Fast Enumeration of All Independent Sets of a Graph Up To Isomorphism

Wendy Myrvold *
Dept. of Computer Science, Univ. of Victoria,
Victoria, BC, Canada V8W 3P6
wendym@csc.UVic.ca
Patrick W. Fowler †
Dept. of Chemistry, University of Sheffield
Sheffield S3 7HF, UK
P.W.Fowler@sheffield.ac.uk

Abstract

An *independent set* of a graph G is a set of vertices of G which are pairwise non-adjacent. There are many applications for which the input is a graph G with a large symmetry group and the goal is to generate up to isomorphism all of the independent sets, all of the maximal independent sets, or all of the maximum independent sets. This paper presents a very fast practical algorithm for these problems. The tactic can also be applied to many other problems: some examples are generation of all dominating sets, colourings or matchings of a graph up to isomorphism.

1 Introduction

Finding the order of a maximum independent set or a maximum clique of a graph (an independent set of a graph corresponds to a clique in its complement) is a problem which is NP-complete [12]. Clique-finding was one of the topics of a DIMACS implementation challenge, and the proceedings volume includes several algorithms and heuristics for this problem [16]. For some applications, the goal is to generate all independent sets, all maximal

*Supported by NSERC.
†Supported by a Royal Society Leverhulme Trust Senior Research Fellowship.

independent sets or all maximum independent sets. This paper describes a very fast algorithm for generating these up to isomorphism for situations where the input graph has a non-trivial symmetry group.

Numerous algorithms exist for generating all maximal cliques in a graph [2, 18, 24, 31, 13, 29]. There are also several algorithms for the equivalent problem of generating all maximal independent sets [17, 22, 21, 32]. Generating all possible independent sets is generally not considered because there can be such a large number of independent sets. The standard tactics for generating objects such as independent sets up to isomorphism are in [1, 25].

Two graphs considered in this paper are the graphs of the 120-cell and the 600-cell. The 120-cell is a well-known polytope that is a generalization of the regular polyhedra. The graph that represents the 120-cell is a 4-regular graph that has 600 vertices. It has 120 3-dimensional "faces" (cells) that are each dodecahedra. To build it, one can think of starting with a dodecahedron. The second layer consists of another 12 dodecahedra, each sharing one face with the one in the middle and one face with each of its five neighbouring dodecahedra. The structure continues to build outwards like this until 120 dodecahedra have been added. Stillwell provides an excellent survey of the 120-cell and its properties [30].

The 600-cell is also a generalization of the regular polyhedra. It is the dual of the 120-cell. It has 120 vertices, each corresponding to one of the dodecahedra of the 120-cell. An edge is placed between two vertices when the two corresponding dodecahedra share a face. Since each dodecahedron has 12 faces, the 600-cell is a 12-regular graph.

Our new algorithm was used to enumerate all of the independent sets of the 600-cell up to isomorphism [7]. The speed of the new algorithm was 1000 times faster than the traditional parent-child type of generation scheme for this problem. A small modification enabled enumeration of the maximal independent sets up to isomorphism. The write-up of this result [7] does not include the algorithm (It refers to the present paper).

The types of problems for which this algorithm is most applicable are cases where the graph has a reasonably sized group that is still small enough that all its permutations can be stored in internal memory. Space issues become a concern if the group is enormous. But note that in this case, a subgroup could be applied instead of the full group, with the result that the independent sets generated would be ones which are different with respect to the actions of that subgroup (Further screening if desired could be applied to the independent sets generated).

A d-dimensional Keller graph, G_d, has vertices which are numbered

with each of the 4^d possible d-digit numbers which have each digit equal to 0, 1, 2, or 3. Two vertices are adjacent if their labels differ in at least two positions, and in at least one position the difference in the labels is 2 modulo 4. These graphs have been used extensively for testing maximum clique algorithms. It seems that most heuristics fall short of the correct maximum clique order (60) for dimension six. The outstanding question for these graphs was whether the 7-dimensional Keller graphs have maximum clique order equal to 128 or less than 128 [19] (for dimension d with $d \geq 8$, the maximum clique order is 2^d [20, 23]). The algorithm presented here was used for some of the preliminary isomorphism testing for a computation which determined that the maximum clique order is 124 for dimension 7 [5] meaning that the maximum clique order is now known for all the Keller graphs.

Fullerenes are all-carbon molecules whose molecular structures correspond to 3-regular planar graphs that have face sizes equal to five or six. The C_{60} fullerene is the unique fullerene on sixty vertices with isolated pentagons. A fascinating observation is that when the C_{60} fullerene reacts with bromine, the endproduct is such that the locations of the bromine atoms correspond to a maximum independent set in the fullerene [9]. The final product is a compromise between a combination of steric constraints (bulky atoms cannot be placed close together) and electronic constraints (it is important that each connected component of the graph induced by the carbons not bonded to bromines is *closed-shell*; that is, each of these components has even order and has an adjacency matrix for which exactly half the eigenvalues are strictly positive).

A picture of C_{60} showing the independent set that corresponds to the positions of the bromine atoms in the molecule $C_{60}Br_{24}$ is shown in Figure 1. Of the 120 automorphisms of the graph, 24 preserve this independent set.

The *closed-shell independence number*, the largest order of an independent set for which a closed-shell π-configuration is possible, encapsulates this combination of steric and electronic constraints. The closed-shell independence number was known for a small number of fullerenes [11]. It is not known if this parameter is computable in polynomial time, but there is now an exponential algorithm for finding closed-shell independence numbers of fullerenes which works well in practice [3].

