17 Combinatorics

17.7 Graph planarity

CATAM coursework for Part II of the Mathematical Tripos. Sections have been numbered as they appear in the manual.

1 Graph drawing

Question 1 Given a graph G with a cycle C containing vertices $v_1, v_2, ..., v_m$ (in order), we assign to each v_i the coordinates $(\sin \frac{2\pi i}{m}, \cos \frac{2\pi i}{m})$ so that the cycle occupies a regular m-gon. Let $V[G] \setminus \{v_1, ..., v_m\} = \{w_1, ..., w_n\}$ be the vertices not in the cycle, and write (x_i, y_i) for the coordinates of w_i. Define two matrices to record adjacency

$$\Delta_{ij} = \begin{cases} 1, & \text{if } \{w_i, w_j\} \in E[G] \\ 0, & \text{if } \{w_i, w_j\} \notin E[G] \end{cases}, \qquad \Omega_{ij} = \begin{cases} 1, & \text{if } \{w_i, v_j\} \in E[G] \\ 0, & \text{if } \{w_i, v_j\} \notin E[G] \end{cases}.$$

If C has a single bridge, Tutte's theorem provides a way to calculate the coordinates (x_i, y_i) such that the representation of G is planar:

$$x_i = \frac{\sum_{j=1}^n \Delta_{ij} x_j + \sum_{j=1}^m \Omega_{ij} \sin \frac{2\pi j}{m}}{d(w_i)}, \quad y_i = \frac{\sum_{j=1}^n \Delta_{ij} y_j + \sum_{j=1}^m \Omega_{ij} \cos \frac{2\pi j}{m}}{d(w_i)}.$$

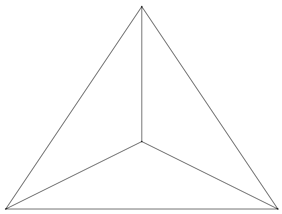
Defining three new matrices by

$$A_{ij} = \begin{cases} -\Delta_{ij}, & i \neq j \\ d(w_i), & i = j \end{cases}, \quad S_i = \sum_{j=1}^m \Omega_{ij} \sin \frac{2\pi j}{m}, \quad T_i = \sum_{j=1}^m \Omega_{ij} \cos \frac{2\pi j}{m},$$

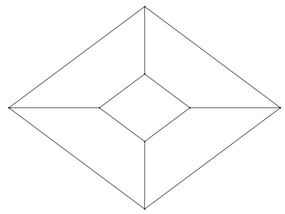
we can express Tutte's formula for the coordinates succintly as

$$A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = S, \quad A \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = T.$$

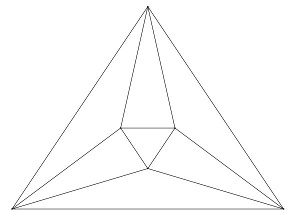
The coordinates of the vertices can now be obtained by calculating $A^{-1}S$ and $A^{-1}T$. Using the above method, we produce planar representations of the five platonic solids and the graph $K2 + P5$:



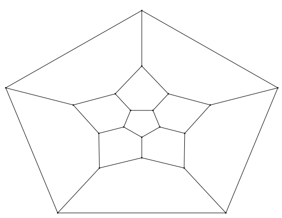
(a) The tetrahedron



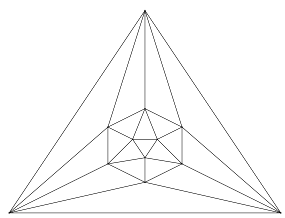
(b) The cube



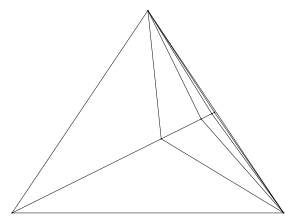
(c) The octahedron



(d) The icosahedron



(e) The dodecahedron



(f) $K2 + P5$

Figure 1: Six planar graphs plotted using Tutte's method.

While Tutte's formula works very well for graphs possessing a high number of symmetries, the disadvantage is apparent for graphs like $K2 + P5$ where the vertices are skewed in a particular direction leading to sub-optimal representations. If we understand the graph sufficiently well, this can be counteracted by applying appropriate weights to the vertices.

