Mitacs Industrial Math Summer School
Maple Graph Theory Package
Final Report

Nikolas Karalis
Raul Aliaga
Andrew Arnold
Wenqian Wu

August 24, 2007

1 Introduction

The four of us were given the project of improving the Graph Theory Package for Maple 12. There were graduate students who had been working on this for a while. From them, we have understood the problems to be solved. At the same time, each of us was guided to start working on the following four specific problems during the month-long stay at CECM.

- Graph Isomorphism
- Graph Colouring
- Fullerenes
- Planar Drawing

Although there had been many collaborations between some of us and with the graduate students, our work remained relatively independent. Halfway through the project, each of us has essentially picked one topic from the four to focus on. We have Nikolas working on Graph Isomorphism, Raul on Graph Colouring, Andrew on Fullerenes and Wendy on Planar Drawing. Therefore at the end, we figured it was more appropriate for each of us to prepare an individual subreport. This final report is the product of the four subreports put together. Thus don't be too suprised when you notice a sudden change of writing style from one section to another. In a way, we are presenting our project and working experience from four perspectives.

Contents

2 Graph Isomorphism

2.1 Introduction

During the past month, I worked at the CECM (Centre for Experimental and Constructive Mathematics) on the improvement of the Graph Theory package for the MAPLE software package. I tried a few different things (like implementing a Depth-first search and Breadth-first search algorithm, as far as an algorithm solving the Knight's tour problem), before ending up to the Graph Isomorphism problem, on which I worked till the end of the Summer School.

2.2 Theory and Problem Description

In graph theory, a graph isomorphism is a bijection (a one-to-one and onto mapping) between the vertices of two graphs G and H $f : V(G) \to V(H)$ with the property that any two vertices u and v from G are adjacent if and only if f(u) and f(v) are adjacent in H.
The graph isomorphism problem is whether two graphs are isomorphic (there exists a graph isomorphism between them) or not.
This problem is characterized as NP, but it is not know yet if it belongs in the polynomial time or the NP-complete class.
What we wanted for Maple, is to have a fast function of determining whether the 2 graphs are isomorphic or not and if they are not, to know which is their isomorphism.
Before proceeding with my experiments and results, it is useful to include some definitions.
The **Adjacency Matrix** of a finite directed or undirected graph G on n vertices is the n x n matrix where A_{ij} is the number of edges from vertex i to vertex j. In the cases we are interested, the graphs have no loops, so all the A_{ii} are 0.
The **Degree** of a vertex is the number of edges incident to the vertex.
The **Degree Matrix** is a diagonal matrix, where the entry $D_{i,i}$ is the degree of the vertex i.
The **Laplacian Matrix** is given by D-A and the **Absolute Laplacian Matrix** is given by D+A.

2.3 Experiments

For all the following, we suppose to have two graphs G and H, and try to find an isomorphism between these two. The first steps towards the isomorphic test, consist of implementing the backtracking algorithm, which is slow, but it is sure that if there exists an isomorphism, it will be found. For doing that, we need to have a partition of the possible vertices of graph H that

correspond to each of the vertices of the graph G. We do that based on the degree of each vertex.
So, i implemented the partition method based on the degree of each vertex, which is used as an input to the backtracking method that Wendy implemented. You can see the results of the partition method in Table 1.
In order to check all the other algorithms, we needed to have a big list of graphs, so i wrote an algorithm for creating all the graphs (with respect to the degree sequence) of a given number of vertices.
One trivial and fast, but really usefull isomorphism test, that is the base for almost every other test, is the Degree Sequence of a graph. So, in our method, we included the check for the sorted degree sequence of the graphs.
After doing these, i had to find a way to make the test faster. In order to do that, i began experimenting with different graph invariants.
You can see a comparison of these in the next tables.
One of the most common graph invariant, is the eigenvalues of the Adjacency Matrix of a graph. However, this is not a very good invariant, since many non isomorphic graphs have the same eigenvalues.
A better method, is the eigenvalues of the Laplacian Matrix of a graph (furthemore, the second smallest eigenvalue is the algebraic connectivity of the graph). Another better matrix to use is the Absolute Laplacian Matrix. When saying better, we mean that less graphs have the same eigenvalues.
Since we need to do all these computations fast, instead of computing the eigenvalues of these matrices, we use the power of modular arithmetic and we compute the characteristic polynomial of the matrices mod p, where p is a very big (random) prime number.
All the previous can be found in published papers on the topic.
The following is a result that i came up with, and it is not yet published (as far as i know).
We define a new matrix D', which is a diagonal matrix, where the entry $D'_{i,i}$ is the square of the degree of the vertex i.

$$D'_{i,i} = deg^2(i)$$

We then define the matrix E = D' + A , where A is the Adjacency Matrix of the graph.
This matrix is a much better graph invariant, since it is a perfect isomorphism test for graphs up to 9 vertices, and there is only at 10 vertices and 14 edges where this test shows it's weaknesses.
There are only 4 pairs of graphs, that falsely pass this test at 10 vertices and 14 edges. So the next step was to find a new matrix, that works for these 8 graphs. After many experiments, i found a new matrix, that works for 6 of these graphs. The new matrix is defined as :

$$M = L \bullet L' \quad , \quad L = A + D \quad , \quad L' = 2A + D$$

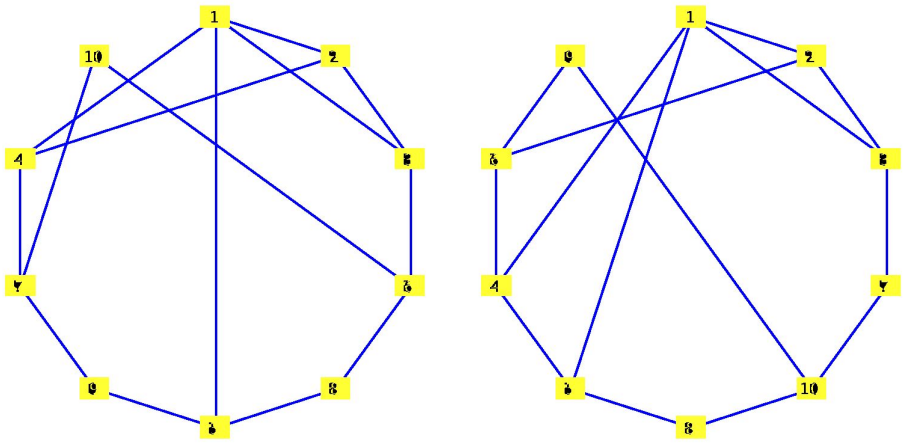


Figure 1: 10 Vertices - 14 Edges

We can also define the matrix as :

