Geometry Homework 1

Andor Vári-Kakas

October 26, 2022

Throughout my solution I will assume that all graphs are connected, using the induced restricted (but easily extensible) definitions.

1 Task 1

(a) Let's show the two directions separately.

First that if there is a C embedding with edges only above the axis then it is outerplanar. A graph is outerplanar if all the vertices are on the boundary of the outer face. If a graph can be drawn only with edges above the x axis, then there are no edges below the x axis, hence, below the vertices. Therefore, all the vertices are visible (reachable without a crossing) from below the x axis, i.e. they are on the boundary of the infinite outer face.

The more complicated direction is the other one. My proof consists of three steps:

(i) Show that if a graph is outerplanar with the boundary of the outer face being a cycle, then it is also C-embeddable with edges above the x-axis

(ii) Argue that removing edges cannot make the graph non-C-embeddable with edges above the x-axis.

(iii) Show that edges can be added to any outerplanar graph such that the outer face becomes a cycle, and the graph remains outerplanar.

Then, combining the all the points, any outerplanar graph can be C-embedded using edges above the x axis by first extending it to have a cyclic outer face, then embedding it, and then removing the added edges.

(i) Let the vertices on the outer face/cycl order in anticlockwise be $v_1, v_2, ..., v_n$. Using the argument proven Theorem 2.44 of the lecture notes, there exists a vertex which has no incident chords. WLOG let it be v_n. Note that while any circular rotation of this order would work, let's embed the vertices on the x axis in the order $v_1, v_2, ..., v_n$ for simplicity. Then, visually, the embedding happens in the following way:

- temporarily cut out the $v_1 - v_n$ edge
- "unfold" all the vertices in clockwise order onto the x axis (imagine that by cutting out the $v_1 - v_n$ edge the whole structure falls apart and falls onto the x axis)
- slightly "bend" all the edges of the outer face upwards to be above the x axis (before this step they were straight lines on the x axis)
- draw back $v_1 - v_n$ above the whole drawing at that point in a C-shape

Formally speaking, we can just put vertex v_i at coordinate i, and make the edge $v_i - v_j$ be a half-circle with radius $\frac{|i-j|}{2}$.

It only remains to argue that this is a plane drawing (i.e. there are no crossings). It is easy to see visually from the description of the embedding process. Formally speaking, there are no vertices v_i, v_j, v_k, v_l such that $i < j < k < l$ and $v_i - v_k$ and $v_j - v_l$ are edges, since then $\{i, j, k, l\}$, using in addition the edges of the outer face would form a subdivision of K_4 which is forbidden for outerplanar graphs. Hence, the only pairs of edges $v_i - v_j$ and $v_k - v_l$ are only between $i < j < k < l$ or $i < k < l < j$. In both of those cases, the half circle edges of the embedding clearly don't cross.

(ii) This is trivial – removing any edge cannot induce extra crossings and all the previous edges remain intact (in particular remain above the x axis).

(iii) Let the vertices on the outer boundary in (clockwise) order be $v_1, v_2, ..., v_k$ ($k \geq n$), where a vertex can be present multiple times in this sequence if the outer face is not a cycle. Let's add edges using the following procedure:

```
visited = []
i = 1
while i<=k:
    j = i + 1
    while (visited[j] && j <= k) j++;
    if(j == k + 1) j = 1;
    if(no edge between v_i and v_j) add_edge(v_i, v_j)
    visited[i] = true
    i = j
```

Basically, we are adding shortcut edges along the way to avoid repeating the same vertex multiple times on the outer boundary.

(b) Let's show the two directions separately.

First that if there is a C-embedding of a graph G, then it is a subgraph of a Hamiltonian planar graph. Let's take a C-embedding with order of vertices on the x-axis being $v_1, v_2, ..., v_n$. Let's create a new graph (and directly its corresponding embedding) G', by for each $1 \leq i < n$, if there is no edge $v_i - v_{i+1}$, add this edge. This edge can be added to the drawing without violating planarity and the CS rules, by drawing it "under" all edge going out from v_i forwards and v_{i+1} backwards. This works, because no (other) edge crosses the x axis so there is space under every edge, and the derivative of the curve at v_i and v_{i+1} can always be made smaller any other curve so far. If it is not already there, let's also add an edge $v_1 - v_n$. This is possible, by kind of the complement of the above argument, drawing this edge "above" the finitely many already existing edges. Then, the resulting drawing is a Hamiltonian plane graph. It is plane, because as we argued throughout the process, we never added a crossing while adding edges. It is Hamiltonian, as the cycle $v_1, v_2, ..., v_n$ has all the edges.

