Generating Polyhedral Molecules from Planar Embeddings of 3-Regular Graphs

Wendy Myrvold
University of Victoria

David Mitchell
University of Victoria

April 29, 2017

1 Introduction

An *undirected graph* G consists of a set of vertices V(G) and a set of edges E(G) where each edge corresponds to an unordered pair of vertices [2]. The sets V(G) = {0, 1, 2, 3, 4, 5, 6, 7} and E(G) = {(0,1), (0,2), (0,3), (1,2), (1,3), (2,4), (3,5), (4,6), (4,7), (5,6), (5,7), (6,7)} represent the undirected graph G shown in Figure 1. A graph is *connected* if for every partition of V(G) into two non-empty sets A and B there is an edge containing a vertex from A and a vertex from B [2].

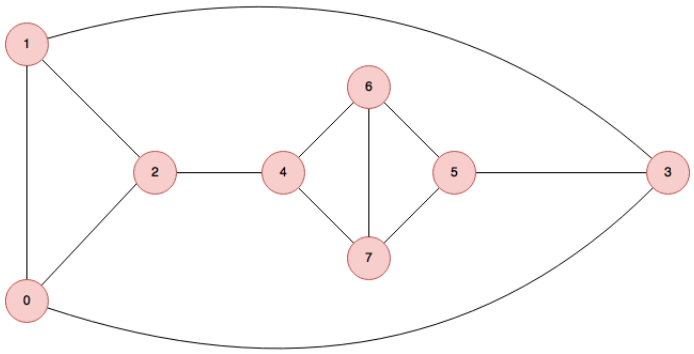

Figure 1: Planar embedding of undirected graph G

The vertices of the graphs we will generate represent either carbon atoms or nitrogen atoms. The edges represent bonds between atoms. While these graphs contain no self-loops in which a vertex is connected to itself by an edge, multiple edges between vertices are allowed. For example, the undirected graph H represented by V(H) = {0, 1, 2, 3} and E(H) = {(0,1), (0,1), (0,2), (1,3), (2,3), (2,3)} shown in Figure 2 contains multiple edges.

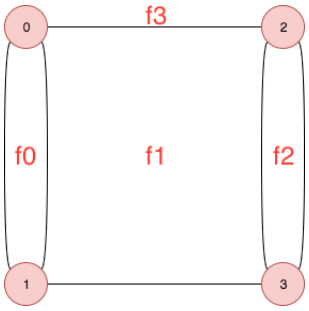

Figure 2: Planar embedding of graph H with multiple edges

The *degree* of a vertex v is the number of edges incident to v [2]. If a graph is 3-regular, then every vertex has degree three and the graph is called *cubic*. The graphs shown in Figures 1 and 2 are examples of cubic graphs.

A graph which can be drawn in the plane so that no two edges intersect except at the vertices is called *planar* and a representation in the plane is a *planar embedding* [4]. A graph can be embedded on the plane if and only if it is also embeddable on the sphere. A planar embedding partitions the plane into faces. The set of faces F(H) of graph H in Figure 2 is $\{f_0, f_1, f_2, f_3\}$ where f_0, f_1 and f_2 are bounded faces and f_3 is the external face.

If S is a subset of the vertices V(G) such that every vertex is either in S or is adjacent to a vertex in S then S is a called a *dominating set* [2]. As well, if S is a subset of the vertices V(G) such that no vertex in S is adjacent to another vertex in S then S is called an *independent set* [2]. The set S = {2,5} of graph G in Figure 1 constitutes a dominating set which is also an independent set.

For S $\subseteq$ V, the subgraph H induced by S has V(H) = S and E(H) = {(u,v): (u,v) $\in$ E(G), u $\in$ S and v $\in$ S} [2]. The graph in Figure 3 is an induced subgraph of G on the vertices S = {0,4,5,6,7}.