2 Bridges and components

Question 2 For a set $U \subset V[G]$, define the *neighbourhood* of U by $\Gamma(U) = \bigcup_{x \in U} \Gamma(x) \cup U$. Write $\Gamma^n(U) = \Gamma(\Gamma^{n-1}(U))$, where $\Gamma^0(U) = U$. For any vertex $v \in V[G]$, the sequence $\{v\}, \Gamma\{v\}, \Gamma^2\{v\}, ...$ is $\subset$-increasing and bounded above by $V[G]$, hence stabilises in finitely many steps and the fixed point is a connected component of G containing v. Repeating this process, we can find all the connected components of the graph.

Finding the bridges of a cycle C now is straightforward by finding the connected components of $G \setminus V[C]$, and the vertices of attachment can be found by examining the edges connecting C to these components. Chords need to be found separately by examining the vertices in the cycle pairwise.

3 Interleaving

Question 3 Suppose G is a planar graph, then every subgraph of G is planar. If C is a cycle in G with ℓ bridges $B_1, ..., B_\ell$, choose a planar representation of G. In this representation, the cycle C is a simple closed curve, hence[1] splits the plane into two regions– call them the *inside* and the *outside*. Since a bridge is connected, it must entirely lie in the (topological) closure of one of these regions. Now a bridge contains a path between its vertices of attachment which does not intersect the cycle except at its end points. Moreover, distinct bridges can only meet at their vertices of attachment. If a, b, c, d are distinct vertices appearing on the cycle in that order such that a, c are vertices of attachment of the bridge B_i and b, d are vertices of attachment of the bridge B_j (not equal to B_i), then B_i contains an $a-c$ path while B_j contains a $b-d$ path; and the only way these paths don't intersect is if they lie in different regions. It follows that B_i and B_j must themselves lie in different regions. On the other hand, suppose the vertices of attachment of both B_i and B_j are precisely a, b, c (all distinct). In particular, neither of the bridges is a chord so we can pick a vertex $x \in V[B_i] \setminus V[C]$. Without loss of generality B_i is drawn on the inside of the plane, so that the $x-a, x-b, x-c$ paths contained in B_i (along with the cycle C) divide the plane into four disjoint regions (the outside, and the three subdivisions of the inside). The drawing of B_j must lie in one of the regions, and moreover that region must have all three of the vertices of attachment on its boundary. The only such region is the outside. We have thus shown that any two interleaving graphs lie in different regions of the plane– this induces a bipartition of the interleave graph H of G.

Conversely, suppose each of the subgraphs G_i with edges $E(C) \cup B_i$ ($1 \leq i \leq \ell$) is planar and the interleave graph H is bipartite. Choose a bipartition $V[H] = A \cup B$ of H. Since G_i is a cycle with a single bridge, Tutte's theorem allows for a planar drawing where C occupies a regular polygon and every other vertex of B_i is placed in the centroid of its neighbours. In particular, the entire drawing of B_i lies in the convex polygon determined by its vertices of attachment. If $i, j \in A$ (likewise B) are distinct, the bridges B_i and B_j do not interleave, and hence the convex hulls of their vertices of attachments are disjoint. It follows that the subgraph G_A with edges $E(C) \cup \bigcup_{i \in A} B_i$ is planar, and has a drawing where C occupies a regular polygon and every other vertex

[1] by the Jordan curve theorem

is at the centroid of its neighbours. A similar drawing is produced for G_B, the subgraph with edges $E(C) \cup \bigcup_{i \in B} B_i$. Now observing that planar graphs are in fact graphs on a sphere (by considering steriographic projection, for instance), we produce spherical drawings of G_A and G_B such that C occupies the equator. These drawings can be put together such that G_A and G_B lie in opposite hemispheres (and they agree on C)– resuting in a spherical drawing of G. By projecting steriographically on the plane, conclude that G is planar.

4 The core of a graph

Question 6 Suppose G is a graph of minimum degree at least two. Then pick a vertex x_1– this has two distinct neighbours x_0 and x_2, giving a path $x_0x_1x_2$ in G. Having constructed a path $x_0x_1...x_n$ (all vertices distinct, $n \geq 2$), observe that x_n has a neighbour $y \neq x_{n-1}$. If $y \in \{x_0, ..., x_{n-2}\}$– say $y = x_i$, we have found a cycle $x_ix_{i+1}...x_n$ in G. Otherwise, we have a longer path $x_0x_1...x_ny$ and can repeat the procedure. But the graph is finite, hence the process terminates and we find a cycle in finite time.