$$M = L \bullet L' \quad , \quad L = A + D' \quad , \quad L' = 2A + D'$$

but this doesn't improve the results, at least in the range of my test.
However, the fact that these tests are graph invariants remains to be proven. I have run extended tests which all show that these tests are *probably* graph invariants, but i haven't proved it.
The only remaining pair of graphs that has the same characteristic polynomial for this matrix, is displayed in Figure 1.
While trying to find the properties of these two graphs, I found that the M matrices, as defined above, for these 2 graphs are *similar*.
When two matrices are similar, they have the same rank, the same determinant, the same trace, the same eigenvalues, the same characteristic polynomial and the same minimal polynomial. So we can say that these 2 graphs are similar under this test and under the M matrix.
This leads us to the following :
We either have to find a new matrix transformation that makes the M matrices not similar or we have to find a test different than the characteristic polynomial or the above mentioned matrix properties in order to distinguish the 2 matrices.
Finally, i found a new matrix, that is graph invariant and that succesfully distinguishes the all the matrices up to (including) 10 vertices and 16 edges. Maybe it works for more, but i didn't have the time to check it since it was a last minute result.

This succesfully solves the problem i have been trying to solve, which is to find a graph invariant with *much* better results all all the previous known

ones. It remains to be shown, up to what number of vertices this graph invariant works.
The matrix is defined as the M matrix above, but instead of using the Adjacency Matrix of the graph, we use the All Pairs Distance Matrix.
You can see all the results in Table 2 and Table 3. Table 3 is the analysis of Table 2, for 10 vertices, since this was the least number of vertices for which i could find graphs that don't pass the matrices. In Table 2 the row for 10 vertices is not complete, since it is a big computational task to calculate all these numbers.

2.4 Results

The following tables were produced using Maple.

These tables are based on the graphs that can be found in : *W.H. Haemers and E. Spence, Enumeration of cospectral graphs, European J. Combinatorics.* I have partially reproduced these results, and added the E and M columns. *Table 1 : Results from the partitioning method*

N	Total	Partition	Non Isomorphic	Connected	Bipartite	Tree
1	1	1	1	1	1	1
2	2	2	2	1	2	1
3	8	4	4	2	3	1
4	64	16	11	6	7	2
5	1024	84	34	21	13	3
6	32768	936	156	112	35	6
7	2097152	16758	1044	853	88	11

Table 2 : Number of graphs with cospectral mates

Vertices	Non Isomorphic	A	A+$\bar{A}$	L	$\|L\|$	E	M
2	2	0	0	0	0	0	0
3	4	0	0	0	0	0	0
4	11	0	0	0	2	0	0
5	34	2	0	0	4	0	0
6	156	10	0	4	16	0	0
7	1.044	110	40	130	102	0	0
8	12.346	1.722	1.166	1.767	1.201	0	0
9	274.668	51.038	43.811	42.595	19.001	0	0
10	12.005.168	2.560.516	2.418.152	1.412.438	636.607		

Table 3 : Graphs on 10 vertices with non-isomorphic cospectral mate

Edges	**Non Isomorphic**	**A**	**A+$\bar{A}$**	**L**	**$\|L\|$**	**E**	**M**	**M'**
2	2	0	0	0	0	0	0	0
3	5	0	0	0	2	0	0	0
4	11	4	0	0	2	0	0	0
5	26	5	0	0	4	0	0	0
6	66	26	2	7	11	0	0	0
7	165	62	6	21	31	0	0	0
8	428	191	22	75	80	0	0	0
9	1.103	412	86	237	155	0	0	0
10	2.769	1.068	278	568	338	0	0	0
11	6.579	1.994	831	1.279	681	0	0	0
12	15.772	4.843	2.178	2.722	1.307	0	0	0
13	34.663	8.874	5.380	5.455	2.344	0	0	0
14	71.318	18.747	11.811	10.428	4.362	8	2	0
15	136.433	31.852	24.094	18.826	8.069	26	14	0
16	241.577	56.827	44.229	31.373	13.909	58	52	0
17	395.166	87.986	75.358	47.972	21.814	166	150	
18	596.191	133.350	116.870	68.692	31.495	364	332	
19	828.728	181.236	166.403	92.350	42.534	624	600	
20	1.061.159	233.250	217.639	119.163	54.427	987	931	
21	1.251.389	273.336	260.561	145.233	65.430		1243	
22	1.358.852	294.399	283.328	161.818	72.165		1424	
23	1.358.852	291.391	283.328	161.818	72.181		1424	

3 Graph Colouring

3.1 Introduction

Graph Coloring it's an important problem and a difficult one. In the work developed during the MITACS summer school, an attempt to explore and study different formulations and heuristic for this problem were pursued.
The metodology to work on this problem is to explore several heuristic algorithm approachs in order to study the feasibility of applying them, and the quality of their solutions, in terms of the computational costs -time and memory- and nearness to the optimum value. This exploration is given by the implementation of these algorithms in Maple, and several computational test of them.
On the next sections, a revision of each of these algorithms is made.

3.2 Binary linear programming formulation

We can define an integer linear programming problem, with binary variables, to solve the graph coloring problem. Let

- $G = (V, E)$ a graph.
- $MaxColors$ the maximum amount of colors to use.
- $X_{i,k}$ a variable whose value is 1 if node i uses color k, 0 otherwise.
- $A_{i,j}$ the Adjacency matrix of the graph G.

Then, we can formulate the following:

$$\min \sum_{k=1\ldots MaxColor} \sum_{i1\ldots|V|} X_{i,k}k$$

s.t: The "proper coloring" condition

$$A_{i,j}(X_{i,k} + X_{j,k}) \leq 1 \quad \forall k = 1 \ldots MaxColor \quad \forall i, j = 1 \ldots |V|$$

and that every vertex uses at least one color -and no more-

$$\sum_{k=1\ldots MaxColor} X_{i,k} = 1 \quad \forall i = 1 \ldots |V|$$

The objective function encourages to use the least available color, since the color number is a cost in it. The first constrain models the coloring condition, that no edge-sharing vertices uses the same color (if they do, the constrain would be violated by 1). The second constrain is best viewed as a "less or equal to one and greater or equal to one", ensuring with it's first part, that

every vertex is colored, and with the second one that every vertex uses at most one color.
The parameter $MaxColors$ is used to make this LP a finite one and with a linear greedy algorithm can be obtained, since the colors given by this approach is an upper bound for the chromatic number of a graph.

