in ascending order of y-coordinate (if they are in descending order, we can simply access the elements of the array in reverse order). Note that due to y-monotonicity (and the assumption P is simple), the edges $p_i p_{i+1}$ for $i = 1, \ldots, n-1$ all lie towards one side of the polygon, with $p_1 p_n$ running from top to bottom (Figure 3). Firstly, we check if q lies below the first point in the array, or if q lies above the last point

in the array; in either case, the y-coordinate of q lies out of range and it cannot possibly lie within the polygon. We can then perform a binary search for the point in the array that has the largest y-coordinate less than or equal to the y-coordinate of q; call this point p_i.

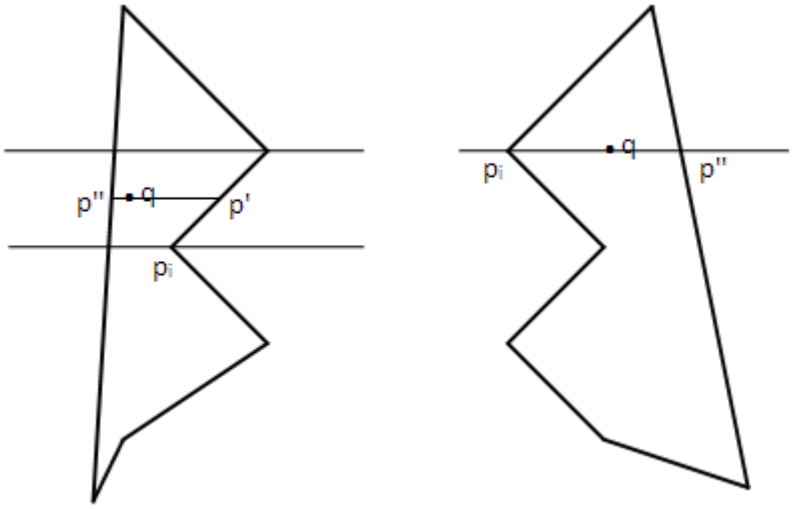

Figure 3: The first polygon has the edge p_1p_n on the left side, and the second polygon has it on the right side. In the first polygon, q lies between two vertices of the polygon in the y direction, and in the second, q is level with one of the vertices. p' in the first polygon is the point on the edge p_ip_{i+1} with the same y-coordinate as q, and p'' in both polygons is the point on the edge p_1p_n with the same y-coordinate as q.

Let p' be the point on the edge p_ip_{i+1} with the same y-coordinate as q (Figure 3). If q has the same y-coordinate as p_i, then $p' = p_i$; otherwise, it can be found by linear interpolation between p_i and p_{i+1}. Let p'' be the point on the edge p_1p_n with the same y-coordinate as q (Figure 3). There are two possibilities as illustrated in Figure 3:

- p_1p_n is the left-most edge of the polygon. Then test to see if $q_x \in [p''_x, p'_x]$.
- p_1p_n is the right-most edge of the polygon. Then test to see if $q_x \in [p'_x, p''_x]$.

The preprocessing can be done in $O(1)$ time. The binary search can be performed in $O(\log n)$ time. The post-processing consists of a series of $O(1)$ operations. In total, the query for whether q lies inside P can be done in $O(\log n)$ time. □

2 Question 2: Bullets

We present an algorithm for finding the index of the segment that is hit by a bullet being shot to the left in Algorithm 1.

Algorithm 1 BULLET-HIT-INDEX(P)

Input: The set of points P sorted in increasing order by x-coordinate
Output: A sequence of the indices hit by the bullets, one for each segment

```
1: l ← {(0,∞)}
2: for all i = 1to|P| do
3:    while l[1][2] < y_i do
4:       Remove the first element from l
5:    end while
6:    indices[i] ← l[1][1]
7:    Add (i, y_i) to the front of l
8: end for
9: return indices
```

The algorithm works by maintaining a list that represents what you can see at any point as you sweep across the points from left to right. l is a list of 2-tuples, with the first element of the tuple being the index of the line segment that you can see, and the second element being the height of that line segment. After iteration i, the contents of l, in order, are the line segments that you can see looking left if you are standing after the ith point, from shortest to the tallest; this is the loop invariant. (The last element of the list is the tuple $(0,\infty)$ so that if we encounter a new line segment that is taller than any other line segments seen thus far, it will report back the index number 0.) Assuming that the loop invariant holds at the beginning of the loop, we remove any line segments that we obscure from view from any position to the right of the current line segment, and add ourselves to the start of l; the loop invariant is re-established because the exit condition of the while loop ensures that the height of the first element of l is greater than or equal to the current line segment's height. The data structure of l will be that of a linked-list, where insertion and deletion at the head of the list is a constant time operation.

Theorem 3. *Algorithm 1 runs in $O(n)$ time.*

Proof. Let us analyse the number of events encountered throughout the execution of the algorithm:

- Initialising l: $O(1)$ run once
- Removing the first element from l: $O(1)$ run $O(n)$ times (because we only ever add n elements to l, and we cannot remove more than we added)
- Calculating the index: $O(1)$ run $O(n)$ times
- Adding a new element to the front of l: $O(1)$ run $O(n)$ times

In total, this gives us $O(n)$ running time. □

3 Question 3: Sizes of Sub-Polygons

Given a simple polygon P with n vertices, we triangulate it, and find its dual graph. The dual graph has vertices of degree at most 3, because each face of the triangulation has at most 3 sides, and it has at least one vertex with degree 1 (the dual graph is a tree). Take one of these vertices with degree 1, say r, and turn the dual graph into a rooted tree with root at r; call this tree T. We define the weight of a (sub)tree $w(\cdot)$ to be the number of nodes contained in the (sub)tree, including the root. If a vertex does not have a left (or right) child, then we say that the weight of the left (or right) child is 0.

