Exploring the use of constraint programming for enforcing connectivity during graph generation

Kenneth N. Brown[1], Patrick Prosser[2], J. Christopher Beck[3] and Christine Wei Wu[1]

[1]Cork Constraint Computation Centre, Department of Computer Science,
University College Cork, Ireland.
{*k.brown,cww1*}*@cs.ucc.ie*
[2]Department of Computer Science, University of Glasgow, Scotland
pat@dcs.gla.ac.uk
[3]Toronto Intelligent Decision Engineering Laboratory,
Department of Mechanical and Industrial Engineering, University of Toronto, Canada.
jcb@mie.utoronto.ca

Abstract

We discuss the problem of using constraint models to force generated graphs to be connected. We represent the graph as a simple adjacency matrix, and then attempt to post constraints ensuring connectivity. Doing this using standard modelling primitives is harder than expected, because of a problem with our use of the implication operator. We develop a global constraint *connected-graph*, and show that it does save time over a class of graph generation problems, but most of the gains come from simple pre-search filters applied to insoluble instances. We finish by discussing a new constraint, *graphical*, which simply ensures that a partially instantiated graph can be completed.

1 Introduction

We consider the problem of enforcing connectivity while generating graphs, a problem which appears embedded within many practical applications. For example, in computational chemistry, generating all possible single molecules formed from a set of atoms involves generating connected multigraphs (where atoms are represented by vertices, valencies are represented by the degrees, and bonds are represented by edges [Wu, 2004]). In telecommunications network planning, base stations and hubs are vertices, the communication links are edges, and the network must clearly be connected. From Operations Research, solving the Travelling Salesperson Problem involves constructing a minimal length path which visits every node: i.e. a connected graph where every vertex has degree 2. Many such problems come with associated side constraints - for example, legal bonds between atoms or acceptable delays on communications links - and thus a constraint programming solution may be desirable. In this paper, we attempt to develop a constraint model which ensures that generated graphs are connected. There has been some previous work on reasoning about properties of graphs using constraint models. For example, [Le Pape *et al.*, 2002] present a new variable type representing paths in a graph, [Sorlin and Solnon, 2004] discusses a global constraint for graph isomorphism problems, and [Pesant and Soriano, 2002] generate optimal cycle covers for networks. Graph theoretic algorithms have been used extensively in constraint programming; see [Régin and Gomes, 2004], for example, or [Simonis, 2004] for a survey.

We have investigated a pure version of the problem, constructing undirected connected simple graphs with no self loops over a set of vertices with known degrees. First, we represent the undirected simple graphs with no self loops. We assume that we are given an empty adjacency matrix and a degree sequence, and that the search process is free to select or reject any edge. Our goal is then to construct constraint-based models which will allow us to enforce connectivity while searching for all graphs that realise a given sequence.

We start by presenting a constraint encoding for producing the basic graphs with the specified degree sequence. We then consider how to enforce connectivity on these graphs using standard constraints, but run into problems. We then present *connected-graph*, a global constraint for maintaining connectivity, and describe a first implementation based on a 'connected-components' algorithm. We then extend that implementation to include some limited propagation. We present some results indicating that the constraint is effective when searching for all solutions to a set of degree sequences, but that most of the efficiency gains come from sequences which have no connected realisation. We then conclude by looking at a more limited constraint which simply enforces graphicality - that is, it ensures that a partially instantiated graph can in fact be fully realised as a graph.

2 Graph Theory Preliminaries

We assume the necessary constraint background, and concentrate here on introducing the graph theory terminology and

basic graph-theoretic results. See [Gould, 1988] for a more comprehensive (and readable) account.

A *graph* G is a finite set $V = \{1, 2, \ldots, n\}$ of *vertices*, and a collection $E = \langle\{v_1, w_1\}, \ldots, \{v_m, w_m\}\rangle$ of *edges*, where each $v_i, w_i \in V$. If the edges in E are ordered pairs rather than sets, then E is a *directed* graph. If the list E is a set, then G is a *simple* graph; if E contains multiple copies of an edge, then G is a *multigraph*. If $\{v_i, v_i\} \in E$, then it is a *self-loop*. From now on, we will assume that E is an undirected simple graph with no self loops.