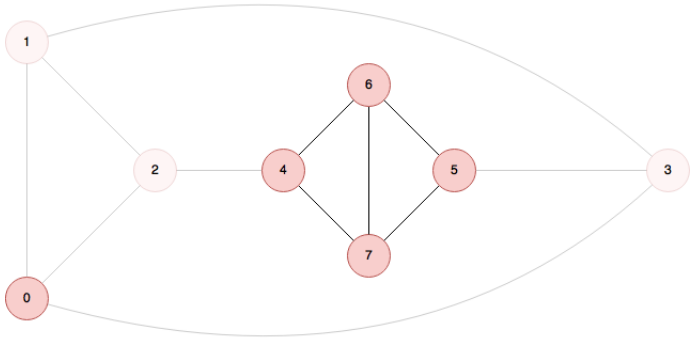

Figure 3: Subgraph of G induced by S

A *matching* (which is also referred to as an independent set of edges) is

a set of edges in which no two edges are incident to the same vertex [2]. In a perfect matching, every vertex of the graph is incident to an edge in the matching. The set of edges {(0,1), (2,4), (3,5), (6,7)} in Figure 4 is a perfect matching.

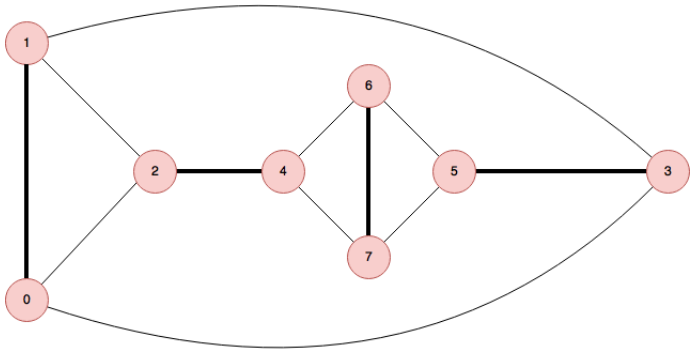

Figure 4: A perfect matching

The objective of our research is to discover a graph-theoretical solution to a problem pertaining to the generation of new polyhedral molecules [3]. Specifically, we wish to generate polyhedral molecules that are composed of carbon, nitrogen and hydrogen atoms and have the steric requirements that bonds between nitrogen atoms are avoided and each carbon atoms has a bond to exactly one other carbon atom. The resulting molecules are restricted to the molecular formula $N_{4q}(CH)_{6q}$ where q is a positive integer. Each nitrogen atom is bonded to three carbon atoms and each carbon atom is bonded to two nitrogen atoms, one carbon atom and one hydrogen atom, although the hydrogen vertices of the CH pairs are suppressed in our representation. Our aim is to generate planar embeddings which fit these requirements by modelling each molecule as a cubic graph composed of vertices labelled N or

C. The graph of Figure 5 is an example of a solution.

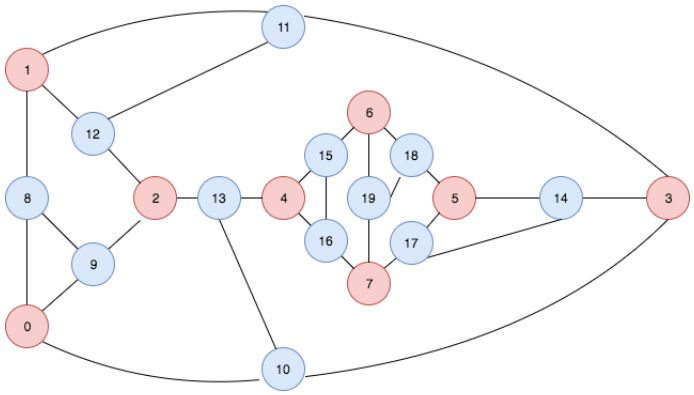

Figure 5: A solution of molecular formula $N_8(CH)_{12}$. N vertices are red and C vertices are blue.