If a simple polygon has n vertices, then its triangulation has $n-2$ faces; hence, in order to show that the number of vertices in the sub-polygons P_1 and P_2 are at most $\lceil 2n/3 \rceil$, we need to show the existence of two vertices in T that have weight at most $\lceil 2n/3 \rceil - 2$. Now, note that if one sub-polygon has $\lceil 2n/3 \rceil$ vertices, the other has at least $\lfloor n/3 \rfloor$ vertices; this implies that the two vertices T must also have weight at least $\lfloor n/3 \rfloor - 2$.

We begin this search at the root, r. Set v to r. We examine the children of v to see if either the left or the right has a weight greater than $\lceil 2n/3 \rceil - 2$. If the left child has a weight that exceeds $\lceil 2n/3 \rceil - 2$, then we set v to be the left child and we repeat this process; the same applies for the right child. We will eventually reach a node where the weight of either child does not exceed $\lceil 2n/3 \rceil - 2$; this is true because the weight of a subtree is strictly less than the weight of its parent. Once we have found such a node, we break out of the loop and continue to the next stage.

By this stage, v is a node with children whose weights are at most $\lceil 2n/3 \rceil - 2$. Note that $w(v) > \lceil 2n/3 \rceil - 2$, by construction in the previous stage; v is the first vertex that we have encountered where both children have weights at most $\lceil 2n/3 \rceil - 2$. Furthermore, at least one of the children must have a weight at least $\lfloor n/3 \rfloor - 2$. If both had a weight less than $\lfloor n/3 \rfloor - 2$, then $w(v) < 2(\lfloor n/3 \rfloor - 2) + 1 \leq \lfloor 2n/3 \rfloor - 3$, which contradicts the previous claim that $w(v) > \lceil 2n/3 \rceil - 2$.

Suppose the left child had a weight at least $\lfloor n/3 \rfloor - 2$, then $\lfloor n/3 \rfloor - 2 \leq w(lc(v)) \leq \lceil 2n/3 \rceil - 2$, which is what we wanted. We break P up by the diagonal between v and its left child. The sub-polygon represented by the subtree rooted at the left child of v will have at least $\lfloor n/3 \rfloor$ vertices and at most $\lceil 2n/3 \rceil$ vertices. This means that the other sub-polygon will also have at most $\lceil 2n/3 \rceil$ vertices, as required. Similarly, if the right child has a weight at least $\lfloor n/3 \rfloor - 2$, we break P up by the diagonal between v and its right child.

By construction, we arrive at the following theorem.

Theorem 4. *There exists a diagonal that partitions a simple polygon P into two sub-polygons P_1 and P_2 such that the number of vertices of P_1 and P_2 are at most $\lceil 2n/3 \rceil$.*

4 Question 4: Stabbing Number

An algorithm that computes a triangulation of a convex polygon with stabbing number $O(\log n)$ is presented in Algorithm 2.

Algorithm 2 LOGARITHMIC-STABBING-TRIANGULATION(P)

Input: A convex polygon P with n vertices with the vertices stored in sorted order around the polygon

Output: The internal diagonals of the triangulation of P that has a stabbing number $O(\log n)$

```
1:  Q ← P
2:  T ← ∅
3:  while Q is not a triangle do
4:      n ← |Q|
5:      Q' ← ()        ▷ (an empty sequence)
6:      for all i = 1 to ⌊n/2⌋ do
7:          d ← diagonal(Q[2i − 1], Q[2i])
8:          T ← T ∪ {d}
9:          Append d to the sequence Q'
10:     end for
11:     if n is odd then
12:         Append the edge/diagonal (Q[n − 1], Q[n]) to Q'
13:     end if
14:     Q ← Q'
15: end while
16: return T
```

At each iteration of the while loop, the algorithm takes the polygon Q, which is initialised to the input polygon P, as it exists at the beginning of the loop and adds a diagonal between every second vertex of Q, with a special allowance for an odd number of vertices (we simply add the final edge back in). The algorithm always terminates because the number of vertices in Q becomes smaller with each iteration of the loop.

The loop invariant is that Q is a convex polygon. P is given to be a convex polygon, and the invariant is re-established because at each iteration of the inner for loop, we are simply cutting an "ear" off the polygon; this is equivalent to intersecting the convex polygon with the half-plane consisting of the diagonal extended into an infinite line and the other side of that line from the ear being cut off. Intersections of convex regions results in a convex region, and thus the invariant is maintained.

Consider all the diagonals/edges added to Q' during a particular iteration of the while loop to be part of the same *level*. Suppose we have an arbitrary line l. l can intersect the diagonals of a particular level at most twice, because the diagonals belonging to a particular level define a convex region, and a straight line cannot re-enter a convex region once it has exited. (If l is coincident with a particular diagonal, we will consider l and the diagonal to have intersected once.) Now, $\left|Q'\right|_k = \lceil |Q|_k / 2 \rceil$ for a particular level k. There are thus $O(\log n)$ levels, and l will intersect with the diagonals $O(\log n)$ times. Thus, the stabbing number for a simply polygon triangulated by Algorithm 2 is $O(\log n)$.

5 Question 5: t-Spanners

Note: for this question, I have assumed that $|uw| \leq |uv|$, to allow for the possibility that multiple points are equidistant to u in the same sector. It turns out that the theorems still hold even for the weaker assumption.