For the other direction, consider an embedding of the graph G. Since it is a subgraph of a Hamiltonian planar graph, we can extend the graph and the embedding by drawing the remaining edges to make it Hamiltonian. Note that planarity remains. The (/a) Hamiltonian cycle in this embedding is a Jordan curve, separating the regions inside and outside the cycle. Letting E_on be the edges on the Hamiltonian cycle $v_1, v_2, ..., v_n$, E_out be the edges outside, and E_in be the edges inside, we can observe that both $E_on \cup E_in$ and $E_on \cup E_out$ have the cycle $v_1, v_2, ..., v_n$ as one of its faces. While in the latter case, this cycle is not (necessarily) on the outer face, by using Theorem 2.2 of the lecture notes, we can obtain a plane embedding of both $E_on \cup E_in$ and $E_on \cup E_out$ which have the cycle as the outer face boundary. Since these cycles are Hamiltonian, we get outerplane graphs. Then, we use part a) to obtain C embedding with edges above the line for both graphs. A crucial point as we will see later in parts c) and d) is that it is possible to obtain the two embeddings such that the order of the vertices is the same on the x-axis. Note that the algorithm I presented in part a) does not guarantee this, but as I noted, any C-embedding with edges above the x-axis can be circularly rotated, and since both have the same order of vertices up to a circular rotation, the two can be aligned. (To show the circular rotation lemma, observe that the last vertex from the x-axis can be moved to become the first one, since all its edges go above all the other edges). To combine the two, we remove E_on from one of the two, and flip one along the x-axis. The resulting graph has only C-edges by construction, and it is planar, since both of the components are planar by part a) and they do not intersect as they are separated by the x-axis. To be formal, we have to at the end delete those edges which were not present in the original graph but we added them.

(c) Let the original graph be G_0 and the subgraph provided G_1. Assume G_0 has a Hamiltonian cycle. Since the added nodes from G_1 to G_0 only have neighbours in the set of vertices already in G_1, any Hamilton cycle passing through them also have to pass through 2 edges connecting them to vertices in G_1. There are 6 added vertices, and since each of the aforementioned edges have an endpoint among the vertices of G_1, the total degree of the vertices of G_1 in any Hamiltonian of G_0 is at least $6 \cdot 2 = 12$. Hence, my the pigeonhole principle, there is at least one vertex in G_1 that has at least 3 neighbours in the Hamiltonian cycle. However, in a (Hamiltonian) cycle every vertex needs to have degree 2, which is a contradiction.

Note: First I thought I should use part b) in this part, and my idea was to show that the graph on Figure 2 is not C-embeddable, and hence, by using part b), the original graph is not Hamiltonian. But the problem is that the graph on Figure 2 is C-embeddable... What is even more exciting is that one can show it is C-embeddable because it has a Hamilton cycle, and we can use part b)...

(d) See my partitioning in Figure 1. Note that this produces the two required outerplanar graphs, but to see that they are really outerplanar, and to actually embed them, one simply has to turns outwards the edges that go from bottom corners of the big triangle and lead to a leaf. This way, those leaf nodes will also be on the outer face. (There was some freedom in the partitioning, e.g. even this construction can be directly modified by changing the color of the bottom or the vertical edges arbitrarily.