2 Data Structures for Graph Embeddings

An *adjacency list* is a collection of n lists, one for each vertex v_i where $0 \leq i \leq n-1$, in which each entry in the list is a vertex to which v_i is adjacent. A *rotation system* is an adjacency list which describes a circular ordering of the graph's edges around each vertex. A rotation system for a planar embedding of a graph lists the neighbouring vertices in clockwise order. For this application, an array is used to store the contents of each list in the rotation system.

If a graph has multiple edges, the edges are not uniquely defined by the vertices they are incident upon. Edge numbers are assigned to each edge e to identify them.

Figure 6 is a representation of the embedding of G in Figure 1 using a rotation system.

v_i	neighbour vertex (edge number)
0	1 (0), 2 (1), 3 (2)
1	0 (0), 3 (3), 2 (4)
2	0 (1), 1 (4), 4 (5)
3	0 (2), 5 (6), 1 (3)
4	2 (5), 6 (7), 7 (8)
5	3 (6), 7 (9), 6 (10)
6	4 (7), 5 (10), 7 (11)
7	4 (8), 6 (11), 5 (9)

Figure 6: A representation of Figure 1 using a rotation system

Given a rotation system, Algorithm 1 can be used to *walk the faces* of an embedding, where walking a face means to traverse the boundary of the face [7].

Algorithm 1 Face walking algorithm for a planar embedding

```
1:  procedure WalkFaces(G)
2:      # G is a graph object which contains a rotation system denoted adj_list that is an array of vertices
3:      # v.return_neighbour(G,j) returns the vertex at position j in v's array of neighbours
4:      # v.next_pos_after_edge(e) returns 1 + position of e mod v.num_nbrs in v's array of edges
5:
6:      for i in 0..G.num_vertices-1 do
7:          v ← G.adj_list[i]
8:          for j in 0..v.num_nbrs - 1 do
9:              v.nbrs[j].visited ← False
10:     for i in 0..G.num_vertices-1 do
11:         v ← G.adj_list[i]
12:         for j in 0..v.num_nbrs - 1 do
13:             if v.nbrs[j].visited == False then
14:                 first_edge = v.edges[j]
15:                 curr_edge = first_edge
16:                 curr_v = v.return_neighbour(G,j)
17:                 first ← True
```

```
18:     while curr_edge != first_edge or first do
19:         first ← False
20:         pos = curr_v.next_pos_after_edge(curr_edge)
21:         curr_v.nbrs[pos].visited ← True
22:         curr_edge = curr_v.edges[pos]
23:         curr_v = curr_v.return_neighbour(G,pos)
```

3 Isomorphism Testing of Embeddings

The *canonical form* for our graph embeddings is the lexicographically smallest labelling of the vertices and their neighbours in the rotation system using clockwise breadth first search. If two rotation systems have the same canonical form, they are isomorphic to one another [6]. An *isomorphism* of a rotation system representing an embedding is a bijection from a rotation system G to a rotation system H such that any two vertices which are adjacent in G are adjacent in H and any ordering of vertices in G are ordered identically in H or reversed.

Algorithm 2 is an algorithm for clockwise breadth first search [5].

Algorithm 2 Clockwise breadth first search

```
 1: procedure CW_BFS(G, root, first_child)
 2:     # v.return_nbr_pos(nbr) returns the position of nbr in v's neighbour array
 3:     # parent_pos[i] maintains the position of the parent of the ith vertex in v's neighbour array
 4:
 5:     parent_pos, queue ← [ ]
 6:     queue.push(root)
 7:     parent_pos[root] ← root.return_nbr_pos(first_child)
 8:     while !queue.empty? do
 9:         v ← q.shift
10:         for i in 0..v.num_nbrs-1 do
11:             nbr_v ← v.return_nbr(parent_pos[v]+i mod 3)
12:             if parent_pos[nbr_v] = nil then
13:                 queue.push(nbr_v)
14:                 parent_pos[nbr_v] ← nbr_v.return_nbr_pos(vertex)
```