5.1 a)

Theorem 5.

$$|uw| + t \cdot |wv| \leq t \cdot |uv|$$

Proof. We first consider the triangle uww', where w' is the projection of w onto uv. Let $\theta = \angle wuw'$. Clearly, $\theta \leq \tau$, since τ is the maximum angle permitted for each sector. By construction, $\angle uw'w = \frac{\pi}{2}$. Therefore, we have:

$$\begin{aligned} \cos\theta &= \frac{|uw'|}{|uw|} \\ \sin\theta &= \frac{|ww'|}{|uw|} \\ \cos\theta - \sin\theta &= \frac{|uw'| - |ww'|}{|uw|} \end{aligned}$$

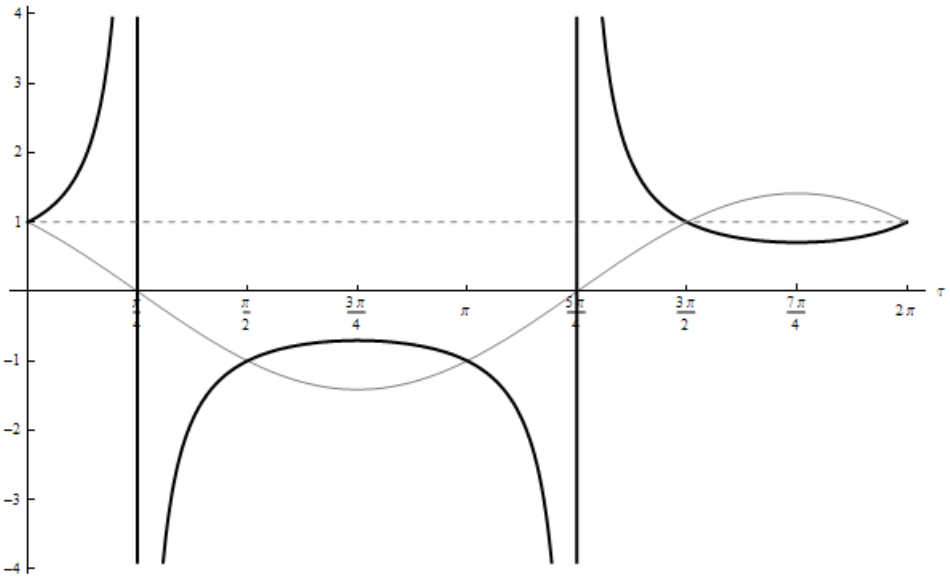

Figure 4: The thick line is $y = \frac{1}{\cos\tau - \sin\tau}$, with the asymptotes at $\frac{\pi}{4}$ and $\frac{5\pi}{4}$ drawn in. The thinner line is $y = \cos\tau - \sin\tau$. The dotted line is $y = 1$.

From Figure 4, we can see that for any value of $t > 1$, there is a corresponding value of τ in the range $\left[0, \frac{\pi}{4}\right)$. Mathematically, this follows from the fact that we can rewrite $\frac{1}{\cos\tau - \sin\tau}$ as $\frac{1}{\sqrt{2}\sin\left(\tau + \frac{3\pi}{4}\right)}$, and then solving for t [needs clarification]. $\frac{1}{\cos\tau - \sin\tau}$ is an increasing function, because

$$\frac{d}{d\tau}\frac{1}{\cos\tau - \sin\tau} = \frac{\cos\tau + \sin\tau}{(\cos\tau - \sin\tau)^2}$$

which is positive for $\tau < \frac{\pi}{4}$.

Hence,

$$\begin{aligned} \frac{1}{\cos\theta - \sin\theta} &\leq \frac{1}{\cos\tau - \sin\tau} \quad \text{for } \theta \leq \tau \\ &\leq t \\ \frac{|uw|}{|uw'| - |ww'|} &\leq t \\ |uw| &\leq t \cdot (|uw'| - |ww'|) \end{aligned}$$

Now, using the triangle inequality in $\triangle ww'v$,

$$|wv| \leq |ww'| + |w'v|$$

Therefore,

$$\begin{aligned} |uw| + t \cdot |wv| &\leq t \cdot (|uw'| + |w'v|) \\ \therefore |uw| + t \cdot |wv| &\leq t \cdot |uv| \end{aligned}$$

□

5.2 b)

An algorithm that takes us from a given u to a given v by following the edges in $G(t)$ is outlined in Algorithm 3.

Algorithm 3 TRAVERSE-T-SPANNER(G, u, v)

Input: The graph G, the starting vertex u and the ending vertex v
Output: A path from u to v

```
Initialise a linked list l to store the path that we will construct
w ← u
while w ≠ v do
    Locate the sector S centred at w in which v lies
    w' ← the nearest point to w in S
    Add ww' to l
    w ← w'
end while
return l
```

The algorithm starts the path at u. It then finds the sector in which the target v lies, and traverses the edge to w, the nearest point to u in that sector; this edge always exists because the sector has at least one point in it (it contains at least v). We then find which sector v is in relative to w, and move towards the vertex closest to w in that sector, and so on. Therefore, the algorithm never "runs out" of edges to follow.

Even if the algorithm never runs out of edges to follow, a problem is that we could end up in a cycle. We will get around this by showing that with each iteration, we become strictly closer (Euclidean distance) to v than we were previously. Let $a = |wv|$, $b = |uw|$, and $c = |uv|$. By the cosine

rule, $a^2 = b^2 + c^2 - 2bc\cos\theta$. Because $b \le c$,

$$\begin{aligned} a^2 &\le 2c^2(1-\cos\theta) \\ &< 2c^2\left(1-\frac{1}{\sqrt{2}}\right) \\ a &< c \end{aligned}$$

where the second inequality is obtained by observing that $\cos\theta > \frac{1}{\sqrt{2}}$ for $\theta < \frac{\pi}{4}$. Hence, by moving to w from u, we strictly decrease the Euclidean distance to v. This forms a strictly monotonically decreasing sequence; cycles are not possible, because that would mean the Euclidean distance to v increases in a particular iteration. Combining this with the fact that $G(t)$ is a finite graph, we conclude that the path from u to v is a finite path without cycles.