One open question is to determine feasible reaction pathways (sequences of additions or removals of bromine atoms) leading to the final product $C_{60}Br_{24}$. Finding all the independent sets of C_{60} up to isomorphism is a possible first step towards investigating this problem. This application of the algorithm and results for the second most common experimental

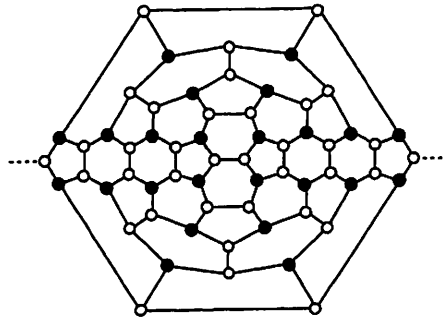

Figure 1: C_{60} with the vertices of a maximum independent set coloured black.

fullerene, C_{70}, are described in Section 3.

Our new algorithm can be modified to avoid the smaller independent sets and those which cannot lead to a closed-shell configuration, in order to compute all the maximum closed-shell independent sets up to isomorphism. We applied this to the icosahedral isomer of C_{80} (with automorphism group order 120) and determined that the closed-shell independence number is 28 and that there are 147 ways up to isomorphism to select a closed-shell independent set of order 28. This result was not previously known. One maximum closed-shell independent set of C_{80} is shown in Figure 2. The independent set is closed-shell because the graph created by deleting the independent set vertices consists of two components as pictured in Figure 3 (which is closed-shell) and the remaining 16 components are K_2's (K_2 is also closed-shell).

The total number without regard to isomorphism of independent sets of a molecular graph (defined to include also the unique set of order zero) is an invariant that appears in structure-property correlations in the chemical literature as the Merrifield-Simmons index [26, 27]. Since the automorphisms of the independent sets are available with the present algorithm, it is straightforward to obtain the index as a byproduct of the calculation, simply counting each set with weight equal to the order of the automorphism group of the graph divided by the number of automorphisms mapping the independent set to itself.

The *d-code problem* is one in which independent sets are constructed such that all pairwise distances are at least d (where $2 \leq d \leq D$, the diameter of the graph). These d-code problems can be considered as independent-set problems on an augmented graph where extra edges have been added

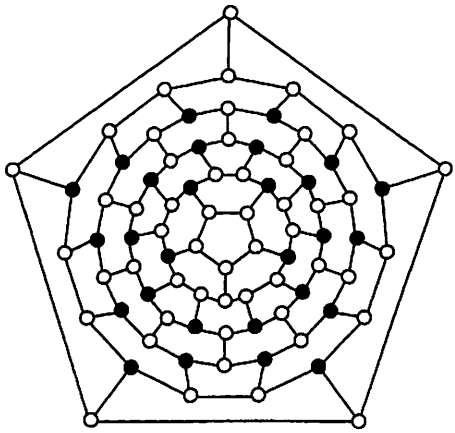

Figure 2: C_{80} with the vertices of a maximum closed-shell independent set coloured black.

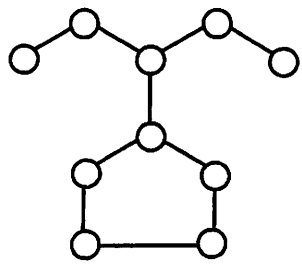

Figure 3: Two of the components of C_{80} minus the independent set vertices from Figure 2 look like this.

to connect all vertices at distances $2, \ldots, d-1$ in the original graph. In a chemical context, the hierarchy of d-code problems is a model for additions in which the addends cast larger and larger steric shadows. Orders and explicit examples of d-codes have been calculated for several families of polyhedra important in chemistry, and for C_{60} and C_{70} [8, 6].

The natural next step is to compute d-codes for regular polytopes. One of the hardest open questions for this project [4] was the order of an independent set of the 120-cell augmented in such a way that edges were added between vertices of distance at most three from each other in the original 120-cell, i.e., the order of the 4-code of the 120-cell. This problem seemed extremely difficult, and timing estimates for all approaches which would have applied more usual tactics showed that completing the computation by these approaches would be infeasible. However, the entire computation was completed in under one day by using this new algorithm.

We have chosen to include the detailed description of the computer search for the 4-code of the 120-cell in this paper instead of in [4] because it demonstrates a novel application of the algorithm. Instead of working with the entire graph (where the search time would have been prohibitive), the search commenced by selecting a subgraph of the graph and the subgroup that fixes the subgraph. A non-standard approach (based on a penalty function instead of restrictions on the number of independent vertices chosen from the subgraph) was applied to restrict the search to independent sets of the subgraph that could possibly lead to an independent set of the desired order.

2 The Algorithm

The algorithm is presented in terms of generating all independent sets up to isomorphism. There are simple modifications described at the end for the cases when the goal is to generate just the maximal or maximum independent sets.

The standard tactic which is often used to generate objects up to isomorphism is a parent-child generation scheme [1, 25]. The idea is that each object to be generated (e.g., an independent set) has a unique *parent* object (e.g., an independent set with one fewer vertex). Objects are accepted in the generation process only when they have been created from their parent, and parents do not spawn isomorphic children. This ensures that for each object, only one representative of its isomorphism class is generated. Various tricks can be used to speed up algorithms of this type [25].