If $\{v_i, v_j\} \in E$, then v_i and v_j are *adjacent*, and the edge $\{v_i, v_j\}$ is *incident* on both v_i and v_j. Two vertices v_1 and v_k are *connected* if there is a *path* $P = \langle v_1, v_2 \ldots, v_k\rangle$ of vertices, such that $\forall i < k,\ \{v_i, v_{i+1}\} \in E$. A graph is *connected* if $\forall i, j,\ v_i$ and v_j are connected. A *connected component*, C_i, is a set of vertices such that $v_j \in C_i \leftrightarrow \forall v_k \in C_i, v_j$ and v_k are connected. A connected graph has exactly one connected component. A connected graph must have at least $n-1$ edges. A graph has at most $n * (n-1)/2$ edges.

The *degree* of a vertex is the number of edges incident on it. By the *handshaking lemma*, the sum of the degrees of a graph is twice the number of edges, and so, as a corollary, the sum of the degrees of the vertices must be even. Let $degree(i)$ be the degree of vertex i. Then $D = \langle degree(1), \ldots degree(n)\rangle$ is the *degree sequence* of G. We can now pose the question: given V (a set of n vertices), and D (a sequence of n integers), does there exist a graph G that *realises* D? That is, can we construct an edge set E such that D is the degree sequence of $G = (V, E)$? Further, can we ensure that G is connected, and can we generate all connected realisations?

3 Representing a simple graph with a given degree sequence

Representing a simple graph with a given degree sequence [Shiloach, 1981] is straightforward. For a problem with n vertices, we create an $n \times n$ array, A, of constrained $0/1$ variables. When $A[i, j] == 1$, there is an edge from i to j, and when $A[i, j] == 0$, there is no edge from i to j. Since our graphs are undirected, we post the constraint $A[i, j] == A[j, i]$ for each pair $i < j$. To stop self-loops we add the constraint $A[i, i] == 0$ for each i. If we then assume that D is an array of integers, such that $D[i]$ is the degree of vertex i, then we can post a constraint for each row i of A to ensure that the vertices have the correct degree: $\sum_{j=1}^{n} A[i, j] == deg[i]$.

We can now ask the solver to generate all solutions. For a degree sequence $\langle 2, 2, 2, 1, 1\rangle$, we will get 7 solutions: 6 of them will be paths, and the 7th will consist of two components, one with a pair of connected vertices (K_2), and the other a triangle (K_3). Obviously, the first 6 are isomorphic to one another, and the 7th is disconnected. For a degree sequence $\langle 2, 2, 1, 1, 1\rangle$, there are no solutions, since the sum of the degree sequence is odd (from the handshaking lemma). Our model, however, requires search to discover this. Therefore there are two extensions required: we must detect choices which would lead to a disconnected graph, and need to identify degree sequences which cannot be realised.

4 Enforcing connectivity: a first attempt

Since connectivity is defined in terms of paths, we first considered enforcing connectivity by introducing n^2 *path* variables $P[i, j]$, such that $P[i, j] = 1$ if there is a path from vertex i to vertex j. The path variables are not intended to be decision variables, but will be linked to the adjacency matrix. Each time we set two vertices to be adjacent (i.e. set $A[i, j] = 1$), we also set $P[i, j] = 1$, and propagate recursively to other path variables. Therefore, we add constraints $A[i, j] == 1 \rightarrow P[i, j] == 1$ (for each pair i and j), and $(P[i, j] == 1 \wedge P[j, k] == 1) \rightarrow P[i, k] == 1$ (for all triples i, j and k). We then add a constraint forcing every vertex to be connected to vertex 1: $P[1, j] == 1$ for all j.

But this doesn't work, and it doesn't work because we have relied on implication. $P \rightarrow Q$ is true when Q is true and P is false, so our encoding allows a solver to cheat by setting $P[i, j] = 1$ whenever it needs to, and thus our path-connectivity constraint is trivially satisfied.