3.3 Alternative degree definitions

A greedy algorithm, can be described in general in the following manner:

1. Define a degree for every vertex.
2. Choose an uncolored vertex with highest degree value given by the previous definition
3. Repeat until every vertex is colored.

Maple's Graph theory package comes with an implementation of the greedy algorithm, with the degree defined by the usual vertex degree definition. It's linear since the degree of each vertex is calculated linearly on the amount of edges, and these values are the same at each stage of the computation of the algorithm, so chosing the least available color is again linear on the amount of edges -as an upper bound because it's linear on the neighbours of each vertex.
New vertex degree definitions can be given. Let:

1. $C(v)$ the color of a vertex v, if is 0, there is no color assigned to v.
2. $N(v)$ the set of "neighbours" of the vertex v.

then, we have:

- **"Traditional" degree**, defined as:

$$d(v) = \sum_{e=uv, e\in E(G), u\in V(g)} \quad \forall v \in V(G)$$

  i.e., the amount of incident vertices to a vertex v by edges in $E(G)$.

- **Color Degree (CD)**:

$$d_{color}(v) = \sum_{e=uv, e\in E(G), u\in V(g), C(v)\neq 0} \quad \forall v \in V(G)$$

  i.e., the already colored vertices incident to v.

- **Different Color Degree (CDD)**:

$$d_{dcolor}(v) = \sum_{e=uv, e\in E(G), u\in V(g), C(v)\neq 0, C(v)\neq C(w) \forall u, w\in N(v)} \quad \forall v \in V(G)$$

  i.e., the amount of different colors that the already colored neighbours of v have.

Then, we can have the same greedy algorithm, but with this degree definitions, but more computational costs are involved since the degrees change at each time a vertex is colored. So a trade-off between time and memory can be considered, depending on the particular implementation of the algorithms.

3.4 Results

For the Binary LP problem, a test were made formulating the problem conveniently with an "if":

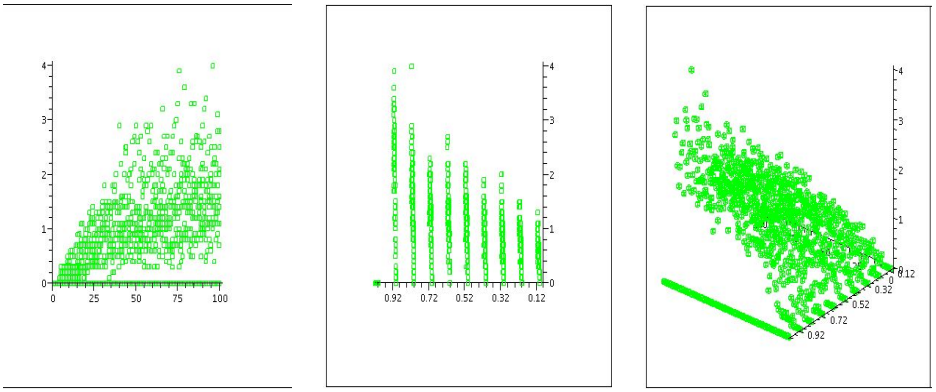
```
NumberOfNodesTest:=1;
EdgeProbability:=0.1;
MaxNodes:=12;
MaxProbability:=0.7;

for NumberOfNodesTest to MaxNodes do
 while EdgeProbability <= MaxProbability do
  G := RandomGraph(NumberOfNodesTest, EdgeProbability);
  NumberOfNodes := NumberOfVertices(G);
  MaxColors := GreedyColor(G)[1];
  Objetivo := add(add(aaPHIaa[i,j]*j, i=1..NumberOfNodes),\
    j = 1 .. MaxColors);
  constraints :=
    [seq(seq(aaPHIaa[e[1], kk]+aaPHIaa[e[2], kk] <= 1,\
    e = Edges(G)),kk = 1 .. MaxColors), \
    seq(`+`(seq(aaPHIaa[i, k],k = 1 .. MaxColors)) = 1,\
    i = 1 .. NumberOfNodes)];
  tiempo[NumberOfNodesTest, EdgeProbability] :=
    time(LPSolve(Objetivo, constraints, assume = binary));
  EdgeProbability := EdgeProbability+.1
 end do;
 EdgeProbability := .1
end do;
```

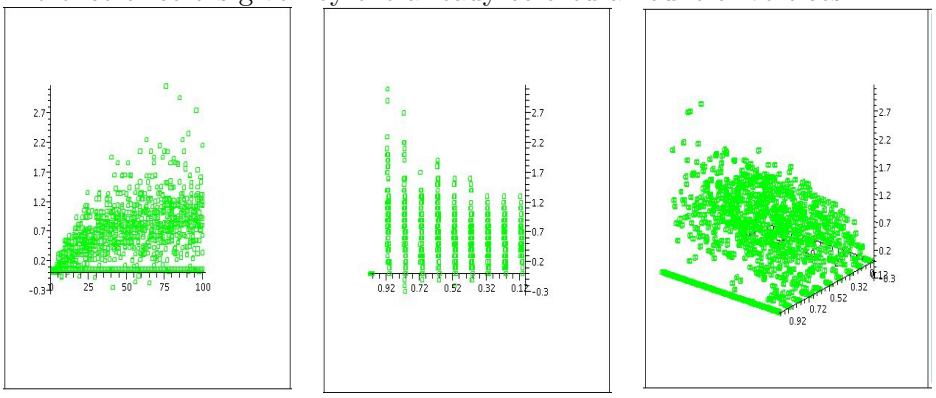
But, given that the amount of constrains grow exponentially on the size of the graph, using a branch-and-bound heuristic to solve this problem (as Maple does), produces an exponentially large solution tree, so the running time grows exponentially as well.

For the degrees defined, we have implementations of the algorithms on Maple, and them we run test on them, with a test that takes a probability parameter that runs from 0.1 to 1 and a number of vertices starting from 2 to 100, and run ten times the algorithms to take the average of the running times, and the difference in the amount of colors used compared to the greedy coloring algorithm.