Likewise, suppose G has minimum degree at least three. From above, we have a path $x_0x_1x_2$. In fact, x_2 must have a neighbour $x_3 \neq x_0, x_1$, giving a path $x_0x_1x_2x_3$. Having constructed a path $x_0x_1...x_n$ ($n \geq 3$), observe that x_n has two distinct neighbours y, z, neither equal to x_{n-1}. If both lie in $\{x_0, ..., x_{n-2}\}$– say $y = x_i$, $z = x_j$ for $i < j$, we have found a cycle $x_ix_{i+1}...x_n$ in G and this has a chord x_nx_j. Otherwise, we have found a longer path $x_0x_1...x_ny$, and can repeat the procedure. This algorithm is again guaranteed to terminate by the finiteness of G.

5 A planarity algorithm

Question 7 Since the operations involved in finding the core decrease the number of vertices, the process terminates and we can find the core G^* of G in finite time. It is clear that a graph is planar if and only if its core is; in particular, if G^* is empty then G is planar. If G^* is non-empty, it has minimum degree 3 hence contains a cycle C with a chord e. We have already exhibited terminating algorithms for finding such a cycle with a chord, and constructing its interleave graph H. Suppose the bridges of C are $B_1, ..., B_\ell$ where $B_1 = \{e\}$. Note that the subgraph with edges $E(C) \cup B_1$ is planar since it is a cycle with a chord. From the discussion in Queston 3, if H is not bipartite then G is not planar– and we have a terminating algorithm to check if a graph is bipartite (in fact, to produce a bipartition). If H is bipartite, then G^* is planar if and only if each of the subgraphs with edges $E(C) \cup B_i$ $(2 \leq i \leq \ell)$ is planar. Lastly, observe that C is a cycle in $G^* - e$ with bridges $B_2, ..., B_\ell$, and the corresponding interleave graph is a subgraph of H. Thus H (and hence its subgraphs) being bipartite, G^* is planar if and only if $G^* - e$ is. Since at each step of the recursion we either terminate by deciding the planarity of G or reduce the problem to checking the planarity of a graph with strictly fewer edges, the algorithm must terminate.

Question 8 We implement the above algorithm and test it on the following graphs:

(i) $K2 + P5$, which we know is planar.

(ii) $K3, 3$, which from Kuratowski's theorem is non-planar.

(iii) $K5$, which from Kuratowski's theorem is non-planar.

(iv) The dodecahedron P_{12}, which we know is planar.

(v) P_{12} with two edges added– since the every face of the dodecahedron is a triangle, it is maximal planar hence adding any edges makes it non-planar.

In each case, the algorithm returns the expected output.

Question 9 If G is a maximal planar graph on n vertices, observe that each face of G must be a triangle– else we could subdivide a non-triangular face to get a planar graph which has G as a proper subgraph. Then each face is surrounded by three edges and each edge by two faces, so if a planar representation of G has F faces and E edges, we can write

$$\#\{(f, e) \mid f \text{ a face, } e \text{ an edge surrounding } f\} = 3F = 2E.$$

Combined with Euler's formula $n - E + F = 2$, we have $E = 3n - 6$.

Starting with the empty graph on n vertices and adding each of the $\binom{n}{2}$ edges in random order as long as planarity is preserved, the graph we end up with is indeed maximal planar, hence has $3n - 6$ edges.

Proposition. *Suppose G is a maximal planar graph on $n \geq 4$ vertices. Then G is 3-connected and contains a cycle with exactly one bridge.*

Proof. Let G be a maximal planar graph on $n \geq 4$ vertices. Pick a planar representation of G. Call an edge e *contractible* if the only triangles containing e are faces (i.e. its end points have at most two common neighbours). Write G/e for the graph formed by contracting the edge e, i.e. identifying its end points. This is still planar by Wagner's criterion since contracting an edge cannot create new minors. Moreover, we lose exactly 3 edges in the contraction hence G/e contains $n-1$ vertices and $3n-9$ edges. It follows that G/e is maximal planar.