5 Enforcing connectivity: a second attempt

There are two standard algorithms for checking whether or not a graph is connected ([Cormen *et al.*, 2001]): depth-first search, and CONNECTED-COMPONENTS(G). This second algorithm maintains data structures for the connected subcomponents of the graph, and its outline is sketched below:

```
Connected-Components(G)
1. for each vertex v in V(G)
2.    MakeSet(v)
3. for each edge (v,u) in E(G)
4.    if FindSet(u) != FindSet(v)
5.       Union(u,v)
```

The algorithm starts by producing an individual set for each vertex in the graph, such that each set contains exactly one vertex. We then iterate over the edges of the graph, combining pairs of sets if they span an edge. On termination, the sets represent the components of the graph: if there is only one component, then the graph is connected, and otherwise it is disconnected. In line 2, MakeSet(v) creates a new set containing vertex v. In line 4, FindSet(u) returns the set that contains u, and in line 5, Union(u,v) unions the sets that contain u and v. Let n be the number of vertices, and e the number of edges. We assume that Union(u,v) takes $O(|v|)$ operations, and that FindSet and MakeSet take $O(1)$. For a connected graph, $n - 1$ of the edges require an application of Union (to establish that each of the remaining vertices connects to the first). In the worst case, the algorithm always applies Union(u,v) when u is a singleton set, and $|v|$ steps from 1 to $n-1$, and thus requires $1+2+\ldots+(n-1) = (n-1)n/2$ operations, plus n operations for the initial sets. Thus the worst case running time is $O(n^2)$. The space required is $O(n)$ (for initially n singleton sets, and finally 1 set of n elements).

We attempted to construct a declarative constraint encoding of this algorithm using the set variables provided in Choco. We introduced n set variables $S[i]$, where $S[i]$ is initialised with the value $\{i\}$. Then, when search selects the edge (i, j), we want to combine sets $S[i]$ and $S[j]$. So we added the following constraint:

$$\forall i\ \forall j\ A[i, j] = 1 \rightarrow ((S[i] \subseteq S[j]) \wedge (S[j] \subseteq S[i]))$$

But now we are back to the implication problem. We could change $A[i,j]$ above to $P[i,j]$, and turn the implication into an 'if and only if', but we then have the same problem as before linking $P[i,j]$ back to the adjacency matrix[1]. The problem is because we are introducing auxiliary variables, but only putting them on the right hand side of an implication, and thus setting the value of an auxiliary variable only partially constrains the decision variables. In terms of the implication operator, the auxiliary variables also need to appear on the left hand side of an implication, with a decision variable on the right (or on the right of a chain of implications).

6 Enforcing connectivity: the *connected-graph* global constraint

Instead of continuing to try different modelling primitives[2], we decided to implement a global constraint, which uses the CONNECTED-COMPONENTS(G) algorithm to update its internal data structures (the components) after each value assignment. The constraint takes the adjacency matrix, A, and the degree sequence, D, as input. It does no propagation, but will be violated if all the variables in A are instantiated and there is more than one component remaining. It requires three reversible data structures (reversible so that their values can be restored when the search process backtracks). C is a list of components, and each component is a list of integers representing vertices. P is an array maintaining for each vertex the index of its component in C. c is the number of components. To initialise, we create a unique component for each vertex. Whenever the search process assigns the value 1 to $A[i,j]$ (i.e selects the edge between i and j), where i and j were in different components, we update the data structures. We take the smaller of the two components, move all of its vertices into the larger, and update P for each of those vertices to point to the new component. Finally, we decrement c. When we reach a leaf node, if $c == 1$ then the graph is connected; if $c > 1$, then the graph is disconnected.

The data structures require $O(n)$ space, to store each vertex in a component, and to store the names of the components. If we assume that we always merge the shorter component into the larger, the updating requires at most $n/2$ operations to merge two components, and $n/2$ operations to update P, and thus is $O(n)$ at each node of the search tree. However, on a complete branch from root to leaf, we require $n-1$ updates, and thus $O(n^2)$ operations. This is the same cost as it would be to run the CONNECTED-COMPONENTS(G) algorithm afresh at each leaf node. In addition, however, we have the cost of updating the data structures on the branches that fail because of other constraints. Therefore, if this constraint is to be effective, we need to extend it by pruning or by detection of search nodes which have no connected realisations below them in order to save enough operations to account for the overhead.

[1] and we also found that Choco wouldn't let us do it anyway, reporting that the opposite of $\subseteq$ was not defined.

[2] although we have one more model, suggested by Ian Miguel, which we have not yet tried.