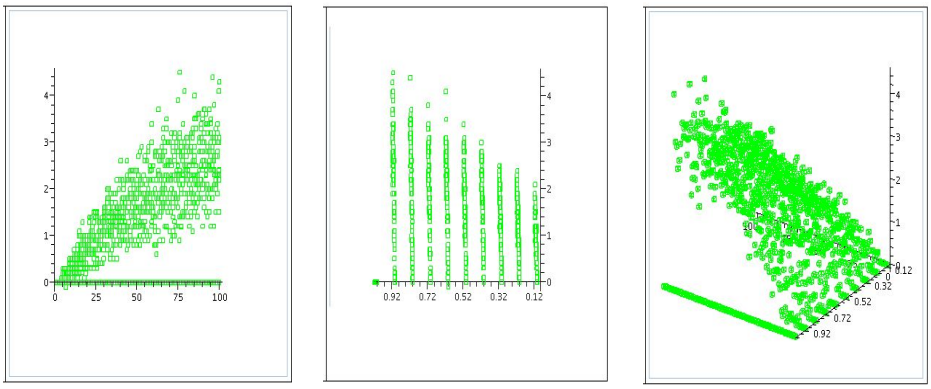
- Difference of colors given by greedy coloring, using the degree-sequence-permutation of vertices:



- Difference of colors given by the already-colored-amount of vertices:

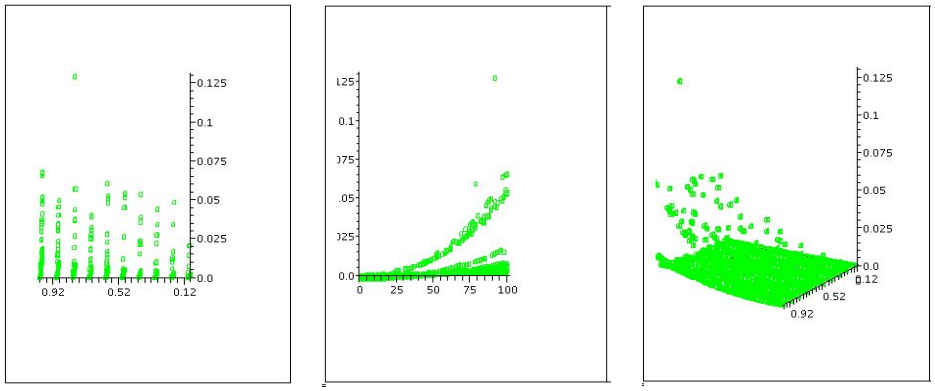


- Difference of colors by the already-colored-with-distinct-colors amount of vertices:

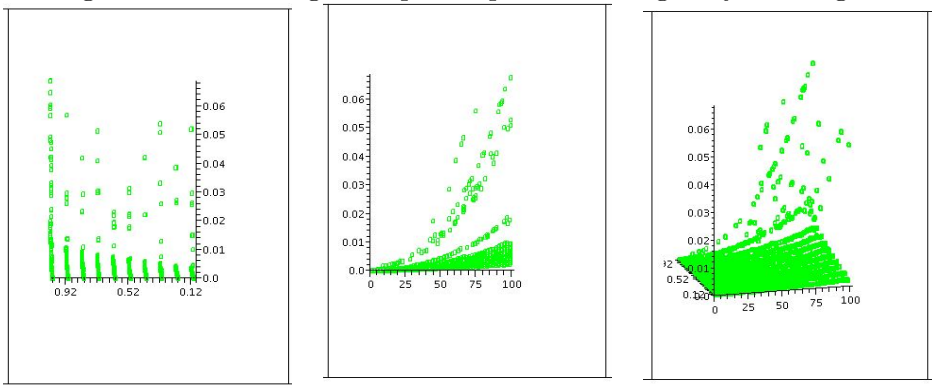


As it can be seen, the amount of different colors grow similar for the three algorithms, but sometimes the already-colored algorithm performs worse than greedy coloring, and consuming more resources. However, the difference grows stronger when bigger graphs are used, suggesting that the difference may stay stable around the value 4 (or maybe growing logarithmically), and the other coloring algorithms seem to grow stronger (even linearly). No more intensive test were performed though, due to the limitation on the summer school time.

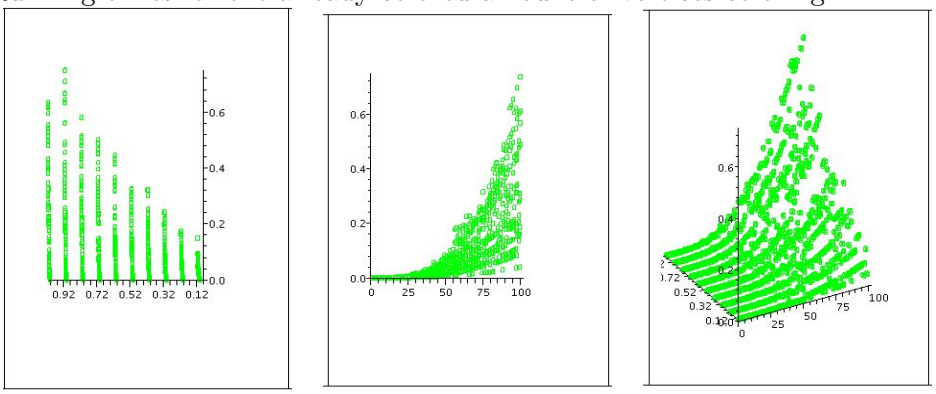
- Running times for the greedy coloring algorithm:



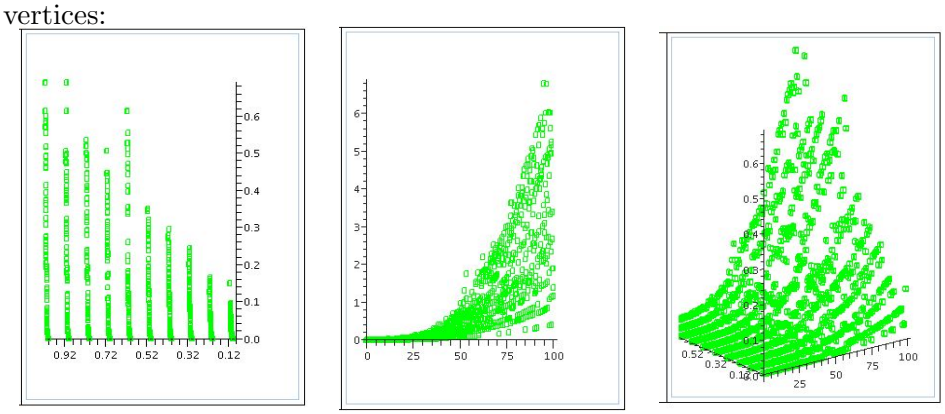
- Running times for the degree-sequence-permutation greedy coloring:



- Running times for the already-colored-amount of vertices coloring:



- Running times for the already-colored-with-distinct-colors amount of vertices:



The time grows linearly on the first cases, maybe logarithmically for the second one, since it performs a sorting procedure first. The other coloring algorithms are more time expensive, because both of them calculate at each step the set of already colored vertices -or the set of visible colors- for each vertex, then performing some searching and sorting operations at each stage of the algorithm. An improvement can be made using some suitable data structure, as a heap, for the values, but after performing some simple test, no real improvement was found -in time neither less colors-.