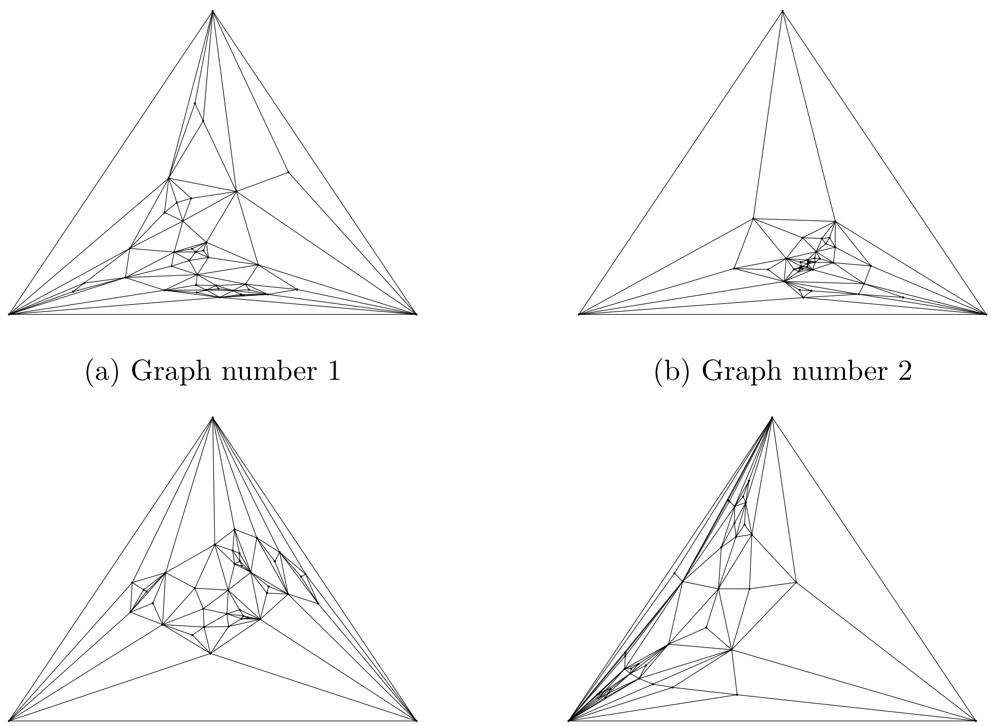
We show by induction that G contains at least n contractible edges: clear if $n = 4$. If $n > 4$, take a planar representation of G and choose a triangle in G that is not a face. The subgraphs G_1 and G_2 contained (topologically) inside and outside the triangle are both maximal planar and together have $|G_1| + |G_2| = n + 3$ vertices, giving $n+3$ possible candidates for contractible edges. Then in G, all of these continue to remain contractible except for (possibly) the edges of the triangle, which can lose the property. Hence G contains at least n contractible edges.

Now suppose there are maximal planar graphs on 4 or more vertices which are not 3-connected. Choose G to be such a graph with the minimal number of vertices, then there are $u, v \in V(G)$ such that $G - \{u, v\}$ is disconnected. Let the components be G_1 and G_2. Note that G has > 4 vertices since the only maximal planar graph on 4 vertices is K_4. From the above discussion, we can pick an $e \in E(G)$ which is contractible, then without loss of generality this edge must lie in G_1. We observe that G/e is maximal planar with $n - 1 \geq 4$ vertices, and removing (the possibly modified versions of) $\{u, v\}$ still disconnects G/e. But G was chosen to be minimal, a contradiction. Hence every maximal planar graph on 4 or more vertices is 3-connected.

Choose a planar representation of G and let C be the 3-cycle bounding a face. C has at least one bridge, and all the bridges lie in the (topological) complement of the face bounded by C. If there were multiple such bridges, we could add edges connecting their bodies while preserving the planarity of G; this contradicts the maximality of G. Hence the boundary of a face in G has exactly one bridge. $\square$

We generate twenty random maximal planar graphs on 40 vertices. Now any edge $x - y$ of such a graph bounds some face, hence is a part of some 3-cycle with exactly one bridge. We can iterate over all the other vertices $z \in$

$V[G] \setminus \{x, y\}$ till we find such a cycle xyz, which we use as the outermost face in our plot. The proposition then guarantees that Tutte's plotting algorithm works, so we use it to exhibit planar representations of a selection of the graphs generated in fig. 2. Appendix A contains edge-lists for the graphs shown.



(a) Graph number 1

(b) Graph number 2

(c) Graph number 3

(d) Graph number 4

Figure 2: Four of the randomly generated maximal planar graphs.

Question 10 Suppose we want to check if a graph G with n edges is planar. Then it contains at most $O(n)$ vertices. To find the core of G, we iterate over all the vertices and remove the ones of degree ≤ 3. Since computing the degree of any vertex (i.e. the length of the list containing its neighbourhood) takes constant time, finding the core has complexity $O(n)$. In the worst case, G is its own core. Now we simply start with a vertex and extend the path one step at a time till it closes into a cycle C with a chord, this taking another $O(n)$ steps. To find the component of a particular vertex in $G - C$, we pick a vertex and keep adding neighbours of vertices found so far. This takes $O(n^2)$ steps, and we do this for $O(n)$ vertices in the worst case hence finding the bridges has complexity $O(n^3)$.