7 Adding propagation to *connected-graph*

We can improve the constraint by reasoning about the residual degrees of vertices and components during search, and by including some of the basic graph theory results. During a search, if vertex i has had k of its possible edges instantiated, then its *residual degree* is $degree(i) - k$. Let the residual degree of a component be the sum of the residual degrees of its vertices. The residual degree of a partially instantiated graph is the sum of the residual degrees over all vertices. To maintain information on the residual degrees, we need the following additional reversible data structures:

- an array RV of integers, maintaining the residual degree of each vertex. Each time we instantiate $A[i,j]$ to 1, we subtract 1 from $RV[i]$ and $RV[j]$.
- an integer r, maintaining the residual degree of the partial graph. Each time we instantiate any edge variable to 1, we subtract 2 from r (since each edge reduces two individual residual degrees by 1 each).
- an array RC of integers, maintaining the residual degree of each component. Each time we instantiate $A[i,j]$ to 1, we find the components p and q of i and j respectively using the array P. If they are the same component (i.e. $p == q$), then we subtract 2 from $RC[p]$; if they are different components, then we will merge them as before. Let p' be the merged component. We then set $RC[p'] = RC[p] + RC[q] - 2$.

We can identify a number of cases in which violations can be identified on initialisation:

1. if any vertex has initial degree of less than 1, and there is more than 1 vertex, then no connected graph is possible, since that vertex must be isolated;
2. if any vertex has an initial degree of more than $n-1$, then no graph is possible, since there are not enough other vertices with which to create the edges;
3. if the sum of the initial degrees is odd, then no graph is possible, by the handshaking lemma;
4. if the sum of the initial degrees is less than $2n-2$, then no connected graph is possible, since there are not enough edges to connect all the vertices;
5. if the sum of the initial degrees is greater than $n*(n-1)$, then no graph is possible, since there are not enough vertices to occupy all the edges;

We can also identify two cases for intermediate search nodes where the constraint must be violated, based on residual degree:

6. if the residual degree of a component drops to 0, and there is more than one component, then no completion of the partial graph can be connected, since all vertices in the component have used up all the edges, and none of those edges connect to the second component (by the definition of a component), then the first component can never become connected to the second;
7. if the residual degree of the graph drops to less than $2c-2$, where c is the number of components, then no

completion of the partial graph can be connected. This is by analogy to 4, in which we replace vertices by components - in order to ensure one component is connected to all the others, we will need to use at least one edge per remaining component (i.e. $c-1$ edges). Each edge contributes 2 to the residual degree, and therefore we need at least $2c-2$ edges to get a connected graph.

Finally, based on these violation checks, we can develop the following propagations:

8. if $n > 2$, then for all pairs of vertices i and j with initial degree of 1, force i and j to be not adjacent (since if we connect two vertices with degree of 1, then they must form an isolated component, and cannot be connected into a larger graph).

9. if $C[i]$ is a component with residual degree of 2, and there is more than one component, then if there is a pair j and k in $C[i]$ each with residual degree of 1, force j and k to be non-adjacent (if there is such a pair, then if we were to connect them together, there would be no more edges able to be instantiated incident on $C[i]$, and so $C[i]$ could not be connected to the rest of the graph). We apply this when $RC[i]$ is reduced to 2.

10. if $C[i]$ and $C[j]$ are two components with residual degree of 1, and there are more than two components, for the vertices v in $C[i]$ and w in $C[j]$ with residual degree of 1, force v and w to be non-adjacent (since each component must have exactly one vertex with residual degree > 0, and if we connect them, then the new combined component would have residual degree of 0, and so could not be connected to the rest of the graph). We apply this when $RC[i]$ is reduced to 1.

11. if the residual degree of the graph is $2c-2$, and there is more than one component, then for all components with residual degree greater than 1, force all pairs of vertices internal to the component to be non-adjacent (by the same analogy to 7, we need at least $c-1$ edges to connect up the components, and hence residual degree at least $2c-2$, so if we connect two vertices that are already in the same component, then we will not have enough edges remaining to connect up the other components). We apply this when r is first reduced to $2c-2$.

Propagation 8 is carried out at initialisation; the rest are carried out at nodes of the search tree.

