Are Adjacency Lists Worthwhile in AllDifferent?

Peter Nightingale

11th November 2009

Abstract

In our AllDifferent paper [3], there was a loose end. The graph algorithms always discover the graph as they traverse it, by querying variable domains. It might be more efficient to store the graph (and backtrack it) in the form of adjacency lists. This is done here, and we show that it doesn't affect the main conclusions of the paper.

1 Introduction

Régin's GAC AllDifferent algorithm [5] is probably important to have in a CP solver. It uses some graph theory to compute the Hall sets. Figure 1 shows a simple example of a flow graph, and its corresponding variable-value graph. In the variable-value graph, a variable is connected to a value iff the variable has that value in its domain.

The first part of the algorithm is to find a maximal flow in the flow graph (or, equivalently, a maximal matching in the variable-value graph). This is done with a standard algorithm. The second part is to reverse the edges which carry the maximal flow from s to t, and then find the strongly-connected (biconnected) components (SCCs) of this graph. Some of the SCCs correspond to Hall sets, and are used to prune the domains. Tarjan's algorithm can be used to find SCCs.

In our paper [3] we investigated lots of variants of Régin's allDifferent algorithm, including ideas that were published before (staging, incremental matching, priority queuing, domain counting) and a couple of others:

- Identifying important domain values. The propagator only needs to be called when an important value is removed. Irit Katriel did theoretical work about this [4]. We created a practical algorithm, and found that it wasn't particularly good. Also, it didn't fit well with the watched literals in Minion, because the important values are not backtrack-stable therefore the triggers have to be backtracked, adding overhead. (We called these backtracking triggers *dynamic triggers*).

- Dynamically partitioning the constraint during search. The idea is also not new ([2] slide 61) but we found no algorithm / data structure in the

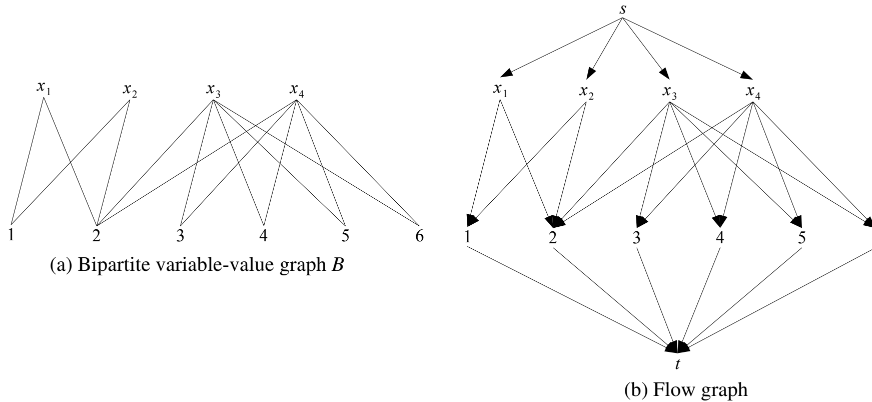

Figure 1: A simple flow graph and corresponding variable-value graph

literature. The constraint naturally partitions into the SCCs of the flow network, which have to be computed anyway. This reduces the average cost of running Tarjan's algorithm (the second stage of allDifferent), and reduces some housekeeping costs as well. The overhead is storing the SCCs, and looking up which part of the constraint is triggered for each variable event. This idea worked well, in some cases speeding things up by over five times.

For all the experiments in our paper, the graph algorithms discover the graph as they traverse it. This is done by reading the variable domains. At a variable vertex, the program iterates through the domain of the variable. Since Minion does not have domain iterators, the program loops from the lower bound to the upper bound, testing each value to see if it is in the domain. At a value vertex, the program iterates through the eligible set of variables, testing if the value appears in the domain of each one. (The eligible set of variables is either all variables, or the variables in the current SCC when doing dynamic partitioning.)

Querying domains is very fast in Minion. However, it might be better to maintain adjacency lists, updating them whenever the domains change. Clearly it's not a new idea in graph algorithms, and Régin hints that he used it originally (by saying that the variable-value graph is maintained within the constraint). Previously we took the line that maintaining lists would probably not pay back its overhead. Testing this hypothesis is the main topic of this note.

2 Implementing adjacency lists

We store the adjacency lists of the bipartite variable-value graph, so for each variable we have a list of all values in its domain[1], and for each value we have

[1] Minion does not have domain iterators. The adjacency list for the variables allows efficient iteration over the domains.

a list of variables whose domain contains the value.

Each adjacency list is an iterable list of integers which is backtracked. Removals must be very efficient, and also iterating through the list, but the order of elements is not important. It is important that it is backtracked efficiently. We settled on the following representation.

List An array of integers, not backtracked.

ListSize A single integer representing the size of the adjacency list. This must be backtracked.

InvList An array of integers giving the position of each item in List. Not backtracked.

This representation has the advantage of minimizing the backtracking memory.

The removal operation for an int a is to swap it with the item at the end of the list (i.e. to the position indicated by ListSize), and then to reduce ListSize by one, thus a disappears from the list. On backtracking, ListSize is restored and a reappears in the list, at the end. InvList allows a to be found in constant time, and is updated when the swap is performed.