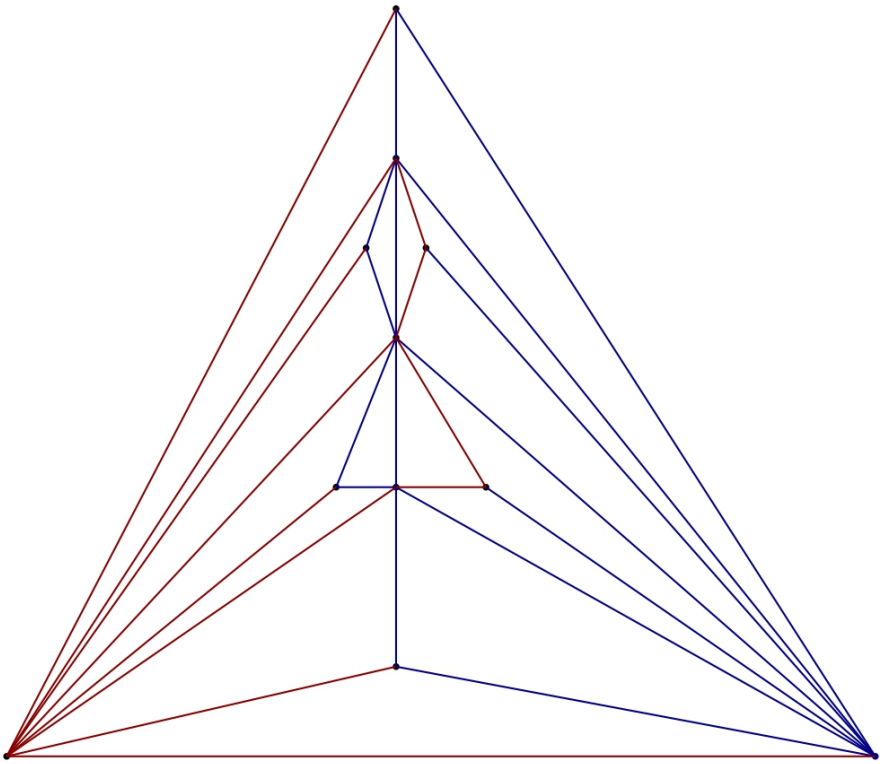

Figure 1: Partitioning for Question 1d, using 2 colors

2 Task 2

(a) Since v_1, v_2, v_n form a facial triangle and G is planar, by Theorem 2.2 there is an embedding of G where v_1, v_2, v_n bounds the outer face. Let's try to find a canonical ordering for such an embedding (note that embeddings of maximal planar graphs are combinatorially unique since they are 3-connected as shown in the exercise classes and we can use Whitney's theorem, but we will not make use of this observation). Observe that from here onwards, even though we don't have the actual embedding available, essentially the same algorithm as presented in the lectures for plane graph canonical orderings works. While it does make use of the actual embedding at a few places, they are very easy to avoid, and works perfectly find having only the outer face and the graph itself. (Relatedly, we cannot have a DCEL, but again, the algorithm presented only uses limited functionality of the DCEL which we can reproduce using our adjacency list as well). I will assume that I can take any edge from the outer cycle as "fixed" edge v_1v_2 from the definition of canonical ordering. In our case it can be $v_1 - v_2$, $v_1 - v_n$ or $v_2 - v_n$ – WLOG let's take $v_1 - v_2$. The best ways to illustrate my points above is I think to provide some pseudocode, highlighting how the algorithm adapts to this scenario:

```
initialise chord counter vector to 0s //since the starting outer face is a triangle it has no
    chords even for the 3 vertices on it
initialise canonical ordering vector to empty
initialise list of eligible vertices to [v_n]
initialise pointers to the eligibility list, null for each vertex except for v_n // I will omit
     the details of the pointer manipulations from now on, as it is some basic data structure
    thing independent of the point of this exercise. I just note that every operation can be
    done in constant time

initialise boolean vector of removed vertices // this is used instead of explicitly removing
    vertices from the adjacency list which is expensive
initialise boolean vector for vertices on the outer cycle // we will not bother changing back
    booleans to false after they are removed from the graph


def change_counter(x, delta): // function for maintining the eligibility list (and the pointers
     as well, omitted) when changing chord counter
```

```
        if x != v_1 and x != v_2: // this is required, as in this implementation we will increase
            and then decrease back the counter of v_1 and v_2 in the else branch of the while loop,
            whereas nothing is really changing for them, it is just an artefact of the required
            workaround (of course, we do not want to remove v_1 and v_2 following the definition of
            canonical orderings)
            if -delta == counter[x]:
                add x to eligibility list
            else if counter[x] == 0:
                remove x from eligibility list
        counter[x] += delta


    while G has more than 2 vertices:
        pop an arbitrary vertex v from the eligibility vector
        add v to the canonical ordering
        if v has only two (non-removed) neighbours u_1 and u_2 in the adjacency list:
            change_counter(u_1, -1), change_counter(u_2, -1) // note that this doesn't make much
                sense on the last step when u_1=v_1 and u_2=v_2 but then it no longer matters
        else:
            let the (non-removed) neighbours be u'_1, u'_2, ..., u'_k
            using the boolean vector for outer cycles find u_1 and u_2 that are on the outer cycle
                from the neighbours
            let the remaining neighbours be u_3, u_4, ..., u_k
            for each of them set their outer cycle flag to true
            for each edge incident to u_3, u_4, ..., u_k:
                if the other endpoint is (also) on the outer cycle and (the other endpoint is not a
                    neighbour of v or has has an index larger than the original endpoint):
                    // we need the second condition to avoid visiting the same edge between
                        neighbours of v twice
                    change_counter(endpoint_1, 1), change_counter(endpoint_2, 1)
            for each u in u_3, u_4, ..., u_k:
                change_counter(u, -2) // this is because edges between neighbouring vertices on the
                    new outer face are not chords, but since we don't have combinatorial
                    information, we could not know in the previous for loop in which order are the
                    u_3, u_4, ..., u_k located, i.e. which edges are indeed chords
            change_counter(u_1, -1), change_counter(u_2, -1)  // same reason

        disable v in the boolean removed vertices vector



    push_back v_0 and v_1 to the canonical ordering
    reverse the canonical ordering
```