The space requirement is still $O(n)$. The updates to the data structures are $O(n)$ as before, since the new updates each require only $O(1)$. The initialisation takes $O(n^2)$, because of 8. For propagation 9 we require at each search node at most n checks to find 2 vertices. For propagation 10, we require at most n checks to find both vertex v and all other vertices representing w. For propagation 11 there are at most $(n-1)^2$ pairs, and thus we require $O(n^2)$ checks. All four propagations only force values of 0, but are only triggered by variables being set to 1, and thus there is no cycle of propagators (although they may be invoked again if the other constraints set a variable to 1).

8 Experiments on *connected-graph*

We have implemented the adjacency matrix and the global *connected-graph* constraint in Ilog Solver 6.0. Each of the dead-end checks and propagations can be switched on or off independently. Recall that our purpose is not to generate all connected graphs as quickly as possible, but to develop a constraint that can be used with an external search procedure on problems with side constraints, to enforce connectivity. In particular, we have not considered symmetry, and there are many symmetries in these problems. We view symmetry as a separate feature, to be maintained independently from connectivity, and in other work we have begun to detect symmetries during the search [Wu, 2004]. However, we do want to evaluate the effectiveness of our model, and so we have tested it on pure connected graph generation problems. We have generated all possible degree sequences of lengths ranging from 6 to 10, with maximum vertex degree of 4. For each of these sequences, we then search for all possible solutions, and we have recorded for each length the total number of solutions (i.e. connected graphical realisations), the total number of backtracks-on-failure, and the total running time. We have run the algorithm with full propagation (*all*), with propagation 11 turned off (*-11*), with propagation only in the initialisation phase (*init*), with the odd degree initialisation filter and leaf node violations checks only (*even*), and with only checks at the leaf nodes (*leaf*) (i.e. no propagation and no other violation checks). We use the variable ordering heuristic `IloChooseMinSizeInt` (minimum domain), and a lexicographic value ordering. The experiments were carried out under Linux, with a 2.6 MHz processor. The results are presented in table 1.

Running with the leaf node violation checks only (*leaf*) is significantly slower than the four other methods which use some degree of filtering. However, we note that most of the improvement in running time for the other methods comes from *even*, the simple initialisation filter which fails sequences with an odd sum (which cannot have graphical realisations). The search with full propagation, *all*, is reducing the backtracks on failure by up to 10% compared to *even*, but is not significantly faster - in fact, for some of the smaller n, it is slightly slower. There could be a number of reasons for this. It is possible that our implementation is inefficient. Secondly, our propagations are relatively shallow - that is, they remove values which are likely to have been discovered at the next one or two depths in the tree, and so much of the work may be wasted. Finally, in this paper, we have only reasoned about residual degrees. We have not yet considered the consequences of setting an edge variable to 0 (i.e. rejecting the edge from the graph). We expect to be able to do more reasoning about the absence of edges to discover that subcomponents cannot be connected. However, even if we do improve our algorithms, when we compare the total fails for *even* with the total number of solutions, it appears that there is simply not that much propagation to be done - once we filter out those sequences of odd degree, only approximately 15% of the leaf nodes in the full search tree are not connected.

n	#	solutions		fails	time
6	84	703	all	193	0.14
			-11	193	0.14
			init	219	0.15
			even	259	0.13
			leaf	1243	0.18
7	120	10544	all	1811	0.37
			-11	1817	0.43
			init	2112	0.38
			even	2303	0.40
			leaf	18449	0.58
8	165	249569	all	38538	4.56
			-11	38604	4.48
			init	42512	4.46
			even	44010	4.51
			leaf	379152	8.25
9	220	7742661	all	1169783	127.19
			-11	1170429	127.08
			init	1230572	126.70
			even	1242061	127.03
			leaf	11764916	241.37
10	286	345052878	all	51550046	5717.91
			-11	51558645	5750.86
			init	52780580	5731.06
			even	52916767	5733.59
			leaf	478361894	10420.87

Table 1: finding all solutions for all degree sequences: n is the length of the sequence, # is the number of sequences of that length, *solutions* is the number of connected realisations, *fails* is the number of backtracks-on-failure for each method, and *time* is the total time in seconds for each method. Note that *solutions*, *fails* and *time* are the aggregated results over all sequences of the indicated length.