If ListSize were trailed, the removal operation (including restoration on backtracking) would be $O(1)$. However, ListSize is currently block-copied because there is no trailed memory manager in Minion 0.9.

3 Experiments with adjacency lists

First, we took the algorithm with all the best options *except* dynamic partitioning. (It has priority constraint queue, incremental matching, staging and uses the Ford-Fulkerson algorithm with BFS to maintain the matching.) This is the variant named Baseline in the paper [3]. We used the same set of instances as in the paper, and a timeout of 1 hour. I refer to the new variant as Lists. Figure 2 shows a comparison of node rates, showing that adjacency lists can significantly speed up the propagator, especially for the larger and harder instances. For the instances which timed out, most showed an improvement of over 30% in node rate.

One of the main conclusions of our paper was that dynamic partitioning with SCCs works very well. Does this still hold when using adjacency lists? The two variants are called Lists-SCC and Lists. Figure 3 shows that dynamic partitioning is still very helpful, increasing the node rate by five times in some cases. Figure 4 is the equivalent graph without adjacency lists. Comparing the two graphs, it is clear that using adjacency lists somewhat decreases the effect of dynamic partitioning. This is because dynamic partitioning helps when iterating through variables (to find those with a certain value — there is no need to consider all variables, only those in the current SCC). Adjacency lists take away the need to iterate through variables in this way, so they somewhat reduce the effect of dynamic partitioning. Even so, the conclusions of the paper regarding dynamic partitioning still stand.

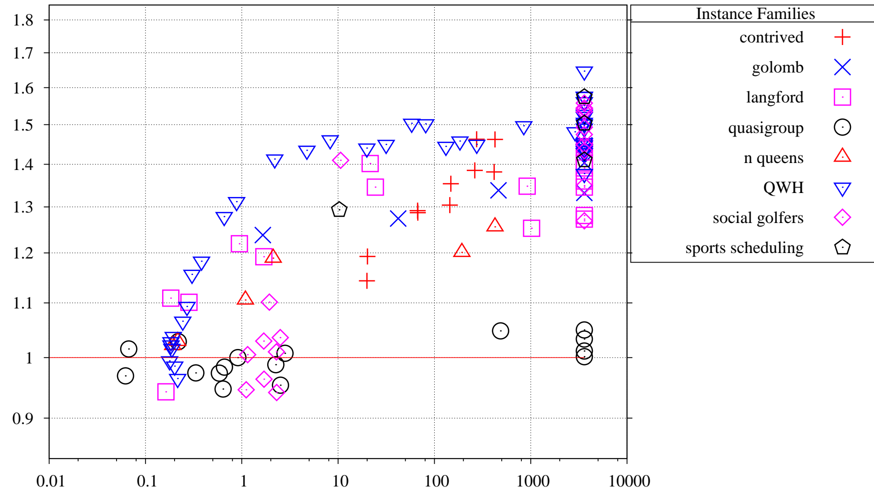

Figure 2: Comparing node rates (nodes per second), Lists over Baseline (y-axis) against Baseline solution time (x-axis)

Previously we concluded that identifying important domain values was not worth the overhead (with the main overhead being moving and backtracking the dynamic triggers). There is no need to re-evaluate this conclusion in the presence of adjacency lists, because adjacency lists are intended to reduce the cost of the graph algorithms, whereas the dynamic triggers were designed to call the graph algorithms less often. If adjacency lists are worthwhile, then there is less opportunity for dynamic triggers to be worthwhile.

Finally, is it worth using adjacency lists when we are doing dynamic partitioning? Figure 5 compares Lists-SCC with Baseline-SCC. It is a mixed picture. Ignoring the contrived instances, adjacency lists can speed it up by more than 30% and slow it down by 10%. 44 of 108 instances (excluding contrived) are sped up by adjacency lists.

On the basis of this experiment, there does not seem to be a case for changing the default implementation of `gacalldiff` in Minion. Therefore Baseline-SCC remains the default implementation in Minion 0.9 (the current release at the time of writing).

4 Conclusions

There are problems (the Contrived and Golomb families) where adjacency lists are a substantial win, although on average there does not seem to be a case for including them in the default implementation of GAC AllDifferent in Minion.