While there are some details of the algorithm I did not elaborate on, I hope I made it clear enough that it could be formalised and that all the obstacles faced by not having an embedding to start with can be tackled. Also note that the algorithm remains linear – the only thing to argue is whether not explicitly removing the vertices can cause any trouble, but it cannot, as still each edge is touched a constant number of times.

(b) There are several ways the algorithm can fail/lead to unexpected behaviour if the preconditions are not satisfied. I will highlight some main points among the many possible scenarios:

- The graph G is not planar. In this case, at some point of the iteration, the algorithm would find no eligible vertices, so it would get stuck. In particular, it is possible in a non-planar graph that all vertices of the outer face are incident to a chord.
- The graph G is planar, but not maximal planar. In this case, the graph is not internally triangulated, hence, no canonical ordering exists. Nevertheless, the algorithm per se does not necessarily get stuck, it can return non-sense results (which can be filtered out in part c)). In particular, the chord counting trick (going through all relevant edges and then subtracting 1 or 2) assumes maximal planarity, and the existence of edges in sequence between each neighbour of v. An example run where the algorithm fails is taking the graph G with 4 vertices v_1, v_2, v_n on the

outer triangle, and v_3 in the middle with a single edge to v_n. Then, after removing v_n, the chord counters will be $-1, -1, -2$, so no vertex will become eligible and the algorithm gets stuck.

- The vertices v_1, v_2, v_n don't form a triangle. In this case, a canonical order might still exist: take e.g. the graph on 4 vertices v_1, v_2, v_n, v_3, which is K_4 except for $v_2 - v_n$ missing. Running the algorithm on this input will get stuck after the first step, since there is no eligible vertex, because the algorithm initialised the chord counter of v_3 to zero (assuming the outer face is a triangle with no chords) and now it drops to -1.
- The vertices v_1, v_2, v_n form a triangle but it is not facial. An example graph is a K_5 with vertices v_1, v_2, v_n, v_3, v_4 lacking the edge $v_3 - v_4$. Then, the graph is planar, v_1, v_2, v_n is a triangle but it is easy to see that it cannot be a face. Running the algorithm on this case, will lead to first removing v_n, then v_3 or v_4 in any order. It will run perfectly fine, just produce an order which is not canonical with respect to any plane embedding of the graph, because after removing v_n the remaining graph is either not internally triangulated or $v_1 - v_2$ is not on the outer cycle.