4 Finding Solutions

One approach to discovering molecules with the desired characteristics would be to factor all planar embeddings of the 3-regular graphs into sets of C and N vertices. However, for the majority of graphs this factorization is not possible. Further, there are too many embeddings to feasibly generate and check each of them beyond graphs with a smaller set of vertices. The number of cubic connected graphs with 2n nodes is given in Table 1.

n	# of cubic graphs with 2*n vertices	n	# of cubic graphs with 2*n vertices
0	1	10	510,489
1	0	11	7,319,447
2	1	12	117,940,535
3	2	13	2,094,480,864
4	5	14	40,497,138,011
5	19	15	845,480,228,069
6	85	16	18,941,522,184,590
7	509	17	453,090,162,062,723
8	4,060	18	11,523,392,072,541,432
9	41,301	19	310,467,244,165,539,782

Table 1: OEIS A002851 Number of unlabeled trivalent (or cubic) connected graphs with 2n nodes. [1]

A different approach is suggested by observing characteristics of embeddings which are solutions. First, the C-C edges of the molecule are a perfect matching on the subgraph induced by the C vertices. Second, the set of N vertices are a dominating set of the solution. Third, the set of N vertices are also an independent set.

If the C-C edges of any solution is removed, the result is a planar embedding in which each C vertex has degree two and the edges from any C vertex are incident to two distinct N vertices. In Figure 7, the C-C edges of the solution in Figure 5 are removed.

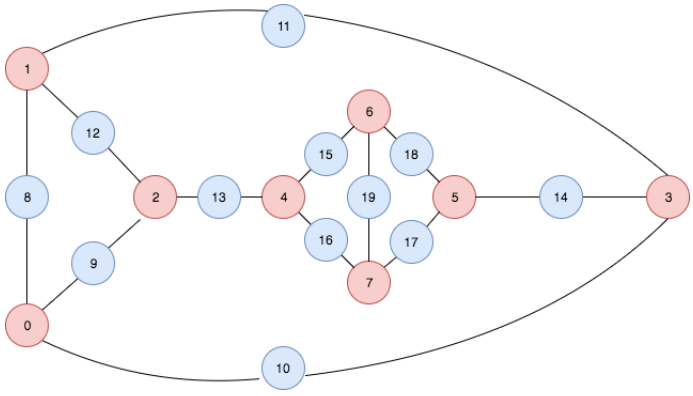

Figure 7: The solution of Figure 5 with C-C edges removed

Further, if we perform a series reduction on a series N-C-N to N-N for every series N-C-N in the embedding, the result is a smaller, 3-regular planar embedding.

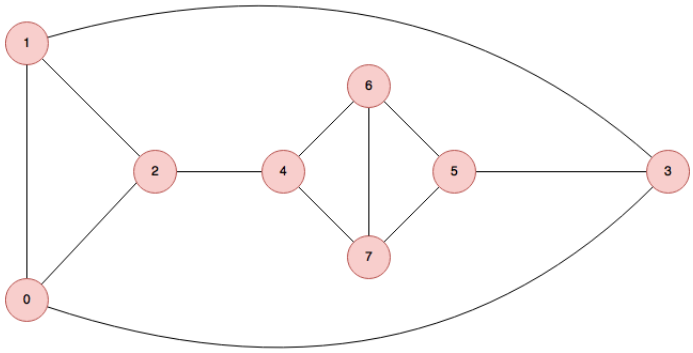

Figure 8: Underlying 3-regular planar embedding

Our observations propose a constructive algorithm for finding molecules which are solutions. We begin with a planar embedding of a smaller cubic graph for which each vertex is labelled N. We subdivide each edge with a vertex labelled C. Finally, we add edges next to vertices of degree two while

maintaining planarity in all ways until the embedding is 3-regular. The constructed 3-regular embedding is a solution.