9 The Erdös-Gallai theorem

Since the solution density is so high in realisable sequences, it appears that graphicality may be more significant than connectivity, and so cheaper propagation to cut out non-graphical sequences, followed by leaf node checks on connectivity, might improve efficiency. We have therefore begun to investigate specific graphicality properties. The Erdös-Gallai theorem [Erdös and Gallai, 1960] states when a given degree sequence is *graphical*, i.e. under what conditions a graph can be produced with a given degree sequence. The theorem states that given a degree sequence $\sigma = d_1 \geq d_2 \geq ... \geq d_n$, this is graphical if and only if equation (1) holds for all $k < n$.

$$\sum_{i=1}^{k} d_i \;\leq\; k(k-1) + \sum_{i=k+1}^{n} min(k, d_i) \qquad (1)$$

This leads to the Havel-Hakimi algorithm [Havel, 1955; Hakimi, 1962] for a *realisation* of that sequence. We reproduce it below, in a version taken from [Gould, 1988], and it tests if the degree sequence σ is graphical.

1 If there exists an integer d in σ such that $d > n - 1$ then halt and report failure. That is, we cannot have a vertex that is adjacent to more than $n - 1$ other vertices.

2 If there are an odd number of odd numbers in σ halt and report failure. That is, there must be an even number of vertices of odd degree.

3 If the sequence σ contains a negative number then halt and report failure.

4 If the sequence σ is all zeros then halt and report success.

5 Reorder σ such that it is non-increasing.

6 Delete the first term d_1 from σ and subtract one from the next d_1 terms to form a new sequence. Go to step 3

Note that sequence $\sigma = 0, 0, 0, 0$ is graphical and realisable, and so there is nothing in the theorem or algorithm that states that the graph must be connected.

10 Using the Havel-Hakimi algorithm as a constraint

In the generation of graphs, edges are selected or rejected by the search process. Could the selection or rejection of an edge result in a dead end because equation (1) is violated? The answer is yes, and the (existence) proof follows.

Proof: Assume we have a degree sequence $S = 2, 1, 1$. This is graphical and can be realised as the path graph. Assume also that vertex v_1 has been constrained to have a degree of 2, and vertices v_2 and v_3 are to have a degree of 1. Further assume that the search process starts by selecting the edge (v_2, v_3). In the residual graph v_1 must have degree 2 and vertices v_2 and v_3 have a residual degree of 0. This is not a graphical sequence. □

Therefore we should expect that our search process can generate dead-ends because equation (1) is violated, and early detection of this may result in reduced search effort. Note that step 2 of the above algorithm is redundant within the constraint encoding, i.e. if at the top of search the sequence σ contains an even number of odd numbers this will continue to be true during search. We prove this by considering 3 cases.

Proof: In case (1) search selects an edge (v, u) where the residual degree of v and u is even. When we add the edge we decrement the residual degrees and we now have two more vertices of odd degree. (2) search selects (v, u) and both vertices have an odd residual degree. When the edge is added we decrement the residual degrees and we remove two vertices of odd degree. (3) v has odd residual degree and u has even residual degree or conversely v has even residual degree and u has odd residual degree. We decrement both residual degrees and have the same number of vertices of even and of odd residual degree. Consequently such a constraint would serve no purpose. □

It is worth noting that in [Mihail and Vishnoi, 2002] it is claimed that for a sequence to be graphical and potentially connected it is necessary and sufficient that (1) holds and that the sum of the degrees is at least $2(n - 1)$, i.e. there are at least enough edges to produce a spanning tree. However, no algorithm is given for realising this other than to produce a spanning tree and then use the Havel-Hakimi algorithm on the residual graph.

n	nGSeq	nCSeq	Sol	nodes-	nodes+
4	11	6	9	6	9
5	31	19	61	62	62
6	102	68	787	1018	1017
7	342	236	15384	21329	21286
8	1213	863	580950	843812	841574

Table 2: Effect of the *graphical* constraint: *n* is the length of the sequence (with no maximum degree), *nGSeq* is the number of graphical sequences, *nCSeq* is the number of those that had connected realisations, *Sol* is the total number of connected realisations, *nodes-* is the number of nodes generated during the search without the *graphical* constraint, and *nodes+* is the number of nodes generated using the constraint.