In the worst case, the cycle we find has length $O(n)$ and there are $O(n)$ components. Then checking if each pair of components interleaves takes $O(n)$ steps (traversing once through the cycle, keeping track of possible interleaves seen so far), hence the interleaf graph takes $O(n^3)$ steps to construct and has size $O(n)$. Checking if the interleave graph is bipartite then has complexity $O(n^2)$.

Hence a single iteration of the algorithm has complexity $O(n^3)$. Each iteration, we remove (in the worst case) one edge from G hence the entire algorithm takes $O(n^4)$ steps to check for planarity.

Appendix A. Random maximal planar graphs

We present the output produced after running the algorithm to randomly generate maximal planar graphs on 40 vertices (labelled `0` to `39`). One evidence that the algorithm works properly is to observe that each graph has $114 = 3 \times 40 - 6$ edges, arranged in a 6×19 grid. Each grid should be read row-by-row, left to right and top to bottom. Across twenty such graphs, the mean number of edges added before the first violation of planarity is 42.55.

1.

1-31	26-30	24-33	18-26	34-37	9-17
4-39	24-26	15-23	8-39	30-39	3-15
9-10	14-24	5-21	23-36	1-26	7-21
1-35	10-38	11-37	18-30	12-20	3-17
23-29	14-26	9-12	23-38	4-34	24-30
18-23	11-39	3-10	22-23	21-25	3-19
31-35	23-31	5-25	27-31	1-21	4-11
14-21	14-30	1- 5	16-21	19-38	11-34
6-31	4-13	23-26	19-22	36-38	10-19
17-34	10-28	1-25	9-30	15-27	4- 6
13-39	14-16	18-32	2- 8	17-27	11-20
30-33	9-32	6- 8	6-17	15-29	3- 9
4- 8	6-35	4-17	12-30	11-13	14-39
28-38	10-32	9-34	15-22	32-38	15-31
26-31	21-26	18-36	20-39	9-18	0-15
17-31	19-28	2-21	6-21	26-33	18-38
22-29	1- 6	9-20	20-34	8-14	3-27

2- 7	19-23	12-39	0-19	2-16	6- 7
15-19	6-25	0- 3	8-16	4-37	7- 8

First exception encountered afer 43 additions.

2.

15-21	16-20	16-29	7-35	20-29	0- 5
34-37	22-35	23-34	3-31	0-14	4-39
13-26	6-20	18-28	20-22	18-37	8-11
3-33	13-19	13-28	16-24	7-21	24-37
13-37	3-17	14-26	23-38	30-34	2-36
12-13	5- 7	0- 2	3-28	3-37	13-14
17-39	19-36	30-36	14-21	31-37	3-30
1- 5	2-13	8-29	24-25	33-37	1-32
6-22	12-26	8-13	1- 7	13-18	7-11
9-30	26-27	20-32	7-32	17-18	10-12
10-21	9-23	28-36	11-29	20-25	12-21
29-37	6-35	4-17	0-13	1-20	11-22
9-25	2-19	5-36	9-34	3-39	24-34
12-14	8-10	34-38	0-15	8-37	9-36
13-27	3- 4	33-39	13-36	22-29	8-12
2- 5	8-21	9-20	3-34	17-33	9-38
24-29	16-25	7-22	24-38	1-36	6-32
31-34	5-15	14-27	3-36	32-35	7-15
18-33	20-36	17-28	9-24	0-21	11-21

First exception encountered afer 40 additions.

3.

26-30	25-32	18-26	20-29	22-26	8- 9
12-25	14-22	0-14	28-30	3-31	15-23
32-39	7-28	7-37	12-18	5-21	17-23
10-29	1-26	7-21	20-33	12-29	12-38
9-21	3-35	19-34	10-31	16-17	2-27
12-13	3-19	31-35	4-18	21-34	10-15
1-12	8-27	30-36	22-25	4-11	12-24
3-21	14-30	4-38	8-20	10-17	7- 9
6-22	3- 5	20-21	29-33	4-13	23-35
10-19	13-18	19-31	1-25	16-23	0-36
26-36	29-35	14-25	4-33	10-12	2- 8
1-18	27-37	8-33	10-30	11-29	2-26
11-38	15-29	8-26	10-23	0-22	10-32
2-28	7-27	5-20	22-36	10-16	5-29
19-28	10-25	17-31	11-24	8-37	11-33
26-33	21-28	1- 6	19-21	8-21	3-34
19-30	24-38	13-38	15-35	2- 7	12-39
14-36	17-35	26-28	6-25	25-30	25-39
18-33	12-32	11-12	5-35	7- 8	1-22

```
First exception encountered afer 39 additions.
```

4.