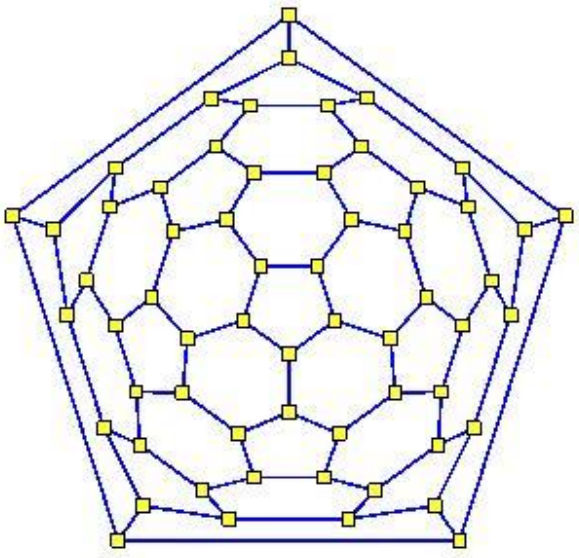


Figure 2: Buckminsterfullerene C_{60}

4 Fullerenes

4.1 Introduction

Fullerenes are carbon molecules that form a hollow ellipsoid or tube. They were first discovered in 1985 by researchers at Rice University and the University of Sussex. They were named after architect Richard Buckminster Fuller, who often used geodesic domes in his buildings. We can represent fullerenes as graphs, treating carbon atoms as vertices and carbon bonds as edges. In figure 2, for instance, is a planar graph representation of Buckminsterfullerene, C_{60}, the first fullerene to be discovered.

In graph theory, fullerenes refer to any 3-regular, planar graphs comprised strictly of pentagonal and hexagonal faces. The smallest such fullerene is the dodecahedron graph, which is comprised of 20 vertices and 12 pentagonal faces. The number of fullerene isomers as a function of the number of vertices, n, is asympotically $O(n^9)$.

There exist various algorithms that ennumerate and construct fullerene isomer graphs. My research aim has been to implement an algorithm in Maple that, given a specified number of vertices, will quickly generate a random fullerene isomer. I will discuss some of the algorithms I considered. In addition,

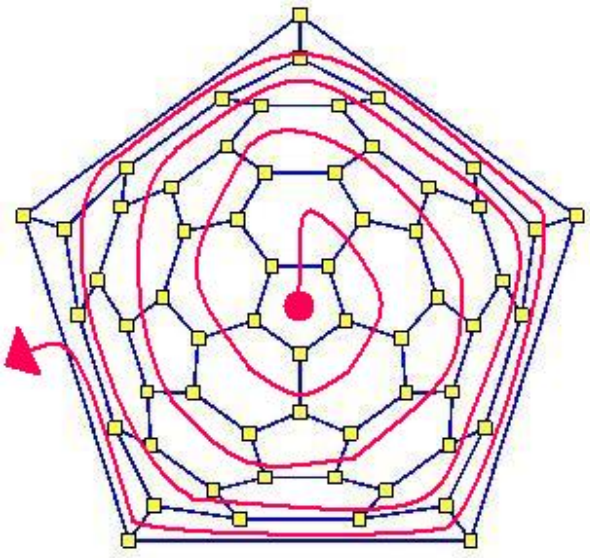


Figure 3: A spiral traversing all the faces of C_{60}. The corresponding spiral code is 5666665656565656656565656565666665.

4.2 The spiral code algorithm

Manopolous and Fowler describe a means of encoding a fullerene. Their idea is to pick a face from the fullerene graph, then another adjacent face, and then to traverse the faces of the graph in an outward spiral. If there exists a spiral traverses all the faces of the graph, we can encode the fullerene as a sequence of numbers denoting the number of edges per face, in the order that the spiral traverses them.

Not every fullerene has a spiral which touches every face of its graph; however, in practise most fullerene graphs have at least one spiral encoding. It is difficult to find a fullerene graph that does not. An n-vertex fullerene has as many as $6n$ unique spirals. We can uniquely determine a spiral by its first and second faces, and then by the direction of the spiral, either clockwise or counterclockwise. A pair of adjacent faces can be uniquely mapped to their shared edge. For every such pair, we can generate a spiral starting from either face, and turning in either direction. An n-vertex fullerene has $\frac{3n}{2}$ edges, and hence $\frac{3n}{2}$ adjacent face pairs, resulting in a maximum of $6n$ spirals. Any fullerene which cannot be encoded by a spiral code would have that every such spiral comes to a dead-end before traversing all the faces.