To show that $G(t)$ is a t-spanner, we need to show

$$|uv|_G \le t \cdot |uv|$$

where $|uv|_G$ is the distance from u to v by traversing the path $\mathscr{P}$ created by Algorithm 3. We will prove this by induction on the distance away from v along $\mathscr{P}$. Trivially, $|vv|_G \le t \cdot |vv|$. Assume that $\left|w'v\right|_G \le t \cdot \left|w'v\right|$ for some $w' \ne u$ along the path from u to v. Suppose the vertex before w' along the path from u to v is w. Then,

$$\begin{aligned} |wv|_G &= \left|ww'\right|_G + \left|w'v\right|_G \\ &\le \left|ww'\right| + t \cdot \left|w'v\right| \qquad \text{by assumption} \\ &\le t \cdot |wv| \end{aligned}$$

using the result from part a). Therefore, because u is on the path from u to v, $|uv|_G \le t \cdot |uv|$.

5.3 c)

Theorem 6. *The number of edges in $G(t)$ is $O(n)$ for any constant $t > 1$.*

Proof. Fix the value of t. We choose the largest value of τ such that $\frac{1}{\cos\tau - \sin\tau} < t$; for $0 \le \tau < \frac{\pi}{4}$, there is precisely one value, because $\frac{1}{\cos\tau-\sin\tau}$ is strictly increasing (see previous). τ is independent of n. By finding a value of τ, we obtain a value for k, which is also independent of n. At each of the n vertices, we have at most k adjacent edges; we therefore have at most $kn = O(n)$ edges in $G(t)$. □

6 Question 6: Point Queries in Rectangles

6.1 One-Dimensional Case

We first consider the 1D analogue of the problem question. How can we query for all (1D) line segments that intersect a query value q? Let the n line segments be of the form $[x_1, x_2]$, where x_1 is the value at which the line segment starts, x_2 is the value at which the line segment ends, and $x_1 \leq x_2$. Such a line is converted into a 2D point (x_1, x_2), and these n 2D points are stored in a 2D kd-tree, which requires $O(n \log n)$ preprocessing, $O(n)$ storage and $O(n^{1/2} + k)$ query time, where k is the number of reported points.

A query value q is converted into a query rectangle for the kd-tree. Note that q intersects a line segment $[x_1, x_2]$ iff $x_1 \leq q$ and $q \leq x_2$. Thus, we desire all points in the kd-tree where $x \leq q$ and $y \geq q$. The query rectangle is therefore $[x', q] \times [q, y']$, where $x' = \min(x_{\min}, q)$ and $y' = \max(y_{\max}, q)$; $x_{\min}$ is the smallest x value and $y_{\max}$ is the largest y value, both of which can be found in linear time during the kd-tree construction. Note that the region is axis-aligned. After the query, the conversion back into line segments takes an additional $O(k)$ time on top of the kd-tree query time. This results in a total query time of $O(n^{1/2} + k)$, where k is the number of reported line segments.

6.2 Two-Dimensional Case

The 2D case is a trivial extension of the 1D case. Let each rectangle be of the form $[x_1, x_2] \times [y_1, y_2]$. We can convert such a rectangle into a 4D point of the form (x_1, x_2, y_1, y_2). These n 4D points can be placed into a 4D kd-tree, which requires $O(n \log n)$ preprocessing, $O(n)$ storage and $O(n^{3/4} + k)$ query time, where k is the number of reported points.

A query point $q = (q_x, q_y)$ is converted into a 4D query orthotope (generalisation of rectangle and prism). Note that q intersects a rectangle $[x_1, x_2] \times [y_1, y_2]$ iff $x_1 \leq q_x \leq x_2$ and $y_1 \leq q_y \leq y_2$. The query box is therefore $[x'_1, q_x] \times [q_x, x'_2] \times [x'_3, q_y] \times [q_y, x'_4]$, where $x'_1 = \min(x_{1,\min}, q_x)$, $x'_2 = \max(x_{2,\max}, q_x)$, $x'_3 = \min(x_{3,\min}, q_y)$ and $x'_4 = \min(x_{4,\max}, q_y)$; $x_{i,\min}$ and $x_{i,\max}$ are the minimum and maximum values in the ith dimension respectively, all of which can be calculated in linear time during the kd-tree construction. Note that the region is axis-aligned. After the query, the conversion back into rectangles takes an additional $O(k)$ time on top of the kd-tree query time. This results in a total query time of $O(n^{3/4} + k)$, where k is the number of reported rectangles.

7 Question 7: Point-Line Duality

Let the primal plane have axes x and y, and the dual plane have axes a and b. Let $p = (p_x, p_y)$ be a point and $l : y = l_a x + l_b$ be a non-vertical line in the primal plane. $p^* : b = p_x a - p_y$ and $l^* = (l_a, -l_b)$.

7.1 Incidence Preserving

Theorem 7. *The duality transform is incidence preserving: $p \in l$ iff $l^* \in p^*$.*

Proof. $p \in l$ iff $p_y = l_a p_x + l_b$. $l^* \in p^*$ iff $-l_b = p_x l_a - p_y$. The two expressions are trivially equivalent by moving the terms from one side of the equation to the other. □

7.2 Order Preserving

Theorem 8. *The duality transform is order preserving: p lies above l iff l^* lies above p^*.*

Proof. p lies above l iff $p_y > l_a p_x + l_b$. l^* lies above p^* iff $-l_b > p_x l_a - p_y$. The two expressions are trivially equivalent by moving the terms from one side of the equation to the other. □

8 Question 8: Graph Inclusions

We are given that $MNN \subseteq NN \subseteq EMST \subseteq RN \subseteq GG \subseteq DG$.

In order to show that $DG \nsubseteq GG \nsubseteq RN \nsubseteq EMST \nsubseteq NN \nsubseteq MNN$, all that we need to show is that there exists an edge in say, DG that does not exist in GG. (If we are given that $B \subseteq A$, and we need to show that $A \nsubseteq B$, it suffices to show that there exists an $a \in A$ such that $a \notin B$.)