The approach here is much simpler. It is this simplicity that makes the

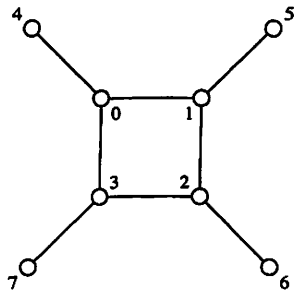

Figure 4: An example of a graph for the algorithm.

algorithm extremely fast. Consideration of the number of objects enumerated (e.g., Table 2 in Section 3) shows that it is critical that each step of the generation process should involve very little work.

The input to the algorithm is a graph G, whose vertices are labelled from 0 to $n-1$, and its automorphism group. The automorphism group is presented by giving the number of permutations in the group followed by the permutations which are in the group.

Vertices are coloured *red* if they are in the current independent set, *blue* if they are not, and *white* to indicate that their status has not yet been decided. The *canonical form* for each independent set is defined for this algorithm to be the one that gives a lexicographically minimum string when the colours are read off in the same order that the vertices are numbered. For the applications described, it is assumed that the colour red is less than the colour blue.

Figure 4 shows a sample input graph for the program. The automorphism group order for this graph is eight. The permutations in the automorphism group are:

$$01234567,\ 12305674,\ 23016745,\ 30127456,$$
$$03214765,\ 32107654,\ 21036547,\ 10325476.$$

The independent sets $\{0,2,5\}$, $\{0,2,7\}$, $\{1,3,4\}$, and $\{1,3,6\}$ are isomorphic. The canonical form for these is $\{0,2,5\}$ since it is lexicographically smallest. The colour sequence for $\{0,2,5\}$ is red, blue, red, blue, blue, red, blue, blue.

The algorithm as presented is recursive. At level k, vertices $0,\ldots,k-1$ have already been assigned the colour blue or red and the remaining vertices

are white. Consider an automorphism p. The automorphism p relabels the vertices of the graph. The interpretation used for the relabelling defined by $p = \pi_0\pi_1...\pi_{n-1}$ is that the vertex that becomes vertex i after the relabelling is vertex π_i. For some integer w, the vertices that p relabels as $0, 1, \ldots, w-1$ have the property that they are relabellings of the vertices $0, \ldots, k-1$ and hence have all been assigned colours (red or blue) but the vertex which is relabelled as w is still white because w is a relabelling of some vertex v with $v \geq k$. That is, the permutation $p = \pi_0\pi_1...\pi_{n-1}$ has the property that the values $\pi_0\pi_1...\pi_{w-1}$ are all less than k but π_w is at least k.

For the automorphism p the new colours of vertices $0, 1, \ldots, w-1$ can be compared to the current colours of vertices $0, 1, \ldots, w-1$. If the colouring resulting from the relabelling of the vertices is lexicographically smaller, the current colouring is rejected. If it is larger, then it will always be lexicographically larger until we back up and change some of the colours on the first k vertices, and so, for any recursive calls spawned with this initial prefix, the automorphism can be ignored. If the colourings up to this point are equal, then the automorphism should be retained for further comparison.

As an example of this, consider the graph in Figure 1. Suppose that the colours of the first $k = 6$ vertices so far are:

Vertex:	0	1	2	3	4	5	6	7
Colour:	blue	blue	blue	blue	blue	red	white	white

If the vertices are relabelled by the permutation 21036547 then the new colours resulting are:

Vertex:	0	1	2	3	4	5	6	7
Permutation:	2	1	0	3	6	5	4	7
Colour:	blue	blue	blue	blue	white	red	blue	white

In this case, $w = 4$ since the first four vertices are coloured red or blue after the relabelling but the 5th vertex is still white. The relabelled colouring is the same up to this point as the one being computed and so this automorphism would be retained for further comparison.

If the vertices are relabelled by the permutation 12305476 then the new colours resulting are:

Vertex:	0	1	2	3	4	5	6	7
Permutation:	1	2	3	0	5	4	7	6
Colour:	blue	blue	blue	blue	red	blue	white	white

In this case $w = 6$ since the first six vertices are all coloured red or blue after the relabelling but the 7th vertex is white. This relabelled colouring is lexicographically smaller than the current one, and so the current one is rejected.

As another example, suppose that the colours of the first $k = 3$ vertices so far are:

Vertex:	0	1	2	3	4	5	6	7
Colour:	red	blue	blue	white	white	white	white	white

If the vertices are relabelled by the permutation 21036547 then the new colours resulting are:

Vertex:	0	1	2	3	4	5	6	7
Perm.:	2	1	0	3	6	5	4	7
Colour:	blue	blue	red	white	white	white	white	white

This case has $w = 3$ and the colouring is lexicographically larger than the current one and so any calls spawned from this can ignore this permutation since the result is always going to be lexicographically larger.

The key to making our algorithm so fast is to use data structures such that only a very small amount of work is being done at each level of recursion on average. The automorphisms that are equal each have a first white vertex whose colour should be decided in order to continue the comparison. Each recursive level v retains a stack indicating the automorphisms whose first white vertex is v. For efficiency purposes it is critical that a stack is used and not a queue. When a colour is assigned at level v, the stack for level v is examined (without removing any of its contents) and the comparison with the automorphisms on that stack is extended until the next white vertex is encountered. If an automorphism still gives an equal colouring and the next vertex it awaits is w then a new record is placed on the stack at level w to record this.