While spiral codes do not encode every possible fullerene isomer, they encode virtually all fullerene isomers up to a relatively large number of vertices. One way we could conceivably generate an n-vertex fullerene isomer in a partially random fashion would be to test random spiral codes until we have one that

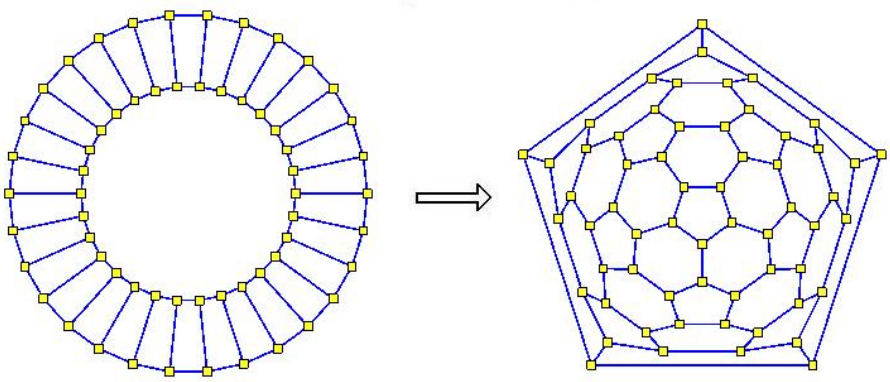


Figure 4: In the algorithm described by Plestenjak, Pisanski, and Graovac, a prism is converted into a fullerene. Observe that the number of edges, vertices, and faces is the same in both graphs.

encodes a fullerene. We can test if a spiral code in face does encode a fullerene graph by effectively piecing the faces back together one at a time. The outer face of our resulting planar graph should have as many edges as the last digit of our spiral code specifies, if it is infact a fullerene graph.
This algorithm, albeit accessible, bodes problematic in two ways. First, we already know it won't generate some fullerene isomers. Secondly, given there are $O(n^9)$ n-vertex fullerene isomers, each of which have at most $6n$ spiral codes. Thus we have $O(n^{10})$ potentially "good" spiral codes; however, the number of candidate spiral codes grows appreciably faster with respect to n. Any sequence of 12 '5's and $(\frac{n}{2} - 10)$ '6's could conceivably encode an n-vertex fullerene. We thus have $\binom{\frac{n}{2}+2}{12}$ candidate spiral codes. Thus the proportion of good spiral codes grows exponentially small as n increases. If may be worthwhile to research means of quickly reducing our candidate set.

4.3 Constructing fullerenes from simpler graphs

Plestenjak, Pisanski, and Graovac describe an algorithm which takes an n-vertex prism and transforms it into a fullerene graph. They transform the polyhedron into a fullerene by way of the polyhedral Stone-Wales (PSW) transformation (see figure 5). The advantage of this algorithm is that a prism is very simple to construct. An n-vertex prism and an n-vertex fullerene both have $\frac{3n}{2}$ edges and $\frac{n}{2} + 2$ faces. This holds constant for every intermediate graph of the algorithm.
Observe in figure 5 that the faces f_1 and f_2 lose an edge whereas faces f_a and f_b gain an edge from the transformation. By performing many PSW transformations, we can change a prism into a fullerene; however, we cannot just perform these transformations indiscriminantly. The questions remains

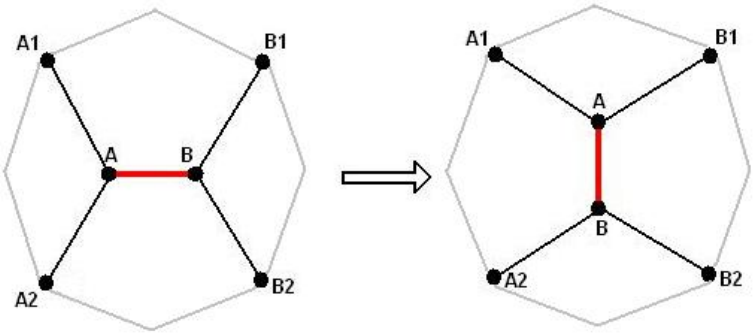


Figure 5: The polyhedral Stone-Wales (PSW) transformation on edge {A,B}. The transformation is effectively two edge insertions ({A,B1}&{B,A2}) and two edge deletions ({A,A2}&{B,B2}).

as to which is the most appropriate edge to transform.
Plestenjak, Pisanski, and Graovac propose a selection rule. They first define a constant $\bar{f}$ and function $E(G)$ on our graph G as follows:

$$\bar{f} = (3v)(\cdot\frac{1}{\frac{n}{2}+2})$$

$$E(G) = [\sum_{i=1}^{\frac{n}{2}+2}(|f_i - \bar{f}|)] - 12(\bar{f}-5) - (\frac{n}{2}+2)(6-\bar{f})$$

f_i is simply the number of vertices (or equivalently, the number of edges) in the face f. $\bar{f}$ is simply the average number of edges per face in a polyhedron graph with n vertices. $E(G)$ has two useful properties:

1. $E(G) \geq 0$,
2. $E(G) = 0 \iff G$ is a fullerene graph

So, our aim is to minimize $E(G)$. We assign every edge $e \in G$ a weight $\omega(e)$:

$$\omega(e) = E(PSW(G,e)) - E(G)$$

, where $PSW(G,e)$ is the graph G after the PSW transformation has been performed on edge e. The polyhedral Stone-Wales tranformation only affects four faces (labelled f_1, f_2, f_a, f_b in figure 5) of our graph. We can reexpress $\omega(e)$ as follows:

$$\omega(e) = |f_a + 1 - \bar{f}| + |f_b + 1 - \bar{f}| + |f_1 - 1 - \bar{f}| + |f_2 - 1 - \bar{f}| - |f_a - \bar{f}| - |f_b - \bar{f}| - |f_1 - \bar{f}| - |f_2 - \bar{f}|$$

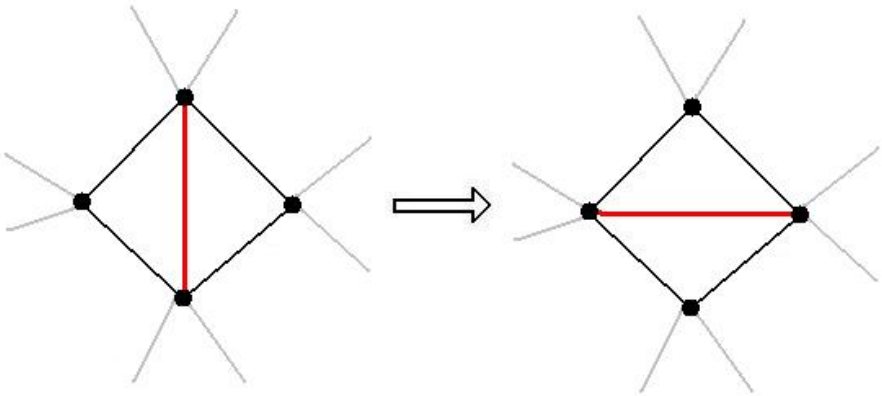


Figure 6: The dual PSW transformation.

If there exists an edge $e \in G$ for which $\omega(e) < 0$, we simply pick an edge with minimum weight in G at random. If all the edge weights are positive, however, we pick a random subset of around 3 to 5 edges, and pick the edge of minimal weight in that set. The reason for this is that there are instances in which strictly choosing a minimum-weight edge at every step will cause the algorithm to loop.