8.1 $DG \nsubseteq GG$

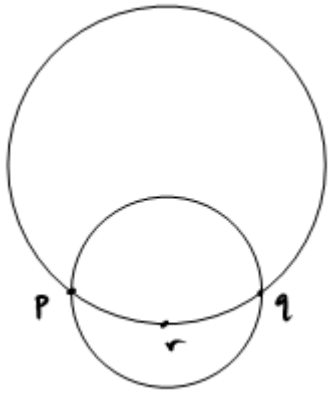

Figure 5: A diagram showing that $DG \nsubseteq GG$

Consider the three points p, q and r in Figure 5. The DG consists of all three edges in the triangle pqr. However, r lies in the circle with diameter pq, and thus pq is not connected in GG.

8.2 $GG \nsubseteq RN$

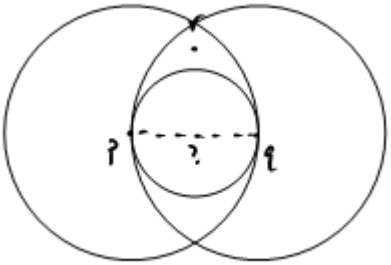

Figure 6: A diagram showing that $GG \nsubseteq RN$

Consider the three points p, q and r in Figure 6. Two points p and q are connected by an edge in GG iff the circle with the diameter pq does not contain any other point of P in its interior. In this case, pq is connected in GG, because r lies outside this circle. However, $|pq| > \max(|pr|, |qr|)$, and thus pq is not connected in RN.

8.3 $RN \nsubseteq EMST$

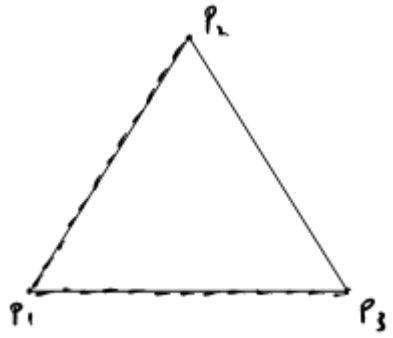

Figure 7: A diagram showing that $RN \nsubseteq EMST$

In Figure 7, we have an equilateral triangle. RN has an edge p_2p_3 that is not in $EMST$. Because $|p_1p_2| = |p_2p_3| = |p_3p_1|$, all of those edges exist because they all satisfy the inequality that defines an RN. For example, for p_1 and p_2,

$$|p_1p_2| \leq \min_{r \in P, r \neq p_1p_2} \max(|pr|, |qr|)$$

On the other hand, an $EMST$ will only have 2 of the three edges, because it suffices to connect the three points together. Any two edges will do, because it is an equilateral triangle.

8.4 $EMST \nsubseteq NN$

Figure 8: A diagram showing that $EMST \nsubseteq NN$

In Figure 8, we have points on a straight line. $EMST$ has an edge p_2p_3 that is not in NN. p_2 and p_3 have their nearest neighbours as p_1 and p_4, while the edge p_2p_3 is essential for the $EMST$, which is always connected.

8.5 $NN \nsubseteq MNN$

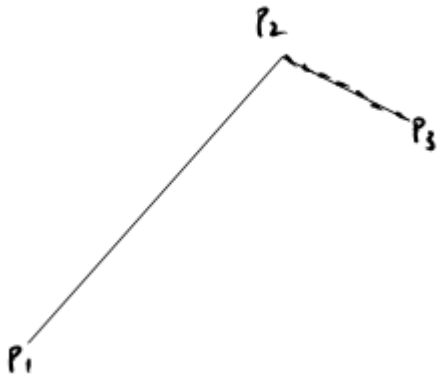

Figure 9: A diagram showing that $DG \nsubseteq GG$

In Figure 9, NN has an edge p_1p_2 that does not exist in MNN. p_1p_2 is not in MNN because while p_2 is the closest point to p_1, p_1 is not the closest point to p_2.

9 Question 9: Minimum Diameter Spanning Tree

9.1 a)

We present an algorithm that computes a minimum diameter monopolar spanning tree (MDMST) in Algorithm 4.

Algorithm 4 MINIMUM-DIAMETER-MONOPOLAR-SPANNING-TREE(P)

Input: A set of n points in the plane, P
Output: A minimum diameter monopolar spanning tree, and its diameter

1: **if** $|P| \leq 2$ **then**
2: **return** complete graph of P
3: **end if**
4: $min \leftarrow \infty$
5: **for all** points $p \in P$ **do**
6: ▷ *Find the two points in P that are furthest away from p*
7: $q_1 \leftarrow (\infty, \infty)$
8: $q_2 \leftarrow (\infty, \infty)$
9: **for all** points $q \neq p \in P$ **do**
10: **if** $|pq| \geq |pq_1|$ **then**
11: $q_2 \leftarrow q_1$
12: $q_1 \leftarrow q$
13: **else if** $|pq| \geq |pq_2|$ **then**
14: $q_2 \leftarrow q$
15: **end if**
16: **end for**
17: **if** $|pq_1| + |pq_2| < min$ **then**
18: $|pq_1| + |pq_2| = min$
19: $p_{\min} \leftarrow p$
20: **end if**
21: **end for**
22: **return** a MDMST centred at $p_{\min}$, by creating an edge from $p_{\min}$ to every point $q \neq p_{\min} \in P$, and the diameter min

For the small cases $n \leq 2$, the MDMST happens to be conveniently expressed as the complete graph (the single vertex for $n = 1$ and an edge joining the two vertices for $n = 2$).

For $n \geq 3$, for an MDMST centred at a given point p, the maximum diameter is obtained by adding the distances from p to the two points furthest away from p; this is obviously true, because using any lower-ranked distance cannot increase the total distance. Each iteration of the outer for loop computes the maximum diameter for a particular p; we find the minimum of this maximum diameter over all p, as per the definition of a MDST. After finding $p_{\min}$, the centre of the MDMST with minimal diameter, we create and return the tree.