The recursive call at level v colours vertex v red, then white, then blue, then white. When the status of a vertex is changed from red or blue back to white, the records that were placed in the stacks for some future level w because of that colouring are removed. Here we take advantage of the fact that the data structure used is a stack and observe that it is sufficient to do a deletion from the top of the stack once for each record that was added to each stack, without worrying about precisely which records are being deleted.

In order to avoid dynamic allocation of the records to be placed on

these stacks, the current implementation preallocates an array *stack_cell* of structures which is of size $n+1$ by g where n is the number of vertices and g is the group order. The entry in *stack_cell*$[v][p]$ is used if automorphism p, which is stored in *auto*$[p][0..n-1]$, has a record in the stack for the level v recursive call.

The record *stack_cell*$[v][p]$ contains the following pieces of information:

next_permutation: the number of the automorphism below this one on the stack at level v.

start_index: the first white corresponds to *auto*$[p][start\_index]$ in the automorphism p.

end_index: *auto*$[p][start\_index] \ldots auto[p][end\_index-1]$ will have colours assigned when *auto*$[p][start\_index]$ is coloured but *auto*$[p][end\_index]$ will be the first white or *end_index* is equal to n if there will be no more whites.

cmp: -1 if the automorphism resulted in a lexicographically smaller colouring, 0 if it is equal so far, and $+1$ if larger.

One way to think of how the algorithm is working is that we are comparing the permuted colouring to the generated one in such a way that as we go along we are 'keeping a finger on the place where the comparison can continue' until a vertex is coloured that permits an extension to the comparison. As an example of how the *start_index* and *end_index* values are computed for a permutation, consider the automorphism p which is 03214765. The levels for which the colouring of the vertex numbered *level* increases the length of the longest prefix of red and blue vertices are 0, 3, 4 and 7 as follows:

Vertex:	0	1	2	3	4	5	6	7
Permutation:	0	3	2	1	4	7	6	5
Level where colours are compared:	0	3	3	3	4	7	7	7

At level 0 the colouring of vertex 0 is compared to the colouring of vertex 0 (a trivial subcase). Then at level 3, the colours of vertices 1, 2, and 3 are compared to those of 3, 2, and 1 which is how these are relabelled by the automorphism. Then at level 4, the colour of 4 is compared to the colour of 4 (another trivial sub-case). Then at level 7 the colours of vertices 5, 6, and 7 are compared to those of 7, 6, and 5.

A loop like this is used to extend the comparison of the colouring:

```
for i = start_index to end_index -1
        compare the colour of vertex i to the colour of π_i
```

The corresponding values of for the *start_index* and the *end_index* for a comparison which uses a loop like this are given in Table 1. For levels v not in Table 1, the values of the *start_index* and *end_index* of *stack_cell*$[v][p]$ are not defined.

Level	*start_index*	*end_index*
0	0	1
3	1	4
4	4	5
7	5	8

Table 1: The *start_index* and *end_index* values for the permutation 03214765.

The values of *start_index* and *end_index* for each record can be precomputed before the algorithm begins. For *stack_cell*$[v][p]$, first define k so that *auto*$[p][k] = v$. The *start_index* of *stack_cell*$[v][p]$, is undefined unless all of *auto*$[p][0] \ldots$ *auto*$[p][k-1]$ are less than v. Otherwise, *start_index* is equal to k. The value of *end_index* is undefined when *start_index* is undefined. Otherwise it is the minimum value $w >$ *start_index* such that *auto*$[p][w] > v$ or n if there is no such value. Choosing n as the default in this case means that the automorphisms of the generated independent sets are available without extra cost; they will be exactly those on the stack at level n. As noted earlier, this permits computation of the total number of independent sets corresponding to each isomorphism class. As a result, the total number of independent sets can also be computed as can any properties defined over all independent sets that are independent of how the vertices are labelled. The high level pseudocode for the algorithm is given in Figure 5.

Note that the total amount of space we request when preallocating these stack cells in advance is proportional to the amount of space required to store the permutations in the group and hence it is not excessive. Also, preallocating permits precomputation of *start_index* and *end_index* for all the stack cells.

One practical observation is that this algorithm is sensitive to how the vertices of the graph are labelled. A Breadth-First-Search ordering is desirable in order to permit faster rejection. This is discussed in more detail in Section 4.

It is not easy to provide a traditional Big Oh analysis of the time taken by this algorithm. The time depends on the structure of the graph and its automorphism group. Note however that for a path in the computation tree which goes from the initial recursive call to one which outputs an object,

```
ind_set(current_level)
current_level is the number of vertices coloured when invoked, initially 0.
g is the order of the automorphism group.
n is the number of vertices.
auto[0..g − 1][0..n − 1] is the set of automorphisms for the graph.
        If current_level is 0 then
                Initialize start_index and end_index for all stack_cell's.
                For j from 0 to n do
                        Set the stack for level j to be empty.
                For j from 0 to g − 1 do
                        Push stack_cell[auto[j][0]][j] on the level auto[j][0] stack.
        End if
        If current_level is n then
                Output or process the current independent set.
                Return.
        End if
        For c = red and then c = blue do
                Colour vertex current_level with colour c.
                Compute cmp for each stack_cell on the stack for current_level.
                If all cmp values are at least 0 then
                        For each automorphism j on the stack for current_level
                                with stack_cell[current_level][j].cmp equal to 0
                                        Let w = auto[j][stack_cell[current_level][j].end_index].
                                        Push stack_cell[w][j] on the stack for level w.
                        End For
                        ind_set(current_level+1)
                        For each automorphism j on the stack for this level
                                with stack_cell[current_level][j].cmp equal to 0
                                        Let w = auto[j][stack_cell[current_level][j].end_index].
                                        Pop the level w stack.
                        End For
                End If
                Colour vertex current_level white.
        End For
End ind_set
```

Figure 5: Pseudocode: generating all independent sets up to isomorphism.

the combined total of all the work that the calls has done is no more work in total than to examine each of the permutations of the group once from start to end due to the way that we 'keep a finger' on each permutation indicating where the comparison should continue when the permutation again becomes relevant. Traditional algorithms are more likely to be using the group at every recursive call.