32-37	26-30	17-21	11-14	5-28	9-35
27-34	1-15	6-20	1-33	14-15	22-28
0- 7	21-39	11-16	9-28	27-36	15-16
8-32	10-29	24-28	18-21	0-37	2-34
16-33	38-39	12-20	21-32	4-25	0-30
0-39	12-13	21-34	4-36	1-12	13-14
11-32	0-32	20-37	31-37	12-33	23-33
1-14	17-32	0-25	2-22	16-21	0-34
6-31	21-38	14-32	2- 6	1-16	28-34
30-31	0-27	0-36	2-33	25-35	3- 7
20-23	6-33	20-32	18-32	17-18	27-28
19-33	11-29	0-38	2-35	6-26	6-35
7-34	21-33	0- 4	4-35	22-34	10-14
5-27	2-19	5-36	10-32	0-31	2-28
6-19	24-34	12-14	7-27	12-23	6-37
4-28	34-38	1-13	11-15	2-21	6-30
0- 8	8-21	11-17	3-34	0-26	0-35
6-23	4- 5	26-35	10-11	28-35	22-24
0- 3	12-32	14-29	2- 9	11-21	0-21

```
First exception encountered afer 45 additions.
```

Appendix B: Programs

graphs.py

```python
import math
import numpy as np
import matplotlib.pyplot as plt

def edge(x,y):
    if x < y:
        return (x,y)
    else:
        return (y,x)


class Graph:
    def __init__(self, adj):
        self.adj = adj # dict{vtx : ngb set}

    def vts(self): # list[vtx]
        return self.adj.keys()

    def eds(self): # list[(v1,v2)]
        eds = set()
        for x in self.vts():
```

```
            for y in self.adj[x]:
                eds.add(edge(x,y))
        return eds

    def degree(self, x):
        return len(self.adj[x])

    def addEdge(self, e):
        for x in e:
            if x not in self.vts():
                self.adj[x] = set()
        self.adj[e[0]].add(e[1])
        self.adj[e[1]].add(e[0])
        return ()

    def rmEdge(self, e):
        (x,y) = e
        if edge(x,y) not in self.eds():
            print(e)
            print(self.adj)
        self.adj[x].remove(y)
        self.adj[y].remove(x)
        return ()

    def rmVert(self,x):
        self.adj.pop(x)
        for y in self.vts():
            self.adj[y].discard(x)
        return ()

    def plotWith(self, cycle, name):
        xCoord = {}
        yCoord = {}
        body = [x for x in self.vts() if x not in cycle]
        # coordinates of cycle
        for i in range(len(cycle)):
            xCoord[cycle[i]] = math.sin(2 * math.pi * i / len(cycle))
            yCoord[cycle[i]] = math.cos(2 * math.pi * i / len(cycle))
        # matrix of coefficients
        mat = []
        for x in body:
            coeffs = []
            for y in body:
                if y == x:
                    coeffs.append(len(self.adj[x]))
                elif y in self.adj[x]:
                    coeffs.append(-1)
                else:
                    coeffs.append(0)
            mat.append(coeffs)
        mat = np.matrix(mat)
        # constant terms
        xConst = np.matrix([ sum([xCoord[y]
                                  for y in cycle
                                  if y in self.adj[x]])
                             for x in body ])
        yConst = np.matrix([ sum([yCoord[y]
```

```
                                  for y in cycle
                                  if y in self.adj[x]])
                             for x in body ])
        # coordinate computation
        xVals = np.matmul(mat.I, xConst.T)
        yVals = np.matmul(mat.I, yConst.T)
        for i in range(len(body)):
            xCoord[body[i]] = xVals.item(i,0)
            yCoord[body[i]] = yVals.item(i,0)
        # plot
        plt.clf()
        for e in self.eds():
            xs = np.array([xCoord[v] for v in e])
            ys = np.array([yCoord[v] for v in e])
            plt.plot(xs, ys, c='black', lw='0.5')
        xs = np.array([xCoord[v] for v in self.vts()])
        ys = np.array([yCoord[v] for v in self.vts()])
        plt.plot(xs, ys, marker='o', ms=0.7, c='black', ls='')
        plt.axis('off')
        plt.savefig('../output/' + name + '.pdf',
                    bbox_inches='tight')
        return ()

    def components(self): # list[set{vts}]
        vertSet = set(self.vts())
        components = []
        def expand(c):
            d = c.copy()
            for x in c:
                d.update(self.adj[x])
            return d
        while vertSet:
            x = min(vertSet)
            xcomp = {x}
            while len(expand(xcomp)) != len(xcomp):
                xcomp = expand(xcomp)
            vertSet.difference_update(xcomp)
            components.append(xcomp)
        return components

    def isBipartite(self):
        def connectedBipartite(comp):
            red = {min(comp)}
            blue = set()
            def expandColouring():
                for r in red:
                    blue.update(self.adj[r])
                for b in blue:
                    red.update(self.adj[b])
                if red.intersection(blue):
                    return False
                else:
                    return True
            xs = comp.copy()
            while xs:
                if expandColouring():
                    xs.difference_update(red)
```

```
                    xs.difference_update(blue)
                else:
                    return False
            return expandColouring()
        for c in self.components():
            if not connectedBipartite(c):
                return False
        return True


def fromString(edges):
    xs = edges.splitlines()
    adj = {}
    for x in xs:
        x = x.strip().replace('  ',' ')
        ys = list(map(int, x.split(" ")))
        for y in ys:
            if y not in adj:
                adj[y]=set()
        adj[ys[0]].add(ys[1])
        adj[ys[1]].add(ys[0])
    return Graph(adj)
```