We can save time by performing all the transformation in the dual of our polyhedron graph, and then constructing our fullerene from the dual upon completion. We half the number of edge insertions and deletions by doing so.

Each step of the algorithm takes $O(n)$ operations. We need to do a find-min operation to select an edge. Additionally, for every PSW transformation, we need update every edge with a vertex on one of the four affected faces. The average number of transformations needed to obtain a fullerene appears in practise to be approximately $O(n)$ as well, thereby resulting in a quadratic-time algorithm.

One area for future study is the edge-selection rule. In practise, the function $E(G)$ decreases very quickly at the start of the algorithm and then approaches 0 very slowly as the algorithm progresses. We hope to shorten the running-time of the algorithm by devising a quick means to determine how to obtain a fullerene in a minimal number of steps, particularly when we have a graph G for which $E(G)$ is close to 0.

5 Planar Drawing

5.1 Introduction

In graph theory, we defined a planar graph to be a graph that can be drawn so that no edges intersect in the plane. A planar graph already drawn in the plane without edge intersections is called a plane graph. It would be nice to include in Maple a graph drawing method to display every planar graph in its plane graph form. A function as such has not yet been implemented in Maple 11. One of our goals this month is to develop a intutive way of planar drawing.

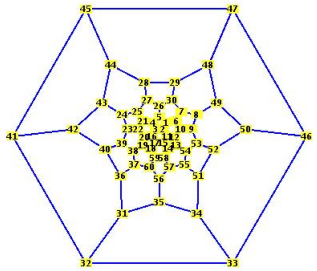
The planar drawing problem seems intuitive. However, it is also difficult in the sense that one has to ensure the computer is smart enough to make a decent drawing for every planar graph. In this project, we did not try to use artificial intelligence to solve the problem instead we used graph theory and some fundamental mathematics.

We can breakdown this problem into smaller ones. It's fairly easy to decompose a general planar graph into its 3-connected subgraph components. Thus, if one can first produce a good plane drawing for any 3-connected planar graph, the problem is nearly solved. This is the essential problem that Wendy has been working on for this part of the project. She began by studying a method called RubberDraw which Mohammad has implemented for drawing 3-connected planar graphs.

5.2 RubberDraw

RubberDraw takes in a graph of n vertices with the underlying constraint that it is a 3-connected planar graph and returns a plane drawing of the graph. The algorithm picks from all its faces the largest cycle with the most number of vertices k to be the outer face. It positions the selected vertices on a unit circle at the roots of unity. Now in order to determine the positions for the rest of the vertices of the graph, the algorithm places each of the remaining vertices at the central of gravity of its neighbours. In this way, the algorithm constructs a system of (n - k) linear equations in (n - k) variables. The positions of the rest of the vertices are obtained by solving the system of equations.

The outcome from calling the RubberDraw function on a 3-connected planar graph gives a plane drawing of the graph with no two edges intersecting. However, the defect of this algorithm is obvious just by testing the function against some planar graphs included in the SpecialGraphs package of Maple. For example, when we draw the DodecahedronGraph using RubberDraw,



.

Figure 7: RubberDraw(S); S:=SoccerBallGraph();

the vertices inside the outer cycle are more concentrated towards the centre. The drawing becomes even worse as the number of vertices of the graph increases. Figure 7 shows the SoccerBallGraph with 60 vertices and we notice that the inner vertices are all crowded towards the center. RubberDraw has taken the first step in making plane drawings of general 3-connected planar graphs. We now need to find ways to improve RubberDraw to address the rescaling problem.

5.3 Experiments and Observations

There is one graph in particular of which the RubberDraw method returns a good plane drawing. RubberDraw displays the GrinbergGraph nicely with the inner verticesbegin spreadout evenly inside the outer cycle. In Figure 8 one would notice that outer cycle of the Grinberg graph is a enneagon and has 9 edges connected to the vertices inside. These edges seem to pull more of the inner vertices towards the outer cycle and thus avoided the cluster of vertices at the center.

In general, if there are more edges connecting the outer vertices with the inner ones of a graph, then RubberDraw may return a better plane drawing of the graph. It's clear that we want to improve our RubberDraw function so that most of the resulting graphs would be displayed in the patterns of the bottom row. Wendy implemented this idea into RubberDraw. The updated RubberDraw again picks the largest cycle to be the outer face, it then adds edges connecting each of the outer vertices to the nearby two inner vertices as shown in Figure 9 [left]. The rest of the algorithm stays the same, except at the end the additionally added edges are removed. The resulting SoccerBall graph is shown on the right hand side of Figure 9.

This trial turns out well but we have to keep in mind that up to now only the nice graphs in the SpecialGraphs Package are tested. We have not yet

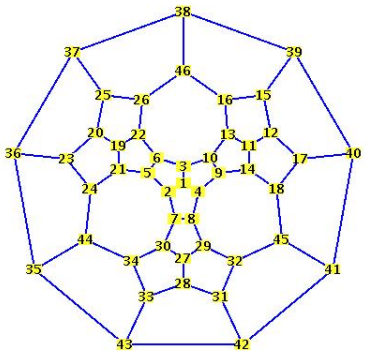


Figure 8: RubberDraw(G); G:=GrinbergGraph();

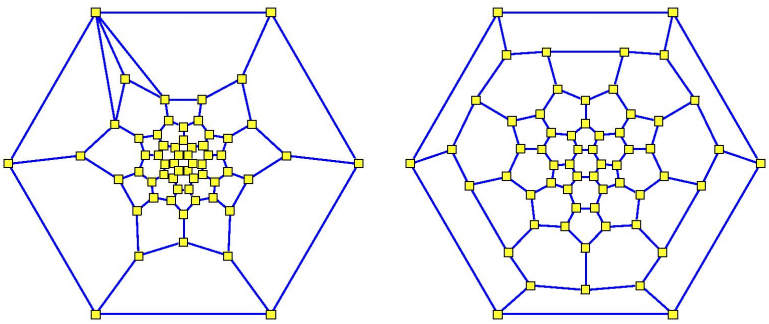


Figure 9: SocceBallGraph redrawn using the updated RubberDraw

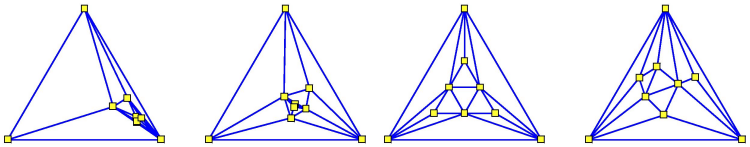


Figure 10: patterns of results from the original RubberDraw

tested our RubberDraw with more general 3-connected planar graphs. In Maple 11, there is no method of generating random 3-connected graphs. However, Alejandron has implemented an algorithm of generating random planar triangulation graphs. In the previous experiment of trying to improve RubberDraw on some graphs in the SpecialGraphs package, we succeeded by triangulating parts of the graphs. It helps to examine how RubberDraw behaves with a random planar triangulation. Mohammad and Wendy tested the original RubberDraw function with 50 randomly generated planar triangulations of 9 vertices. Most of the outcoming graphs fall into the following 4 patterns as shown in Figure 10.