If the aim is to generate just the maximal independent sets up to isomorphism, the algorithm can back up if there is a blue vertex whose neighbours are all blue. If the aim is to generate just the maximum independent sets, the algorithm can be enhanced by adding tests which can determine if a set cannot possibly extend to a maximum one. One example of a very simple test is that the total number of red vertices possible is the current number of red vertices plus the number of white vertices.

A more sophisticated upper bound for an r-regular graph can be defined in terms of the edges. An edge is *coloured* if both of its endpoints are blue or if it has a red endpoint (one endpoint is red and the other one is blue or forced to be blue). An upper bound on the total number of red vertices is the number of red so far plus the number of uncoloured edges divided by r since each added red vertex must map to r uncoloured edges. The algorithm can back up if this upper bound is less than the desired independent set order.

Other techniques such as colouring as done in [28] can be used for computing an upper bound. To ensure a fast algorithm, it is critical that these tests can be performed quickly.

3 All Independent Sets of Two Fullerenes

For any 3-connected planar graph, there is an easy well-known algorithm for computing the group of automorphisms. One place where it is used is in [10], but it is not clear where it originated. A *clockwise-BFS* is a special BFS that starts with a root node r and a first child c. The neighbours of the root node are processed in clockwise order starting with the first child c. The neighbours of the other vertices are traversed in clockwise order starting with the BFS parent. Each vertex is labelled with its breadth first index. Note that these restrictions give a unique BFS ordering of the vertices given a specific choice for r and c.

To obtain the automorphisms, apply a clockwise-BFS to an embedding and to the same embedding with a reversed sense of clockwise for each possible choice for r and c. Fix one of these labelled embeddings to be the basis for comparison. The renumberings which give the same labelled

k	$I_{60}(k)$	$I_{70}(k)$	k	$I_{60}(k)$	$I_{70}(k)$
1	1	5	16	264826099	135562358417
2	21	135	17	169008768	146606652792
3	257	2434	18	84588520	131720907835
4	3019	35132	19	32635885	97751862269
5	26333	380566	20	9482106	59489105820
6	180316	3228620	21	2001772	29426188579
7	967944	21854349	22	289893	11701128926
8	4158712	120185015	23	25712	3688070063
9	14406889	543279243	24	1085	904303709
10	40549092	2036505834	25		167994660
11	93003775	6367825587	26		22727292
12	174077292	16673558630	27		2097980
13	265534967	36635050437	28		117232
14	329091742	67585907694	29		2972
15	329601116	104608090007			

Table 2: Independent sets of C_{60} and C_{70}. $I_n(k)$ counts non-isomorphic independent sets of order k.

embedding as the fixed one give the automorphisms of the graph.

For fullerenes, it is judicious to start with some edge (r, c) which is part of a fixed pentagon P. Automorphisms can only result when some other pentagon Q is mapped to P and this limits to 120 the number of clockwise-BFS's which must be completed. Since the number of clockwise-BFS's is a constant, this very simple algorithm takes linear time for fullerenes.

The notation C_{60} refers to the 60-vertex fullerene whose structure is analogous to a soccer ball (it is the unique 60-vertex fullerene with all pentagons isolated). A picture is shown in Figure 1. It has automorphism group order 120. The number of independent sets up to isomorphism in the C_{60} fullerene is $1,814,461,317$.

We had attempted to generate all of these using another program, which used a parent/child generation scheme, but the algorithm appeared not to be close to completion even after about 6 weeks of run time. The new approach took just 28.4 minutes on a PC with an Intel P4 3.0 GHz processor. The numbers of independent sets of each order are given in Table 2.

To further show the applicability of this approach, the algorithm was also applied to the second most common experimental fullerene, C_{70}, which consists of two hemi-C_{60} caps, spaced by a belt of five disjoint hexagons. The C_{70} graph has automorphism group order 20. A picture of C_{70} is given

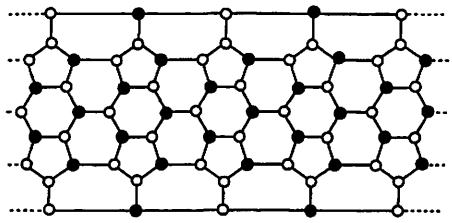

Figure 6: C_{70}. The black vertices are a maximum independent set.

in Figure 6. The black vertices are a maximum independent set. Two automorphisms of the graph preserve this independent set (in terms of this picture, the symmetry is a reflection through the horizontal axis).

Using a standard Big Oh analysis one could argue that the number of independent sets up to isomorphism for this graph is a constant. However, because this number is so large (851, 639, 422, 235), this is not an appropriate way to analyse the difficulty of the problem. For comparison, the maximum integer which can be stored in type **int** in C is 2,147,483,647 (so we were not able to use 32-bit integers to count these).