There are n points, and we perform $n-1$ lots of $O(1)$ operations in the inner loop per point, resulting in a time complexity of $O(n^2)$ for the main loop. Tree construction takes $O(n)$ time, as there are $n-1$ edges to construct. In total, we have a time complexity of $O(n^2)$.

9.2 b)

We present an algorithm that compute a minimum diameter dipolar spanning tree (MDDST) in Algorithm 5.

Algorithm 5 MINIMUM-DIAMETER-DIPOLAR-SPANNING-TREE(P)

Input: A set of n points in the plane, P
Output: A minimum diameter dipolar spanning tree, and its diameter

1: **if** $|P| \leq 2$ **then**
2: **return** complete graph of P
3: **end if**
4: ▷ *Pre-compute sorted lists of points*
5: **for all** points $p \in P$ **do**
6: $sorted[p] \leftarrow$ a list of points $q \neq p \in P$ sorted on increasing distance from p
7: **end for**
8: ▷ *Try each pair of points*
9: $d'_{\min} \leftarrow \infty$
10: **for all** points $p \in P$ **do**
11: **for all** points $q \neq p \in P$ **do**
12: $S_{-q} \leftarrow sorted[p]$ with q removed
13: ▷ *Pre-compute distances to q*
14: $Q[n-2] \leftarrow \left|S_{-q}[n-2], q\right|$
15: **for all** $i = n-3$to1 **do**
16: $Q[i] \leftarrow \max(\left|S_{-q}[i]\right|, Q[i+1])$
17: **end for**
18: ▷ *Attach everything to q, and then in increasing order of distance from p, re-attach them to p*
19: $d_{\min} \leftarrow \infty$
20: **for all** $i = 1$to$n-2$ **do**
21: $d \leftarrow \left|S_{-q}[i], p\right| + \left|pq\right| + Q[i]$
22: **if** $d < d_{\min}$ **then**
23: $d_{\min} \leftarrow d$
24: $i_{\min} \leftarrow i$
25: **end if**
26: **end for**
27: **if** $d_{\min} < d'_{\min}$ **then**
28: $d'_{\min} \leftarrow d_{\min}$
29: $i'_{\min} \leftarrow i_{\min}$
30: $poles \leftarrow \{p, q\}$
31: **end if**
32: **end for**
33: **end for**
34: **return** a dipolar spanning tree constructed with the poles at *poles*, and with all points $S_{-q}[i]$ for $1 \leq i \leq i'_{\min}$ attached to the first pole, and the remainder attached to the second pole; and the diameter $d'_{\min}$

The algorithm works as follows. We first handle the degenerate cases as before. We go through every possible pair of dipoles p, q and consider which set of dipoles gives us the smallest diameter. We first compute S_{-q}, which is a list of points excluding p and q sorted in increasing distance away

from p. We then pre-compute the maximum distance to q of any point in $S_{-q}[k]$ for $j \le k \le n-2$, for all $1 \le j \le n-2$; this is used to allow us to calculate the diameter in $O(1)$ time in a later stage. Initially, we attach all points in S_{-q} to q; we then re-attach them one by one to p, in increasing order of distance from p. The diameter can be calculated as the distance from the new point to p (say r), plus $|pq|$, plus the maximum distance from q to any of its remaining adjacent vertices, which can be looked up in constant time from the pre-computed table. We do not get a longer diameter by considering the path from the new point attached to p to another pre-existing point attached to p, because all the points attached to p are at most a distance r away from p, while all points attached to q are at least a distance r away from p. Finally, we compute the dipolar spanning tree by re-constructing the state of the attachments at the point when we encountered the smallest diameter.

Pre-computing the sorted list of points takes $O(n^2 \log n)$ time. We have $O(n^2)$ pairs of points to consider. For each pair of points, we perform the following:

- Create S_{-q}: this can be done in $O(n)$ time
- Pre-compute distances to q: there are $O(n)$ distances to compute, each of which takes $O(1)$ time
- Re-attach points one by one to p: there are $O(n)$ re-attachments, each of which takes $O(1)$ time

Reconstructing the dipolar spanning tree takes $O(n)$ time, as there are $O(n)$ edges to create. In total, we will take $O(n^3)$ time.

9.3 c)

We acknowledge the assistance of a certain paper[1] *in guiding us to the solution for this problem. We make use of Lemma 9 in formulating the approximation algorithm, which appears as Lemma 10; the paper however, does not provide full details.*

Lemma 1 (Lemma 9 in the paper)**.** *Let $(A_1, B_1), \ldots, (A_l, B_l)$ be a WSPD of P w.r.t. τ and let p and q be any two points in P. Then there is a pair (A_i, B_i), such that for every point $u \in A_i$ and every point $v \in B_i$, the inequality $diam S_{uv} \le (1 + 8/\tau) \cdot diam S_{pq}$ holds. (S_{pq} denotes a spanning tree with dipole $\{p, q\}$ whose diameter is minimum among all such trees.)*

Proof. By the definition of a WSPD, there is a pair (A_i, B_i) such that $p \in A_i$ and $q \in B_i$. Take any point $u \in A_i$ and any point $v \in B_i$. We construct a tree T with poles u and v as follows. Join u and v, p with each of its neighbours in S_{pq} except for q, and q with each of its neighbours in S_{pq} except for p. As proved in class, $|uv| \le (1 + 4/\tau)|pq|$.

For the tree T, let r_u be the length of the longest edge of T incident to u, excluding uv, and define r_v similarly. By the triangle inequality, we can bound it by considering the path from u to a neighbour via p: $r_u \le |up| + r_p$. As proved in class, for two points p, p' in the same ball, $|pp'| \le |pq|/\tau$. Hence, $r_u \le r_p + 2|pq|/\tau$, and similarly, $r_v \le r_q + 2|pq|/\tau$. From the definition of a WSPD, the longest path must span both balls, and thus, $\operatorname{diam} T = r_u + |uv| + r_v$.