(c) I am going to use an extended version of the shift algorithm to both detect if a canonical ordering is valid and if so then generate an embedding (which is useful in part d)). Note that the exact details of where the vertices are placed (e.g. on a grid) are not important for us, as we are not looking for some kind of minimal embedding – we are only interested in finding an embedding. Nevertheless, it is convenient to use the well-studied shift algorithm. To be able to detect when a canonical ordering is valid, we have to inspect where and how the shift algorithm fails if the ordering is not canonical.

First of all, canonical orderings are only defined for plane graphs. Therefore, the question is slightly vague and not well-defined per se. I will interpret "...if r is a valid canonical ordering of G" as "if r is a valid canonical ordering of any plane embedding of G".

Now I provide pseudocode for the extended shift algorithm, not going into details about the actual coordinates and the shifting mechanism:

```
let v_1, ..., v_n be the canonical ordering
embed v_1, v_2 and v_3 in a triangle with v_1 and v_2 on the x axis
initialise outer face to out = [v_1, v_3, v_2]

for i in 4...n:
    let the neighbours of v_i that are already embedded be u = [u_1, u_2, ..., u_k]
    if size of u < 2:
        // canonical ordering is invalid as the graph at the next step would not be
            biconnected and hence internally triangulated
        return false
    if there exists a j <= n - k such that set(u) = set(out[j:(j+k)]): // this can be
        checked using a boolean vector
        embed v_i ''above'' all the vertices embedded so far, do the shifting (not
            necessary)
        update out by setting next(v_j) = v_i and next(v_i) = v_{j+k-1} in the linked
            list
    else:
        // this branch collects the many things that can go wrong. First of all, if v_i
             has neighbours that are in the interior of the outer cycle (i.e. embedded
            already but not on outer cycle), then planarity would be violated so the
            canonical ordering is invalid. Second, the most important point, if the
            neighbours don't form a contiguous segment on the outer cycle, the graph
            after adding the new vertex would not be internally triangulated. Note that
            it might be the case that by drawing v_i under the x-axis the internal
            triangulation property can be maintained, but then v_1-v_2 is no longer on
            the outer cycle.
        return false

return true
```

Note that instead of this, one could do just the combinatorial part of the algorithm checking if the canonical ordering is valid or not, and in a later stage do the actual embedding.

If we were to be formal, we would have to show the equivalence between the canonical embedding being valid and this algorithm. Clearly, one direction is easy, because if a canonical embedding is valid, then this algorithm returns true because it is just extending the shift algorithm with checks which hold if the assumptions are not violated. For the other direction,

showing the contrapositive, i.e. that if the algorithm returns true then the canonical ordering is valid for some embedding is easier. Assume the algorithm returns true. By construction, the embedding it creates in the meanwhile is planar (if a new vertex had neighbours from the interior the algorithm returns false immediately). It remains to argue that the canonical ordering provided forms a valid pair with this embedding. There are 3 properties to check:

- At each step($k \geq 3$) G_k is internally triangulated. This is enforced by the if-else statement requiring all the neighbours to form a continuous sequence on the outer cycle.
- $v_1 - v_2$ is on the outer cycle of each $G_k (k \geq 3)$. This follows the property of the shift algorithm that it places everything above the x axis while v_1 and v_2 are on the x axis.
- The new vertices always get a position on the new outer face. This follows from the property of the shift algorithm that it places the new node above, hence outside of the existing embedding.

(d) In this part we just have to build a pipeline consisting of three elements: guessing an outer face, constructing a candidate canonical ordering from it, checking if it really is a valid canonical ordering while also creating an embedding.

```
let n = |G|, m = |E|
if m != 3n-6:
    return FAIL // a graph with more than that many edges cannot be planar by
        Corollary 2.5. A graph with less than 3n-6 vertices cannot be maximal planar as
        shown during the lectures.
// in this case, the average degree is less than 6, so there is a vertex with degree at
    most 5
let v be a vertex with degree at most 5
// if there exists an embedding of G, then there are neighbours of v u and w such that
    v, u and w form a facial triangle (since all faces are triangles and v is part of at
    least one triangle)

for each pair u, w in pairs of distinct neighbours of v:
    with v_1=u, v_2=w, v_n=v run modified version of the algorithm in part a) which
    returns FAIL whenever it gets stuck, otherwise returns a candidate canonical
        ordering
    store result in canonical_candidate
    if canonical_candidate != FAIL:
        run a modified version of the algorithm in part c) which if finds that r is a
            valid canonical ordering, also constructs the embedding along the way and
            returns it, otherwise returns FAIL
        store result in variable embedding
        if embedding != FAIL:
            return embedding

return FAIL // no outer faces worked
```

This algorithm is clearly linear time, because there are at most $\binom{5}{2} = O(1)$ iterations of the for loop which runs two linear algorithms at most once per iteration.