4 Finding a Maximum 4-Code for the 120-cell

As mentioned earlier, one problem that failed to fall to standard approaches was that of finding a maximum 4-code for the 120-cell (an independent set having the property that the independent set vertices are at distance four or more from each other). Our new algorithm together with some other tricks allowed us to find the answer to this problem.

It is easy to generate the list of permutations in the automorphism group of the 120-cell. The automorphism group has 14,400 elements, and goes by various names, such as the 'hecatonicosahedroidal' group. The graph is vertex symmetric and each selection of a vertex to be labelled as vertex 0 combined with one of the 24 orderings of its four neighbours defines an automorphism. As in the case of the planar graphs, it is also possible to define the rest of the isomorphism in terms of a modified BFS algorithm. The basis of the algorithm is that for each path a, b, c of the 120-cell, there is a unique pentagon that contains those three vertices.

Once the selection for vertex 0 and its 4 neighbours (1, 2, 3, and 4) has been made, the BFS proceeds as usual with the additional rule that when a neighbour u of vertex v is labelled, there is only one pentagon which

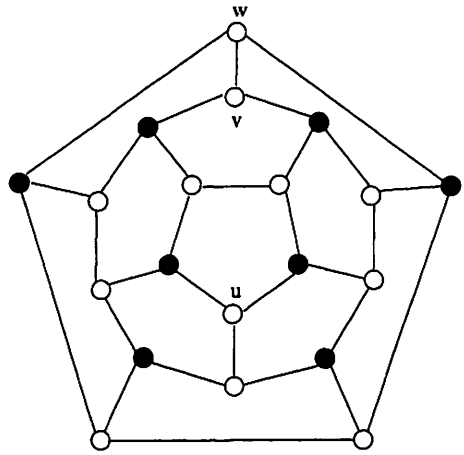

Figure 7: A dodecahedron, showing a maximum independent set (black vertices) and also the two pairs, up to isomorphism, of vertices at distances 4 or more, i.e., u, v and u, w.

contains u, v, and the BFS parent p of v and this pentagon also contains some vertex w adjacent to p. The labels of these w vertices are used to decide the order in which the children of v are labelled: sort according to these values to decide the order in which to process them.

The 120-cell has 120 induced subgraphs which are dodecahedra. Figure 7 depicts a dodecahedron with the vertices of a maximum independent set shown in gray. Up to isomorphism, there are only two ways to select a maximum independent set having the property that independent set vertices are at distance at least four from each other: select either vertices u and v in Figure 7 or vertices u and w.

One way to count the vertices of an independent set of the entire 120-cell is to count the independent set vertices of each dodecahedral cell, sum over all the dodecahedra, and finally divide by four as each vertex is in four dodecahedra. Applying this approach, if there existed a 4-code such that it was possible to choose exactly two vertices in every dodecahedron, then the number of vertices in the 4-code would be $120 \times 2/4 = 60$ because each vertex is in four dodecahedra. However, the largest 4-code we had been able to find with other methods had order 48. The objective was to determine if there is a 4-code of order 49 or more.

In order to get a better upper bound on the order of a maximum 4-code, we work with a penalty function (the definition follows) which is a measure

of how far short the dodecahedra are with respect to the ideal situation of having two 4-code vertices each. Mathematically, we define the *penalty* for each dodecahedron to be $2 - r$ where r is the number of 4-code vertices in the dodecahedron. The *total penalty* P for the 120-cell is defined to be equal to the sum of the penalties of the dodecahedra. The formula for computing the order of the 4-code in terms of the total penalty is $(120 \times 2 - P)/4$.

Various subgraphs of the 120-cell can be created by starting at a dodecahedron and adding on layers of dodecahedra out to a certain distance. The symmetries we considered for these subgraphs were the 120 automorphisms that fixed the initial dodecahedron. Consider such a subgraph which has δ dodecahedra. For each such subgraph of the 120-cell, compute the sum over all δ dodecahedra of the number of 4-code vertices in each dodecahedron. If the augmented 120-cell has a 4-code of order r, then the average x of these sums satisfies this equation:

$$x/\delta = (r \times 4)/120, \tag{1}$$

since every vertex of the 120-cell graph appears in four of the 120 dodecahedra. Since the average is x, at least one such sum is greater than or equal to x. The implication of this equation in terms of the penalties is that in order to have a 4-code of order r or more, the sum of the penalties in the δ dodecahedra is at most:

$$\delta \times 2 - x \tag{2}$$

for at least one of the subgraphs containing the δ dodecahedra.

We first tried applying the algorithm to a 270-vertex subgraph of the 120-cell centered at a dodecahedron and containing 33 dodecahedra. The penalty for an independent set of this 270-vertex subgraph is defined to be the sum over each of its 33 dodecahedra D of $2 - r_D$ where r_D is the number of 4-code vertices in dodecahedron D. If there is a 4-code of order 49 or more in the whole graph, then there must be some way to get it from augmenting a 4-code of this subgraph which has penalty at most 12. This is because in order to have a 4-code of order 49 or more, from Equation 1, $x/33 \geq (4 \times 49)/120$ for at least one of these subgraphs implying that $\lceil x \rceil \geq 54$. Hence from Equation 2 the sum of the penalties is at most $2 \times 33 - 54 = 12$.

We were discouraged to discover that the number of non-isomorphic 4-codes of penalty at most 12 is 7,045,230 and these ranged in size from 19-30 vertices. Initial timing estimates indicated it could take more than two years of computation time to test all these to check whether they extended to a 4-code of order 49 or more.