We have coded up the Havel-Hakimi algorithm as a constraint $graphical$. The constraint takes as arguments an adjacency matrix of 0/1 constrained variables and a degree sequence. The constraint is tested whenever an edge is selected or rejected, i.e. a test is performed to determine if the residual graph continues to be graphical. If the residual graph is not graphical a contradiction is raised and a backtrack is forced. We have tested the effect of the constraint, and the resuls are shown in Table 2. Note that the results in this table are not comparable to the results in Table 1, since we have used a different experimental set-up. In particular, the sequences now have no maximum degree, only graphical initial sequences were considered, and a different connectivity filter was applied during search. From the results, we can see that the $graphical$ constraint does propagate, even in the search for realisations of sequences that are initially graphical, and that the effect does increase as we move to larger problems.

11 Conclusion and Future work

We have described some explorations of constraint programming in graph generation, concentrating on forcing the graphs to be connected. We have presented a straightforward encoding of the simple graph generation problem. We have briefly described two failed attempts to model the connectivity constraint using standard modelling primitives. We have discussed a new global constraint, *connected-graph*, and developed some violation checks and propagations. We have tested the constraint on some pure generation problems, and we have shown that the constraint does reduce search time. However, perhaps because of the very high solution density, almost all of the efficiency gains come from a filter rejecting non-graphical degree sequences, rather than from reasoning about connectivity. We have then discussed an alternative constraint, *graphical*, which simply enforces graphicality.

As stated in the introduction, the problem of enforcing connectivity during graph generation has practical applications, and we are continuing to develop constraint-based solutions for those applications. Although there does not appear to be much scope for propagation in the pure problem, based on the ratio of the number of solutions to the total leaf nodes, we intend to develop better propagators based on reasoning about the rejection of an edge, and on other methods based on graph theory. Our first step will be to unify the different methods discussed in this paper. We will also compare the degree-sequence model to the cardinality matrix constraint [Régin and Gomes, 2004], which could be used to enforce a degree sequence. The most scope for improvement in the application problems is in symmetry detection, and we have begun to categorize and break the symmetries in the graphs.

12 Acknowledgments

We thank Ian Miguel and Barbara Smith for comments on an earlier version of this paper. We are grateful for support from Science Foundation Ireland ((00/PI.1/C075), Enterprise Ireland (SC/2003/81) and Ilog SA.

References

[Cormen *et al.*, 2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2001.

[Erdös and Gallai, 1960] P. Erdös and T. Gallai. Graphs with prescribed degrees of vertices. *Mat. Lapok*, 11:264–274, 1960.

[Gould, 1988] R. J. Gould. *Graph Theory*. 1988.

[Hakimi, 1962] S.L. Hakimi. On the realization of a set of integers as degrees of the vertices of a graph. *J. SIAM Appl. Math.*, 10:496–506, 1962.

[Havel, 1955] V. Havel. A remark on the existence of finite graphs. *Casopis Pest. Mat.*, 80:477–480, 1955.

[Le Pape *et al.*, 2002] C. Le Pape, L. Perron, J.-C. Régin, and P. Shaw. Robust and parallel solving of a network design problem. In *CP2002 (ed. P. van Hentenryck), LNCS 2470*, pages 633–648, 2002.

[Mihail and Vishnoi, 2002] M. Mihail and N. K. Vishnoi. On generating graphs with prescribed vertex degrees for complex network modelling. In *ARACNE 2002*, pages 1–11, 2002.

[Pesant and Soriano, 2002] G. Pesant and P. Soriano. An optimal strategy for the constrained cycle cover problem. *Annals of Mathematics and Artificial Intelligence*, 34:313–325, 2002.

[Régin and Gomes, 2004] J.-C. Régin and C. Gomes. The cardinality matrix constraint. In *CP2004 (ed. M. Wallace), LNCS 3258*, pages 572–587, 2004.

[Shiloach, 1981] Y. Shiloach. Another look at the degree constrained subgraph problem. *Inf. Proc. Letters*, 12:89–92, 1981.

[Simonis, 2004] H. Simonis. Constraint applications using graph theory results. CPAIOR 2004 masterclass, 2004. http://www.icparc.ic.ac.uk/ hs/.

[Sorlin and Solnon, 2004] S. Sorlin and C. Solnon. A global constraint for graph isomorphism problems. In *CPAIOR 2004, LNCS 3011*, pages 287–301, 2004.

[Wu, 2004] C. W. Wu. Modelling chemical reactions using constraint programming and molecular graphs. In *CP2004 (ed. M. Wallace), LNCS 3258*, page 808, 2004.