3 Task 3

Summary of Geometric Spanners

The basic idea behind geometric spanners as introduced by L.P. Chew in [7] is that we would like to build a network on a set of nodes located on the plane, such that for any two nodes, the shortest path between them using the edges of the network is not much longer than the Euclidean distance of the two nodes. This would be an easy question without constraints, as we could take the complete graph matching the Euclidean distances exactly. However, the challenge is maintaining the aforementioned property as much as possible, while satisfying other properties, such as small number of edges or planarity. Geometric spanners have many applications, including designing telecommunication / road networks, approximating shortest paths in dynamic systems, and solving other combinatorial geometry problems, such as the closest pair problem (see e.g. [18]). The simplest example is perhaps designing the rail network of Switzerland such that the costs of building and maintaining it are not too large but people can travel between any two cities reasonably quickly. Interestingly, even river networks seem to showcase geometric spanner properties [20]. The area is very well-studied, with comprehensive books and surveys available ([17], [4]), however, there are several open problems still.

The general problem of spanners is defined formally as follows. Given a (positively) weighted graph $G = (V, E)$, a graph $G' = (V, E')$ with $E' \subseteq E$ is a t-spanner (often called a spanner with stretch factor t) of G, if $d_{G'}(x, y) \leq t \cdot d_G(x, y)$ for all $xy \in E$, where $d_{G'}(x, y)$ is distance on the shortest path between x and y in G' and $d_G(x, y)$ is so in G, which is turn fact equal to the length of the edge xy. The geometric spanner problem is a special case, where G is the complete graph derived by placing V in $\mathbb{R}^2$ and taking the Euclidean distances as edge weights (generalisations to non-complete graphs and higher dimensions exist [12]).

There are several objectives we might want to optimise alongside the stretch factor: number of edges, maximum degree, connectivity, fault tolerance, diameter, planarity, etc. (see [17] for a complete list). It is not hard to see that some of these objectives are contradictory (e.g. bounded degree and diameter), so the aim is usually to optimise a small subset of these properties at the same time. Unfortunately, the classical version of the problem, where for a given point set V, we have to find the graph G' having the best stretch factor with at most m edges is NP-hard as shown in [14]. Similar NP-hardness reductions exist for other subsets of objectives as well [6]. Hence, most of the research aims to construct spanner finding algorithms that result in a t-spanner for any point set V, satisfying/optimising the additional objectives. Such an algorithm is called a spanner, if it generates a t-spanner for any point set for a global constant t. In this work I am going to highlight some of the most important spanner constructions, investigating their pros and cons with respect to the several objectives.

Perhaps the simplest spanner is the so-called greedy geometric spanner, first introduced in [8]. The idea is to fix a target stretch factor t, and then go through the edges in non-decreasing order of their lengths, and always add the next edge xy to G' only if the current shortest path between x and y in G' does not yet satisfy $d_{G'}(x, y) \leq t \cdot d_G(x, y)$. Noticing the similarity to Kruskal's spanning tree algorithm, one can show that the graph generated is always a superset of a spanning tree [1]. By construction, the resulting graph is a t-spanner. This simple idea has several desirable properties. First, the resulting graph has bounded degree (depending on t), which also means that the number of edges is linear (which is the best achievable up to a constant factor) [8]. Also, its total edge weight is at most a constant times the total edge weight of the minimum spanning tree [11]. A greedy spanner can be constructed in $O(|V|^2 \log |V|)$ time [2].