[1] J. Gudmundsson, et al. *Facility location and the geometric minimum-diameter spanning tree.* Computational Geometry 27 (2004) 87-106

Hence,

$$\begin{aligned}\operatorname{diam} T &\leq \left(r_p + 2\frac{|pq|}{\tau}\right) + \left(|pq| + 4\frac{|pq|}{\tau}\right) + \left(r_q + 2\frac{|pq|}{\tau}\right) \\ &= (r_p + |pq| + r_q) + 8\frac{|pq|}{\tau} \\ &= \left(1 + \frac{8}{\tau}\right)\operatorname{diam} S_{pq}\end{aligned}$$

If we take the minimum S_{uv} over all such trees T, we obtain $\operatorname{diam} S_{uv} \leq (1 + 8/\tau) \cdot \operatorname{diam} S_{pq}$. □

We now present Algorithm 6, which computes a dipolar tree T with $\operatorname{diam} T \leq (1+\epsilon) \cdot d_{\min}$ in $O\left(\frac{n^2}{\epsilon^2} \log n\right)$ time, where $d_{\min}$ is the diameter of a minimum diameter dipolar spanning tree.

Algorithm 6 APPROXIMATE-DIPOLAR-TREE(P, ϵ)

Input: A set of n points P, and an approximation factor ϵ
Output: A dipolar tree T with at most $1+\epsilon$ times the diameter of the optimum tree

1: $d'_{\min} \leftarrow \infty$
2: **for all** pairs (A_i, B_i) in the WSPD of P w.r.t. $\tau = 8/\epsilon$ **do**
3: Choose any $p \in A_i$
4: Choose any $q \in B_i$
5: $S_{-q} \leftarrow P \backslash \{p, q\}$ sorted by distance from p
6: ▷ *Pre-compute distances to q*
7: $Q[n-2] \leftarrow \left|S_{-q}[n-2], q\right|$
8: **for all** $i = n-3$to1 **do**
9: $Q[i] \leftarrow \max(\left|S_{-q}[i]\right|, Q[i+1])$
10: **end for**
11: ▷ *Attach everything to q, and then in increasing order of distance from p, re-attach them to p*
12: $d_{\min} \leftarrow \infty$
13: **for all** $i = 1$to$n-2$ **do**
14: $d \leftarrow \left|S_{-q}[i], p\right| + |pq| + Q[i]$
15: **if** $d < d_{\min}$ **then**
16: $d_{\min} \leftarrow d$
17: $i_{\min} \leftarrow i$
18: **end if**
19: **end for**
20: **if** $d_{\min} < d'_{\min}$ **then**
21: $d'_{\min} \leftarrow d_{\min}$
22: $i'_{\min} \leftarrow i_{\min}$
23: $poles \leftarrow \{p, q\}$
24: **end if**
25: **end for**
26: **return** a dipolar spanning tree constructed with the poles at *poles*, and with all points $S_{-q}[i]$ for $1 \leq i \leq i'_{\min}$ attached to the first pole, and the remainder attached to the second pole; and the diameter $d'_{\min}$

This algorithm computes goes through each possible pair (A_i, B_i) in the WSPD and it computes the minimum diameter for some $u \in A_i$ and $v \in B_i$. The tree with the minimum diameter encountered is chosen to be the returned answer. This works because, according to Lemma 9 (of the paper),

there always exists a pair (A_i, B_i) such that regardless of which two points we pick, the diameter of the resultant minimum diameter tree, say S_{uv}, is at most $1 + 8/\tau$ times the diameter of a tree with dipoles p, q for any two points p, q in P. For this to be always true, $\text{diam}S_{uv} \leq (1 + 8/\tau) \times d_{\min} = (1 + \epsilon) \times d_{\min}$, and thus the smallest one we encounter must be able to satisfy that inequality.

In Algorithm 6, we iterate over all possible pairs. From the lectures, there are $O(\tau^d n)$ such pairs, and for $d = 2$ (two dimensions), we have $O(\tau^2 n)$ pairs. For each pair, we perform a sort in $O(n \log n)$ and then create S_{pq} in $O(n)$ time. In total, we consume $O(\tau^2 n^2 \log n) = O\left(\frac{n^2}{\epsilon^2} \log n\right)$ time.

Note that Algorithm 6 only computes an approximate minimum diameter dipole spanning tree. However, the minimum diameter monopole spanning tree can be computed in $O(n^2)$ time, by part a), and the time complexity of Algorithm 6 dominates this. Hence, we formulate the complete approximation algorithm in 7, which runs in $O\left(\frac{n^2}{\epsilon^2} \log n\right)$ time (the diameters are returned straight from the functions called).

Algorithm 7 APPROXIMATE-MINIMUM-DIAMETER-SPANNING-TREE(P, ϵ)

Input: A set of n points P, and an approximation factor ϵ

Output: A minimum diameter spanning tree with at most $1 + \epsilon$ times the diameter of the optimum tree

```
1: m ← MINIMUM-DIAMETER-MONOPOLAR-SPANNING-TREE(P)
2: d ← APPROXIMATE-DIPOLAR-TREE(P, ε)
3: if diameter of m < diameter of d then
4:     return m
5: else
6:     return d
7: end if
```

10 Question 10: Line Intersections

Theorem 9. *n lines in the plane can result in at most $\frac{n(n-1)}{2}$ intersections.*

Proof. Each of the n lines can participate in at most $n-1$ intersections, because two straight lines may only intersect once (for a pair of coincident lines, treat that as one intersection). This results in a count of at most $n(n-1)$ intersections. However, we have counted each intersection at least twice (because an intersection necessarily involves at least two lines), and therefore we have at most $\frac{n(n-1)}{2}$ intersections for a set of n lines. □

11 Question 11: Line with Maximal Points

We describe an algorithm to find the line containing the maximum number of points in S, a set of n points in the plane in Algorithm 8.