The next candidate subgraph to consider was the 330-vertex graph which is the skeleton of the partial model of the 120-cell given as an exam-

ple of constructions using Zome Models [14, Ch. 21]. This subgraph has 45 dodecahedra, arranged in concentric shells of 1, 12 and 32 face-sharing cells. The penalty for an independent set of this 330-vertex subgraph is defined to be the sum over each of its 45 dodecahedra D of $2 - r_D$ where r_D is the number of 4-code vertices in dodecahedron D. If there is a 4-code of order 49 or more then there is some way to get it from augmenting a 4-code of this subgraph with penalty at most 16. This is because Equation 1 gives that $x/45 \geq (4 \times 49)/120$ so $\lceil x \rceil \geq 74$. Hence from Equation 2, the sum of the penalties is at most $2 \times 45 - 74 = 16$.

In order to make elimination of those 4-codes for which the penalty was too high occur as soon as possible, the subgraph was relabelled so that the dodecahedra would be completed as soon as possible. Before this was done, the computation for finding all the independent sets of maximum penalty 16 appeared to be out of reach. This relabelling made a dramatic difference and enabled the generation of all independent sets with penalty at most 16 to be completed in under one day.

Only 3677 4-codes had penalty at most 16. The orders of these were 24, 25, 26, 27, 28, or 29 and there are respectively 2, 87, 1048, 2199, 334 and 7 at each of these orders. These were quickly tested in about 6.5 minutes to determine if any of them extended to a 4-code of order 49 or more. None did, which permits us to conclude that the maximum 4-code of the 120-cell has order 48.

A more standard approach to this problem is to argue that a selected subgraph must contain at least k independent set vertices for some integer k and then to plant these into the subgraph in all ways up to isomorphism. The reason that this is not as effective is that on the perimeter of the subgraph (vertices which have lower degree in the subgraph than in the original graph), it tends to be easy to plant lots of independent set vertices in such a way that it does not lead to a solution of the desired size since they interfere too much with vertices that might be added later. The penalty approach that described above forces independent set vertices to also be chosen from the part of the subgraph not on the perimeter (otherwise the penalty is too large).

5 Additional Applications

There are some graphs which have non-trivial symmetry groups for which finding a maximum clique is very difficult. One approach to these is to use the algorithm in this paper to generate all cliques in the graph up to isomorphism of some small order, say k, then to find for each of those the

maximum clique of the whole graph which contains the k-clique. Timing experiments are required to find the best break point k to stop the isomorphism testing. Initially, the isomorphism testing saves a lot of duplicated computation. But at some point, the amount of work done for isomorphism becomes slower than testing duplicated cases.

It is often possible to argue that if a graph has a clique of order k then there is some subgraph of the graph which is required to have some clique of order at least r for some r less than k. A natural approach for applying this concept is to choose a subgraph such that there is a rich subgroup of the group mapping it to itself, plant r-cliques in the subgraph in all ways up to isomorphism, and then try to extend these using a backtrack without worrying about isomorphism testing.

There are many other applications for this algorithm. The examples here used two colours for the vertices; red for independent set vertices and blue for the vertices not in an independent set. The rule for selection of the red vertices was that they should be independent. Other conditions could be chosen for allowable sets of red vertices; if there are no restrictions placed on them, the algorithm would just generate all subsets of the vertices up to isomorphism. Or they could be selected so that they give a dominating set. Also, an arbitrary number k of colours could have been used in order to generate all non-isomorphic k-colourings of a graph.

A *matching* of a graph is a collection of disjoint edges. Ordering the edges in a graph in a natural way and expressing the automorphisms in terms of how they map edges would yield an algorithm for generating all matchings up to isomorphism. In the mathematical chemistry literature, the total number of matchings of a graph is called the *Hosoya Index* [15] and is used for structure-property correlation in chemoinformatics.

A colouring of the vertices of a graph is a *distinguishing colouring* if there are no automorphisms of the graph that preserve the colours. Unlike the traditional approach to vertex colouring, there is no requirement that adjacent vertices receive different colours. The *distinguishing number* of a graph is the minimum number of colours required for a distinguishing colouring. This algorithm is well-suited for generating all distinguishing k-colourings because it provides along with each colouring generated, the automorphisms that preserve the colouring. The colouring is distinguishing if there is exactly one automorphism associated with the colouring (the identity). One interesting open question is whether all graphs that have a k-distinguishing colouring also have a k-list colouring that is distinguishing for every assignment of lists of size k.

The approach could also easily be applied to other combinatorial ob-

jects, one example being the construction of combinatorial designs up to automorphism (problems of this type can sometimes be expressed in terms of finding independent sets or cliques of a graph).

References

[1] D. Avis and K. Fukuda, Reverse search for enumeration, *Discrete Appl. Math.* 65 (1996), 21–46.

[2] C. Bron and J. Kerbosch, Algorithm 457: finding all cliques of an undirected graph, *ACM Comm.* 16 (1973), 575–577.

[3] S. Daugherty, W. Myrvold and P. W. Fowler, Backtracking to compute the closed-shell independence number of a fullerene, *MATCH Commun. Math. Comput. Chem.* 58 (2007), 385–401.

[4] S. Debroni, E. Delisle, W. Myrvold, A. Sethi, J. Whitney, J. Woodcock, P. W. Fowler, B. de La Vaissière and M. Deza, Maximum independent sets of the 120-cell and other regular polytopes. *Ars Math. Contemp.* 6 (2013) 197–210.