An alternative, perhaps more geometrically motivated spanner is the family of θ-graphs. The idea is to split the space radially equally around each vertex v into k cones, and have at most one edge in of them going from v to a "nearest" vertex in that cone (for details on the construction see [17]). Intuitively, this construction makes sure that we can travel roughly in the direction of the target of our journey in each step. Opposing to the greedy geometric spanner, here we have bounded degree linear number of edges ($\leq k|V|$) by construction, but we have to show that it is indeed a spanner (i.e. has a global constant t such that it is a t-spanner for any point set). As explained in [17], θ-graphs are indeed spanners, with $t = (\cos 2\pi/k - \sin 2\pi/k)^{-1}$. While θ-graphs can have unbounded degree, there exist transformations of the graph that can achieve bounded degree for a quadratic tradeoff in the stretch factor [3]. θ-graphs can be constructed in $O(|V| \log |V|)$ time using a sweepline technique [17].

Underlying many of the constructions and efficient implementations of spanners is the so-called well-separated pair decomposition (WSPD) [19]. While the exact definition of WSPD is slightly technical (see e.g. [17]), the intuition is that we would like to represent the $O(n^2)$ edges of the complete graph induced by the point set using $O(n)$ "objects" (pairs of subsets of vertices) which capture the structure of the points. Alternatively, it can be viewed as partitioning the edges into $O(n)$ groups (possibly using the same edge multiple times). Using the WSPD technique, we can construct spanners with similar properties as the greedy spanner much faster (in $O(|V| \log |V|)$ time), and it also provides a way to construct spanner with linear number of edges and $O(\log |V|)$ diameter [19]. Apart from the WSPD technique, there are several other advanced data structures and algorithms used in many of the algorithms (e.g. range trees, θ-frames) [17].

Instead of enumerating further techniques and algorithms, I remark that there are very rich and sophisticated geometric observations that underpin most of the techniques and ideas used to optimise various objectives. These ideas are not only used in constructing algorithms but also in analysing them (e.g. bounding their stretch factor). Among various other properties (e.g.

leapfrog property, gap property), arguably the most important one is the empty region property [5] which says that there is a region of some shape around each edge of G' that does not contain any other point. Intuitively, if we forbid in our construction pairs of edges xy and yz such that x and z are nearby, we can guarantee that a lengthy zig-zag shortest path will not exists between any pairs of points. For example, the greedy spanner also exhibits the empty region property, but characterising the (shape and) size of the empty region is still an open problem [4].

One of the most important areas of geometric spanners is that of plane spanners, i.e. spanners which are planar graphs. The whole concept of geometric spanners was also first introduced via plane spanners in the influential paper by L.P. Chew [7]. The planarity constraint raises several simple-looking questions which are still unsolved. Perhaps the most central question, is what is the best stretch factor t such that there is a plane t-spanner for any point set. The best known lower bound is $\sqrt{2.005367532}$, which is achieved by letting the point set be the vertices of a regular 21-gon [16]. Note that a slightly worse lower bound is easy to show by placing 4 vertices on a square. The best known upper bound, 1.998 is achieved by the standard Delaunay triangulation [21]. Surprisingly, however, even determining the worst case stretch factor of the Delaunay triangulation is an open problem: it is somewhere between 1.5846 and 1.998. Also, the first constructions were Delaunay triangulations with other convex distance functions [7] which were easier to analyse than the standard Delaunay triangulation. Back to the original open problem, overall, t is somewhere between $\sqrt{2.005367532}$ and 1.998. Another main challenge in the area of plane spanners is minimising the maximum degree of a plane spanner. After a long sequence of papers decreasing the number, the best known spanner so far achieves a maximum degree of at most 4 on any point set, using a construction based on a Delaunay triangulation using the equilateral-triangle distance [13]. 2 is a fairly obvious lower bound, so the question remains whether 3 is achievable or not. If it is, it would mean that planarity imposes no additional constraints in this challenge, as in the unrestricted case the optimal maximum degree is also 3 [9].