In Minion, we are constrained to make one choice for all problem classes. However, the new DOMINION solver synthesizer project [1] removes this re-

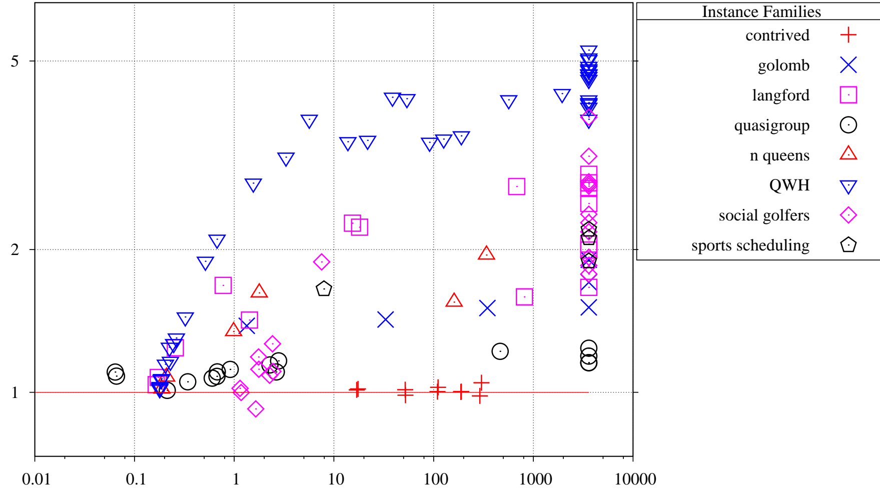

Figure 3: Comparing node rates (nodes per second), Lists-SCC over Lists (y-axis) against Lists solution time (x-axis)

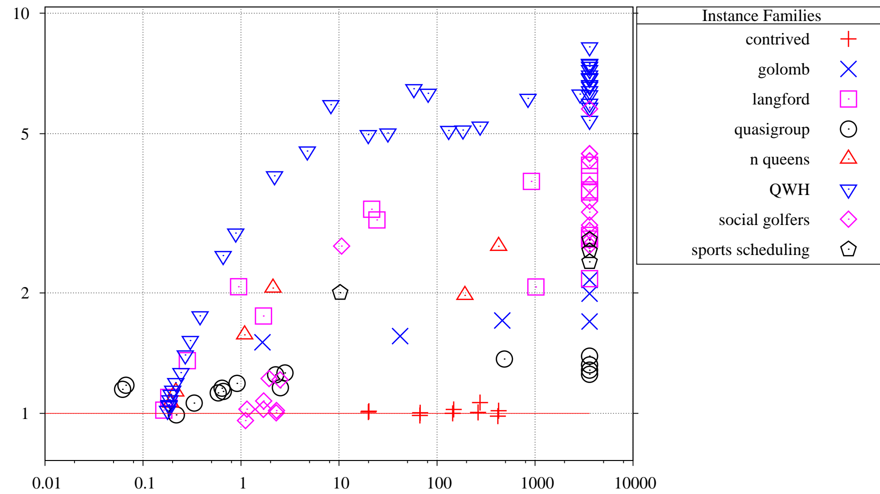

Figure 4: Comparing node rates (nodes per second), Baseline-SCC over Baseline (y-axis) against Baseline solution time (x-axis)

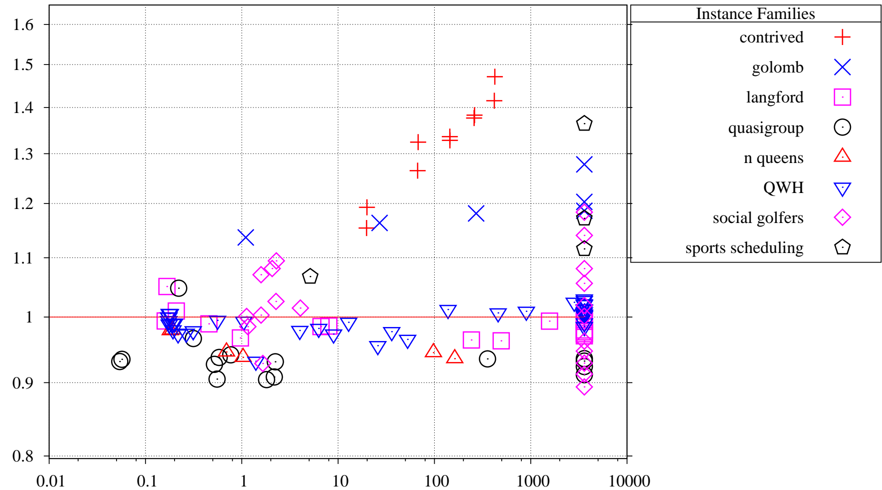

Figure 5: Comparing node rates (nodes per second), Lists-SCC over Baseline-SCC (y-axis) against Baseline-SCC solution time (x-axis)

striction, by synthesizing solvers for problem classes and individual instances. Therefore, in DOMINION we can make different solver implementation decisions for each problem class. Adjacency lists may be an interesting option in this context. With domains that are very sparse (i.e. many large gaps in the domain), adjacency lists are likely to be very useful.

References

[1] Dominion project website. http://dominion.cs.st-andrews.ac.uk/.

[2] Mats Carlsson and Christian Schulte. Finite domain constraint programming systems, 2002. Available from ftp://ftp.sics.se/pub/isl/papers/FD%20Systems.pdf.

[3] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.

[4] Irit Katriel. Expected-case analysis for delayed filtering. In J. Christopher Beck and Barbara M. Smith, editors, *CPAIOR*, volume 3990 of *Lecture Notes in Computer Science*, pages 119–125. Springer, 2006.

[5] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings 12th National Conference on Artificial Intelligence (AAAI 94)*, pages 362–367, 1994.