bridges.py

```
import graphs as g
import copy

def bridges(graph, cycle):
    bridges = []
    # find chords
    for i in range(len(cycle)):
        x = cycle[i]
        x1 = cycle[ (i - 1) % len(cycle)]
        x2 = cycle[ (i + 1) % len(cycle)]
        for y in graph.adj[x]:
            if y in cycle and y not in {x1, x2} and y < x:
                bridges.append({x,y})
    # find bridges
    tempGraph = copy.deepcopy(graph)
    for x in cycle:
        tempGraph.rmVert(x)
    for c in tempGraph.components():
        d = c.copy()
        for x in c:
            for y in cycle:
                if y in graph.adj[x]:
                    d.add(y)
        bridges.append(d)
    return bridges

def interleave(cycle, bridges):
    # vertices of attachment in cycle order
    def attachVerts(br):
        return [x for x in cycle if x in br]
    # check if bridges interleave
```

```
    def isInterleaf(b1, b2):
        xs = attachVerts(b1)
        ys = attachVerts(b2)
        if len(xs) == 3 and xs == ys:
            return True
        else:
            changes = 0
            cur = ""
            for i in cycle:
                if cur == "":
                    if i in xs:
                        cur += 'x'
                    if i in ys:
                        cur += 'y'
                elif cur == "x":
                    if i in ys:
                        changes += 1
                        cur = ""
                        if i in xs:
                            cur += 'x'
                        cur += 'y'
                elif cur == "y":
                    if i in xs:
                        changes += 1
                        cur = ""
                        cur += 'x'
                        if i in ys:
                            cur += 'y'
                else:
                    if (i in xs) or (i in ys):
                        changes += 1
                        cur = ""
                        if i in xs:
                            cur += 'x'
                        if i in ys:
                            cur += 'y'
            return (changes >= 3)

    # create graph
    adj = {}
    for i in range(len(bridges)):
        adj[i] = set()
        for j in range(len(bridges)):
            if j != i and isInterleaf(bridges[i], bridges[j]):
                adj[i].add(j)
    return g.Graph(adj)
```

core.py

```
import graphs as g
import bridges as b
import copy

def core(graph):
    gr = copy.deepcopy(graph)
    i = 0
```

```python
    while True:
        x = list(gr.vts())[i]
        if gr.degree(x) < 2:
            gr.rmVert(x)
            i = 0
        elif gr.degree(x) == 2:
            y = list(gr.adj[x])[0]
            z = list(gr.adj[x])[1]
            gr.rmVert(x)
            gr.adj[y].add(z)
            gr.adj[z].add(y)
            i = 0
        else:
            i = i + 1
        if i == len(gr.vts()):
            break
    return gr


def findCycle(graph):
    # use only if it is certain that minimum degree is 3
    cycle = [min(graph.vts())]
    while True:
        x = cycle[-1]
        ys = copy.deepcopy(graph.adj[x])
        if len(cycle) < 4:
            y = min([y for y in ys if y not in cycle])
            cycle.append(y)
        else:
            ys.remove(cycle[-2])
            if len([y for y in ys if y in cycle]) < 2:
                y = min([y for y in ys if y not in cycle])
                cycle.append(y)
            else:
                y = [y for y in ys if y in cycle][0]
                z = [y for y in ys if y in cycle][1]
                while cycle[0] not in {y,z}:
                    cycle.pop(0)
                if cycle[0] == y:
                    chord = g.edge(x,z)
                else:
                    chord = g.edge(x,y)
                return (cycle, chord)


def isPlanar(graph):
    gStar = core(graph)
    if len(gStar.vts()) == 0:
        return True
    else:
        (cycle, chord) = findCycle(gStar)
        bridges = b.bridges(graph, cycle)
        interleave = b.interleave(cycle, bridges)
        if interleave.isBipartite():
            gStar.rmEdge(chord)
            return isPlanar(gStar)
        else:
```

```
            return False
```