As a final scenario, I will briefly present a version of geometric spanners which differs from the original problem. With the so-called Steiner-spanners, we are allowed to inserts additional "Steiner-points" as intermediate vertices, while for the stretch factor still taking into account only the original vertices. There are several flavours of this challenge, but one of the central questions is whether we can get less edges for a fixed stretch factor t by introducing Steiner-points. Indeed, it was just very recently shown that a quadratic improvement compared to the best results available for the standard setting is possible, in particular, for $t = 1 + \epsilon$, $n/\sqrt{\epsilon}$ edges are enough [15]. A related setting is the so-called dilation setting, where the Steiner-points are also taken into account for the stretch-factor calculation, and we also require planarity. The best achievable (by adding Steiner-points and edges) stretch factor is called the dilation of the point set. This leads to a famous open problem: what is the largest possible dilation a (finite) point set can have? It was shown in [10] that this value is somewhere between 1.00157 and 1.1247. An even simpler open problem is calculating the dilation of the regular pentagon [10].

In conclusion, geometric spanners is a very versatile, applicable and challenging area with several open problems, where solutions require ingenuity from geometry, algorithms, combinatorics, graph theory and complexity theory.

References

[1] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993.

[2] Prosenjit Bose, Paz Carmi, Mohammad Farshi, Anil Maheshwari, and Michiel Smid. Computing the greedy spanner in near-quadratic time. *Algorithmica*, 58(3):711–729, 2010.

[3] Prosenjit Bose, Joachim Gudmundsson, and Pat Morin. Ordered theta graphs. *Computational Geometry*, 28(1):11–18, 2004.

[4] Prosenjit Bose and Michiel Smid. On plane geometric spanners: A survey and open problems. *Computational Geometry*, 46(7):818–830, 2013.

[5] Jean Cardinal, Sébastien Collette, and Stefan Langerman. Empty region graphs. *Computational geometry*, 42(3):183–195, 2009.

[6] Paz Carmi and Lilach Chaitman-Yerushalmi. Minimum weight euclidean t-spanner is np-hard. *Journal of Discrete Algorithms*, 22:30–42, 2013.

[7] L Paul Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205–219, 1989.

[8] Gautam Das. *Approximation schemes in computational geometry*. The University of Wisconsin-Madison, 1990.

[9] Gautam Das and Paul J Heffernan. Constructing degree-3 spanners with other sparseness properties. *International Journal of Foundations of Computer Science*, 7(02):121–135, 1996.

[10] Annette Ebbers-Baumann, Ansgar Grüne, Marek Karpinski, Rolf Klein, Christian Knauer, and Andrzej Lingas. Embedding point sets into plane graphs of small dilation. In *International Symposium on Algorithms and Computation*, pages 5–16. Springer, 2005.

[11] Arnold Filtser and Shay Solomon. The greedy spanner is existentially optimal. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 9–17, 2016.

[12] Sariel Har-Peled, Piotr Indyk, and Anastasios Sidiropoulos. Euclidean spanners in high dimensions. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 804–809. SIAM, 2013.

[13] Iyad Kanj, Ljubomir Perković, and Duru Türkoğlu. Degree four plane spanners: Simpler and better. *arXiv preprint arXiv:1603.03818*, 2016.

[14] Rolf Klein and Martin Kutz. Computing geometric minimum-dilation graphs is np-hard. In *International Symposium on Graph Drawing*, pages 196–207. Springer, 2006.

[15] Hung Le and Shay Solomon. Truly optimal euclidean spanners. *SIAM Journal on Computing*, (0):FOCS19–135, 2022.

[16] Wolfgang Mulzer. Minimum dilation triangulations for the regular n-gon. *Master's thesis Freie Universität Berlin, Germany*, 2004.

[17] Giri Narasimhan and Michiel Smid. *Geometric spanner networks*. Cambridge University Press, 2007.

[18] Christian Schindelhauer, Klaus Volbert, and Martin Ziegler. Geometric spanners with applications in wireless networks. *Computational Geometry*, 36(3):197–214, 2007.

[19] Michiel HM Smid. The well-separated pair decomposition and its applications. *Handbook of approximation algorithms and metaheuristics*, 13, 2007.

[20] Hans-Henrik Stølum. River meandering as a self-organization process. *Science*, 271(5256):1710–1713, 1996.

[21] Ge Xia. Improved upper bound on the stretch factor of delaunay triangulations. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 264–273, 2011.