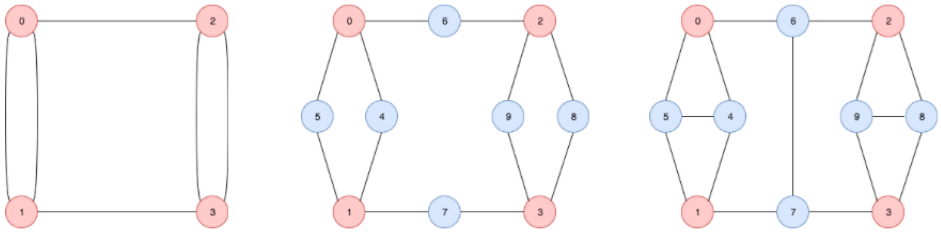

Figure 9: Constructive algorithm for finding solutions

5 Generating Algorithm - Connected Case

We begin with the rotation system of a planar embedding of a connected 3-regular graph in which every vertex is labelled N. Each edge of the embedding is then bisected by a C vertex numbered the current number of vertices in the graph.

Algorithm 3 N-N edge bisection

```
1:  procedure BISECT(G)
2:      #v.new(i,'C') creates a new vertex numbered i with label 'C'
3:      #v.add_nbr(u) adds the vertex number of u to the first empty position in v's neighbour array
4:      #v.replace_nbr(u,w) replaces the vertex number of u with w in v's neighbour array
5:
6:      for i in 0..G.num_vertices-1 do
7:          v ← G.adj_list[i]
8:          for j in 0..v.num_nbrs-1 do
9:              nbr_v ← v.return_nbr(G,j)
10:             if nbr_v.label == 'N' then
11:                 edge_num ← v.edges[j]
12:                 new_v ← v.new(G.num_vertices,'C')
13:                 new_v.add_nbr(v)
14:                 new_v.add_nbr(nbr_v)
15:                 G.add_v(new_v)
16:                 v.replace_nbr(nbr_v,new_v,j)
17:                 pos ← nbr_v.return_edge_pos(edge_num)
18:                 nbr_v.replace_nbr(v,new_v,pos)
```

Chords are then added to each C vertex in all ways maintaining planarity using a recursive backtracking algorithm as follows. Beginning with the lowest numbered vertex labelled 'C', v, a list of the vertices to which a chord can be added is generated by walking the faces v is incident to and adding any degree two vertices u_i such that the number of vertices before bisection $\leq$ i and i $<$ number of vertices after bisection. A chord is added from v to the first u_i on the list and the algorithm is called on the following C vertex. When every vertex on the graph has degree three, the rotation system is printed. When the algorithm returns, the chord is removed and a chord is added to the next vertex u_i on the list. When the list is exhausted, the algorithm returns.

Algorithm 4 Generate graphs by adding chords in all ways and maintaining planarity

```
1:  procedure ADD_CHORDS(G, level)
2:      # level is an int indicating the number of the vertex to which chords are being added to
3:      # G.get_candidates(v) walks the faces of v and returns an array of candidate objects
4:      # a candidate contains a vertex to add a chord to and its position in the rotation system
5:      if level == G.num_vertices then
6:          G.output_graph
7:          return
            v = G.adj_list[level]
8:      if v.num_nbrs < 3 then
9:          candidates = G.get_candidates(v)
10:         for Each candidate c do
11:             G.connect_vertices(v,c)
12:             add_chords(G,level+1)
13:             G.disconnect_vertices(v,c)
14:     else
15:         add_chords(level+1)
```

6 Canonical Form

If two graphs output by the graph generation algorithm are isomorphic to one another, they are considered duplicates. A set of unique solutions is found by listing the canonical form of each solution and removing duplicates. The canonical form for each graph is found by traversing the embedding and the flip of an embedding using clockwise breadth first search on each vertex and its neighbours. In the flip of an embedding the neighbours are listed in reverse order. The vertices and its edges are then renamed in the order in which they are discovered in the breadth first search.