Algorithm 8 MAXIMUM-POINTS-LINE(S)

Input: A set S of n points in the plane
Output: The line that contains the maximum number of points in S

▷*Find the non-vertical line that contains the most number of points*
$S' \leftarrow$ the set of lines dual to the points in S
count $\leftarrow$ a hash table that maps (machine-precision) points to integer counts
for all lines $l \in S'$ **do**
  for all lines $m \neq l$ in S' **do**
    if gradient of $l \neq$ gradient of m **then**
      $i \leftarrow$ intersection of l and m
      if *count* already has an entry for i **then**
        $count[i] \leftarrow count[i] + 1$
      else
        $count[i] \leftarrow 1$
      end if
    end if
  end for
end for
Go through the entries of *count* and find the intersection i with the highest count, say c_1. Let l_1 = the primal line of i.
▷*Find the vertical line that contains the most number of points*
vcount $\leftarrow$ a hash table that maps (machine-precision) x-coordinates to integer counts
for all points $p \in S$ **do**
  if *vcount* already has an entry for p_x **then**
    $vcount[p_x] \leftarrow vcount[p_x] + 1$
  else
    $vcount[p_x] \leftarrow 1$
  end if
end for
Go through the entries of *vcount* and find the value of x, say x_1 with the highest count, say c_2. Let l_2 = the vertical line $x = x_1$.
▷*Merge step*
if $c_1 > c_2$ **then**
  return l_1
else
  return l_2
end if

The algorithm makes use of the fact that intersections between lines in the dual plane correspond to points lying on the same line in the primal plane. For each point on a line that has at least two points on it (in the primal plane), the point maps to a line in the dual plane that goes through an intersection; the number of lines going through the intersection is equivalent to the number of points lying on the line in the primal plane that corresponds to the intersection point in the dual

plane. From question 10, we know that there are $O(n^2)$ intersections for n lines; each one of these intersections is generated and inspected by the first nested loop. The insertion into the hash table takes (amortised) $O(1)$ time. Finding the intersection with the highest count means that we have to examine $O(n)$ entries in the hash table. In total, finding the non-vertical line with the most number of points going through it takes $O(n^2)$ time.

We need to treat vertical lines separately, because they have no dual analogue. Points that lie on the same vertical line in the primal plane appear as parallel lines on the dual graph, and thus they have no intersection in the plane. Instead, we go through each point and count how many times a particular x coordinate is seen. This part runs in $O(n)$ time, because there are n points to examine, and the insertion into a hash table takes (amortised) $O(1)$ time.

We then compare the optimal non-vertical line with the optimal vertical line, and return the line with the highest number of intersections. In total, therefore, the algorithm runs in $O(n^2)$ time.

We conclude with a comment on the use of hash table. We assume that it is possible to hash real numbers and real-valued points for use in a hash table. This can be done if we assume that the real numbers and real-valued points are only represented to machine-precision (and not to arbitrary precision). This is fine, as long as the smallest change in x or y coordinate is not smaller than what can be distinguished; if not, we can simply scale the axes accordingly to enlarge any differences in values.

12 Question 12: Monotonicity of TSP and MST

12.1 a)

Theorem 10. *If $S \subseteq S'$, then the length of $TSP(S) \leq$ the length of $TSP(S')$.*

Proof. Let us start with S' and find a solution to the TSP for it. If $S \subseteq S'$, there are three possibilities:

- $S = S'$: then the theorem is true trivially
- $|S| = |S'| - 1$: we remove a point, say u, from S' to obtain S. Let u be the city before t in the tour and v be the city after u; if u is the first city in the tour, set t to be the last city, and if u is the last city, set v to be the first city. We remove u, and join up t to v. This results in a tour that visits each city once and returns to the original city, and by the triangle inequality, $|tv| \leq |tu| + |uv|$, the path from t to v cannot be longer than the original route via u.
- S contains multiple points fewer than S': just remove the points one at a time using the above, and in total, the length of the TSP cannot increase at any stage.

□

12.2 b)

Theorem 11. *If $S \subseteq S'$, the length of $MST(S)$ is not necessarily less than or equal to the length of $MST(S')$.*

Proof. We show an example where the theorem is true.

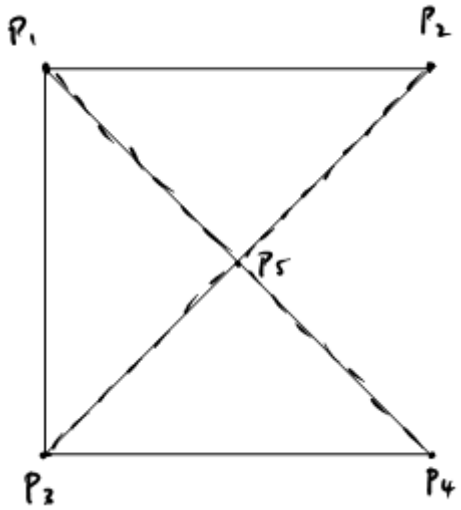

Figure 10: An example where the theorem is true

Consider Figure 10. We place 4 points, $p_1, \ldots, p_4$ on the corners of a square of unit side length. The MST consists of three edges p_1p_2, p_1p_3 and p_3p_4. This has a length of 3 units. (There is no shorter way to join the four points; any tree involving a diagonal can have the diagonal replaced by

a horizontal or vertical edge.) We then add p_5 into the middle of the square. The MST now consists of four diagonal lines p_1p_5, p_2p_5, p_3p_5 and p_4p_5. This has a total length of $4\frac{1}{\sqrt{2}} < 3$. □