[5] J. Debroni, J. D. Eblen, M. A. Langston, W. Myrvold, P. Shor, and D. Weerapurage, A complete resolution of the Keller maximum clique problem, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 11)*, pages 129–135, 2011.

[6] B. de La Vaissière, P. W. Fowler and M. Deza, Codes in Platonic, Archimedean, Catalan and related polyhedra: a model for maximum addition patterns in chemical cages, *J. Chem. Inf. Comput. Sci.* 41 (2001) 376–386.

[7] M. Dutour Sikirić and W. Myrvold, The special cuts of the 600-cell, *Beiträge Algebra Geom.* 49 (2008), 269–275.

[8] P. W. Fowler, B. de La Vaissière and M. Deza, Addition patterns, codes and contact graphs for fullerene derivatives, *J. Mol. Graphics Model.*, 19 (2001), 199–204.

[9] P. W. Fowler, P. Hansen, K. M. Rogers and S. Fajtlowicz, $C_{60}Br_{24}$: A chemical illustration of graph theoretical independence, *J. Chem. Soc. Perkin Trans. 2* (1998), 1531–1533.

[10] P. W. Fowler, D. Horspool and W. Myrvold, Vertex spirals in fullerenes and their implications for nomenclature of fullerene derivatives, *Chem. Eur. J.* 13 (2007), 2208–2217.

[11] P. W. Fowler, K. M. Rogers, K. R. Somers and A. Troisi, Independent sets and the prediction of addition patterns for higher fullerenes, *J. Chem. Soc. Perkin Trans. 2*, (1999), 2023–2027.

[12] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Co., San Francisco, CA, 1979.

[13] A. Gély, L. Nourine and B. Sadi, Enumeration aspects of maximal cliques and bicliques, *Discrete Appl. Math.* 157 (2009), 1447–1459.

[14] G. W. Hart and H. Picciotto, *Zome Geometry: Hands-on Learning with Zome Models*, Key Curriculum Press, Emeryville, CA, 2000.

[15] H. Hosoya, Topological index. A newly proposed quantity characterizing the topological nature of structural isomers of saturated hydrocarbons, *Bull. Chem. Soc. Japan* 44 (1971), 2332–2339.

[16] D. S. Johnson and M. A. Trick (eds.) *Cliques, Colouring, and Satisfiability, Second DIMACS Implementation Challenge, Oct. 11-13, 1993*, Volume 26 in DIMACS Ser. Discrete Math. Theoret. Comput. Sci. AMS, Providence, RI, 1996.

[17] D. S. Johnson, M. Yannakakis and C. H. Papadimitriou, On generating all maximal independent sets, *Inform. Process. Lett.* 27 (1988), 119–123.

[18] H. C. Johnston, Cliques of a graph - variations on the Bron-Kerbosch algorithm, *Int. J. Comput. Inform. Sci.* 5 (1976), 209–238.

[19] O. H. Keller, Über die lückenlose Einfüllung des Raumes mit Würfeln, *J. Reine Angew. Math.* 163 (1930), 231–248.

[20] J. C. Lagarias and P. W. Shor, Keller's cube-tiling conjecture is false in high dimensions, *Bull. Amer. Math. Soc. (N.S.)* 27 (1992), 279–283.

[21] E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, Generating all maximal independent sets: NP-hardness and polynomial-time algorithms, *SIAM J. Comput.* 9 (1980), 558–565.

[22] E. Loukakis, A new backtracking algorithm for generating the family of maximal independent sets of a graph, *Comput. Math. Appl.* 9 (1983), 583–589.

[23] J. Mackey, A Cube Tiling of Dimension Eight with No Facesharing, *Discrete Comput. Geom.* 28 (2002), 275–279.

[24] K. Makino and T. Uno, New algorithms for enumerating all maximal cliques, *Lecture Notes Comput. Sci. (from SWAT 2004)*, 3111 (2004), 260–272.

[25] B. D. McKay, Isomorph-free exhaustive generation, *J. Algorithms* 26 (1998), 306–324.

[26] R. E. Merrifield and H. E. Simmons, Enumeration of structure-sensitive graphical subsets: Theory, *Proc. Nat. Acad. Sci. USA* 78 (1981), 692–695.

[27] R. E. Merrifield and H. E. Simmons, Enumeration of structure-sensitive graphical subsets: Calculations, *Proc. Nat. Acad. Sci. USA* 78 (1981), 1329–1332.

[28] W. Myrvold, T. Prsa and N. Walker, A Dynamic Programming Approach for Timing and Designing Clique Algorithms, in: S. Fajtlowicz, P.W. Fowler, P. Hansen, M. Janowitz and F.S. Roberts (eds.), *DIMACS Ser. Discrete Theoret. Comput. Sci.: Graphs and Discovery*, (69), AMS, Providence, RI, 2005, 333–340.

[29] M. C. Schmidt, N. F. Samatova, K. Thomas and B.-H. Park, A scalable, parallel algorithm for maximal clique enumeration, *J. Parallel Distrib. Comput.* 69 (2009), 417–428.

[30] J. Stillwell. The story of the 120-cell. *Notices Amer. Math. Soc.*, 48 (2001), 17–24.

[31] E. Tomita, A. Tanaka and H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, *Theoret. Comput. Sci.* 363 (2006), 28–42.

[32] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, A new algorithm for generating all the maximal independent sets, *SIAM J. Comput.* 6 (1977), 505–517.