We see that the first two graphs are badly drawn. There is again clustering of the inner vertices in some areas of the graphs. Unfortunately out of the 50 graphs drawn, many outcoming graphs exhibit these two patterns. The last two graphs are much better. However, notice that the third graph has a lot of symmetry which is not the case with other general graphs. The best we can do is to bring most of our graphs to the pattern of the very last one.

5.4 A improvement to RubberDraw

We studied the bad graphs and the good graphs by going through lists of random planar triangulations generated in the experiment. Soon it's clear that in all the better drawn graphs, the vertices of the selected outer face has similar degrees. Here the degree of a vertice means the number of neighbours of the vertex. On the other hand, bad graphs such as the one at the top left have inbalancies in the degrees of vertices of its outer face. However, the graph at the top right does have good balancy in its degrees of vertices on the outer cycle. The problem with this graph is that the outer vertices have small degrees and thus some inner vertices with higher degrees are all crowded in the center. Therefore, the goal is to pick the right outer face in order to achieve balancy in the degrees of outer vertices while maximizing

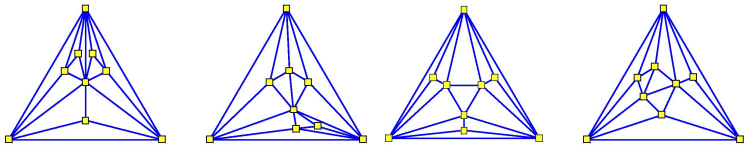


Figure 11: the same four graphs redrawn by RubberDrawv

the sum of their degrees.

Simple math was used to solve the problem in this approach. Wendy revised RubberDraw and named it RubberDrawv for now. In RubberDrawv, let F be the list of faces of a graph G obtained from IsPlanar(G, 'F'). Each face is represented by a list of vertices in the face. RubberDrawv goes through F and gets the list of degree sequences of faces. In the following step, it calculates the mean deviation and the sum of degrees for each degree sequence. They are stored in lists 'mdev' and 'dssum' respectively. Now we take the ratio

$$\frac{mdev[i] + NumberOfVertices(G)}{dssum[i]}$$

where the i is the index to the faces of the graph. The ratios are stored in a list called dv. The function then picks F[i] with the smallest ratio dv[i] to be the outer cycle.

```
# a list of degrees of vertices of graph G
ds := DegreeSequence(G);
# the list of degree sequences for faces of G
dsof := [seq([seq(ds[x], x=i)], i=F)];
# the mean deviation in each degree sequence
mdev := [seq(MeanDeviation(i), i=dsof)];
# the sum of degrees of each degree sequence
dssum := [seq(add(x, x=i),i=dsof)];
# the list which stores the ratios dv := [seq((mdev[i] + NumberOfVer-
tices(G))/dssum[i], i=1..nops(dssum))];
```

The rest of the function is the same as the original RubberDraw method. Figure 11 shows the same four graphs redrawn by RubberDrawv.

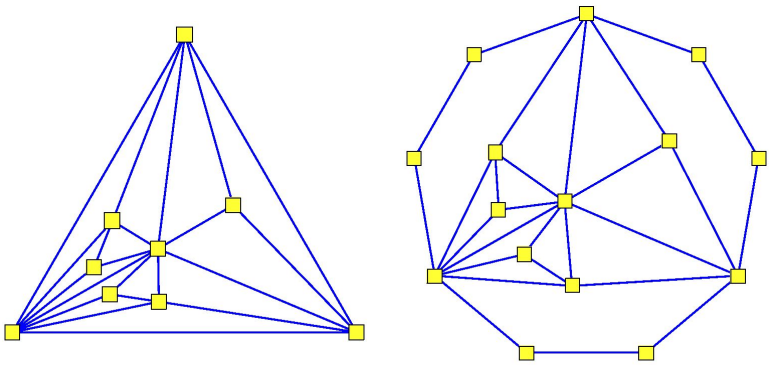


Figure 12: the last experiment

5.5 Future Work

RubberDrawv has not eliminated all the rescaling problems. As we mentioned at the beginning, it's hard to find a drawing method which can display all planar graphs nicely in the way we would draw them by hand. For example, GeneralizedPetersen graphs of many vertices have two big cycles. The plane drawings of these graphs turn out to be rings. In the past month, we have used some time to improve RubberDraw so that most of the plane drawings would be displayed properly. However, there is still rooms for further improvements. A suggestion from Mohammad is that we can replace the straight edges with arcs therefore allowing more room for positioning the inner vertices. Since there is no arc implementation at the moment, the last experiment being done in the project is adding vertices to the outer edges of a planar triangulation. This would cause the outer cycle to become an enneagon resembling the arcs between the 3 vertices. We observe that this has allowed more room for drawing the graph as shown in Figure 12.

6 Bibliography

- "New Methods to Color the Vertices of a Graph", Daniel Br'elaz, 'Ecole Polytechnique F'ed'erale de Lausanne.
- "Enumeration of cospectral graphs", W.H. Haemers and E. Spence, European J. Combinatorics.
- P. W. Fowler, D. E. Manolopoulos, *An Atlas of Fullerenes*, Clarendon,Oxford, 1995.
- T. Pisanski, B. Plestenjak, and A. Graovac. *Generating Fullerenes at Random*. J. Chem. Inf. Comp. Sci., 36:825–828, 1996.
- G. Brinkmann and , A.W.M Dress. *A Constructive Enumeration of Fullerenes*. J. Algorithms, 23:345 - 358, 1997.