main.py

```
import graphs as g
import core as c
import bridges as b
import random
import sys

# plot Platonic solids
platonic = {}
for n in [4, 6, 8, 12, 20]:
    file = open("../data/II-17-7-Platonic_" + str(n) + ".txt")
    platonic[n] = g.fromString(file.read())
    if n in [4, 8, 20]:
        platonic[n].plotWith([1,2,3],"Q1-platonic-" + str(n))
    elif n == 6:
        platonic[n].plotWith([1,2,3,4],"Q1-platonic-" + str(n))
    else:
        platonic[n].plotWith([1,2,3,4,5],"Q1-platonic-" + str(n))

# plot k2+p5
k2PlusP5 = g.fromString("1 2\n3 4\n4 5\n5 6\n6 7")
for i in [1,2]:
    for j in [3, 4, 5, 6, 7]:
        k2PlusP5.addEdge((i,j))
k2PlusP5.plotWith([1,2,3],"Q1-k2-plus-p5")

# two standard non-planar graphs
k33 = g.fromString("1 2\n1 4\n1 6\n3 2\n3 4\n3 6\n5 2\n5 4\n5 6")
k5 = g.fromString("1 2\n1 3\n1 4\n1 5\n2 3\n2 4\n2 5\n3 4\n3 5\n4 5")

print("k2+p5 is planar:" + str(c.isPlanar(k2PlusP5)))
print("k3,3 is planar:" + str(c.isPlanar(k33)))
print("k5 is planar:" + str(c.isPlanar(k5)))
print("Dodecahedron is planar:" + str(c.isPlanar(platonic[20])))

platonic[20].addEdge((1, 10))
platonic[20].addEdge((2, 10))
print("Dodecahedron+(1,10)+(2,10) is planar:" + str(c.isPlanar(platonic[20])))

# random maximal planar graphs
def randomMaximal(n):
    allEdges = []
    for i in range(n):
        for j in range(i+1, n):
            allEdges.append((i,j))
    random.shuffle(allEdges)
    graph = g.Graph({})
    rejectionsAt = []
    for i in range(len(allEdges)):
        sys.stdout.write("\rTrying edge number % i" % (i+1))
        sys.stdout.flush()
        x = allEdges[i]
        graph.addEdge(x)
```

```
        if not c.isPlanar(graph):
            rejectionsAt.append(i)
            graph.rmEdge(x)
    rejectionsAt.append(0)
    return (graph, rejectionsAt[0])

for i in range(20):
    print("\nGenerating graph number " + str(i+1) + "...")
    (graph, n) = randomMaximal(40)
    print("\nFirst violation encountered after " + str(n) + " additions.\n")

    edges = list(graph.eds())

    # clear contents before proceeding
    f = open("../output/Q9-maximal-"+str(i+1)+".txt", "w")
    f.close()

    # generate table
    f = open("../output/Q9-maximal-"+str(i+1)+".txt", "a")
    def monostr(x):
        if x < 10:
            return " "+str(x)
        else:
            return str(x)
    for j in range(19):
        f.write("   ".join([monostr(x)+"-"+monostr(y)
                          for (x,y) in edges[6*j:6*(j+1)]]))
        f.write("\n")
    f.write("\nFirst exception encountered afer " + str(n) +" additions.")
    f.close()

    (u,v) = edges[0]
    for w in range(40):
        if (g.edge(u,w) in edges
            and g.edge(v,w) in edges
            and len(b.bridges(graph, [u,v,w])) == 1):
            graph.plotWith([u,v,w],"Q9-random-40"+str(i+1))
            print("Plotted. \n")
            break
```