Algorithm 5 Output the canonical form of the embedding

```
1:  procedure FIND_CF(G)
2:      # CW_BFS is a modified version of clockwise breadth first search that returns an embedding
        with vertices and edges renumbered by the index in which they are discovered
3:      # compare_cfs(min_cf,cf) returns -1 if min_cf is lexicographically smaller than cf, 0 if equal and
        otherwise 1
4:      # flip(G) returns an embedding of G in which each vertex's neighbours are listed in reverse order
5:
6:      min_cf
7:      F ← flip(G)
8:      for d in 0..1 do
9:          if d == 0 then
10:             g ← G
11:         else
12:             g ← F
13:         for i in 0..g.num_vertices-1 do
14:             v ← g.adj_list[i]
15:             for j in 0..v.num_nbrs-1 do
16:                 cf ← CW_BFS(g,v,j)
17:                 if min_cf == nil then
18:                     min_cf ← cf
19:                 else
20:                     if compare_cfs(min_cf,cf) == -1 then
21:                         min_cf ← cf
```

7 Computational Results

Solutions were generated for the connected case of 3-regular planar embeddings without multiple edges for graphs with up to 12 vertices. To obtain these results, the planar 3-regular embeddings that were used in the construction were generated by plantri by Gunnar Brinkmann (University of Ghent) and Brendan McKay (Australian National University).

The canonical form of each solution using clockwise breadth first search

was calculated and duplicate solutions were removed. The results are stated in Table 2.

# vertices in underlying graph	# vertices in bisected graph	# of solutions generated	# of solutions generated less duplicates
4	10	8	1
8	20	790	121
12	30	420,078	172,821

Table 2: Results of generating solutions in the connected case with no multiple edges

8 Future Work

Solutions for the cases of 3-regular planar embeddings with multiple edges and multiple components have not been generated. The algorithm we have described can be applied to the cases of multiple edges to arrive at a complete solution. However, in the cases of a graph with multiple components, the face walking algorithm must be revised to traverse the external edges of all disconnected components for every vertex incident to the external face. The constructive algorithm for the disconnected case is demonstrated in Figure 10.

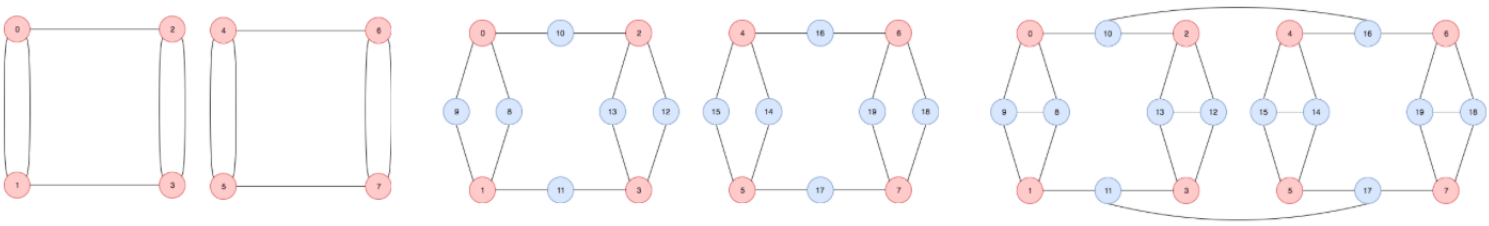

Figure 10: Constructive algorithm for finding solutions in the disconnected case

References

[1] OEIS Foundation Inc. (2017). The on-line encyclopedia of integer sequences.

[2] A. Bondy and USR Murty. *Graph Theory*. Springer-Verlag London, London, 2008.

[3] P. Fowler. Graph-theoretical solution to a chemical problem: Dominating sets and polyhedral carbon-nitrogen cages. SE Conference, FAU, March, 2017.

[4] W. Myrvold. Simpler projective plane embedding. *ARS COMBINATORIA*, 2005. 20 pages.

[5] W. Myrvold. Clockwise bfs. CSC 482/582 Class Notes, 2016.

[6] W. Myrvold. Graph isomorphism. CSC 482/582 Class Notes, 2016.

[7] W. Myrvold. Walking faces on non-orientable embeddings. CSC 482/582 Class Notes, 2016.