Negative Cycle Detection in Dynamic Graphs

Nitin Chandrachoodan, Shuvra S. Bhattacharyya*and K.J.Ray Liu
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742
(nitin,ssb,kjrliu@eng.umd.edu)

September 30, 1999

Abstract

We examine the problem of detecting negative cycles in a dynamic graph, which is a fundamental problem that arises in electronic design automation and systems theory [1, 2, 3, 4]. Previous approaches used for this have tried to modify Dijkstra's algorithm since it is the fastest known Single-Source Shortest Path algorithm.

We introduce the concept of *batch mode* negative cycle detection, in which a graph changes over time, and negative cycle detection needs to be done periodically. Such scenarios arise, for example, during iterative design space exploration for hardware and software synthesis. We present an algorithm for this problem, based on the Bellman-Ford algorithm, which outperforms previous approaches.

We also show that this technique leads to very fast algorithms for the computation of the maximum-cycle mean (MCM) of a graph, especially for a certain form of *sparse graph.* Such sparseness often occurs in practice, as demonstrated for example by the ISCAS 89/93 benchmarks.

We present experimental results that demonstrate the advantages of our batch-processing techniques, and illustrate their application to design-space exploration by developing an automated local-search technique for multiple-voltage scheduling of iterative data-flow graphs.

1 Introduction

Detecting the presence of negative cycles in a weighted graph has several applications in systems theory[1]. One of the most important applications is to determine whether a system of constraints has a feasible solution. Finding minimum-cost flows in a network also require the detection of negative cycles.

There are also situations in which it is useful or necessary to maintain a feasible solution to a set of difference constraints as a system evolves. Typical examples of this would be real-time or interactive systems, where constraints are added or removed one (or several) at a time, and after each such modification it is required to determine whether the resulting system has a feasible solution and if so, to find it. When more than one change is made at a time, we say that the changes are being made in a "batch" mode.

*Also with the University of Maryland Institute for Advanced Computer Studies

In this paper, we present an improved algorithm for the batch mode problem, and show how it can be used to derive a fast algorithm for the problem of computing the maximum cycle-mean of a weighted digraph. We show by experimental results that the resulting MCM algorithm is competitive with the fastest known algorithm for calculating the cycle-mean (Howard's algorithm [3, 5]).

We also present a search technique for finding Iterative-DFG schedules, which uses the batch-mode negative cycle detection algorithm as a subroutine. We illustrate the use of this local search technique by applying it to a couple of problems from High-Level Synthesis, namely homogeneous multiprocessor scheduling and resource constrained scheduling for minimum power in the presence of multiple voltage functional units.

2 Previous Work

[1] contains an extensive survey of algorithms for detecting negative cycles, and they also present several problem families that can be used to test the effectiveness of a cycle-detection algorithm. One surprising fact is that the best known theoretical bound ($O(nm)$) for solving the shortest path problem (with arbitrary weights) is also the best known time bound for the negative-cycle problem.

Recently, there has been increased interest in the subject of *dynamic* algorithms for solving problems[6, 7, 4]. This uses the fact that in several problems where a graph algorithm such as shortest paths or transitive closure needs to be solved, it is often the case that we need to repeatedly solve the problem on variants of the original graph.

In our case, [4] presents an algorithm for maintaining shortest paths in arbitrary graphs which performs better than starting from scratch, while [8] presents a generalization of the shortest path problem and shows how it can be used to handle the case where there are few negative weight edges. In both these cases, they have considered one change at a time (not batch mode), and the emphasis has been on the theoretical time bound, rather than experimental analysis. [9] contains an experimental study, but for the case of positive weight edges only.

The most significant work along the lines we propose is [2]. In this, the authors use the observation that in order to detect negative cycles, it is not necessary to maintain a tree of the shortest path lengths to each vertex. They then suggest an improved algorithm based on Dijkstra's algorithm, which is able to recompute a feasible solution (or detect a negative cycle) in time $O(m+n\log n)$, or in terms of output complexity (defined and motivated in [6]) $O(||\Delta|| + |\Delta|\log|\Delta|)$. The output complexity is basically a measure of the total change in input and output. Thus, the change in input would reflect the number of edges which have been changed, while the change in output would count the total number of nodes whose shortest paths (or equivalently the constraint solution) need to be recomputed. Note that this work also considers the addition/deletion of constraints only one at a time.

The above problem can be generalized to allow several changes to the graph between calls to the negative cycle detection algorithm. In this case, the above algorithms would take time linear in the number of changes. Such situations would arise naturally in an interactive environment (if we prefer to accumulate changes between refreshes of the state) or in design space-exploration, as can be seen, for example, in section 5.2.

Algorithm 1 Batch Bellman-Ford Algorithm

Input: Graph $G(V, E)$, $dist(v)$, $length(e)$

Output: updated $dist(v)$ such that $\forall e = (u \to v) \in E : dist(v) - dist(u) \leq length(u \to v)$

```
1: $Q1 \leftarrow \phi, Q2 \leftarrow \phi$
2: for all $e \in E$ do
3:   if $dist(v) - dist(u) > length(u \to v)$ then
4:     append $u$ to $Q1$
5:   end if
6: end for
7: while $Q1$ not empty do
8:   $u \leftarrow pop(Q1)$
9:   for all $v$ adjacent to $u$ in $G$ do
10:    if $dist(v) - dist(u) > length(u \to v)$ then
11:      delete subtree rooted at $v$
12:      if $u$ was in the subtree deleted above then
13:        negative cycle detected: return
14:      else
15:        make $v$ a child of $u$ {constructing subtree}
16:        append $v$ to $Q2$
17:      end if
18:    end if
19:  end for
20:  if $Q2$ is empty then
21:    return {completed: $dist$ satisfies constraints}
22:  else
23:    $Q1 \leftarrow Q2, Q2 \leftarrow \phi$
24:  end if
25: end while
```

In this paper, we show that our approach performs almost as well as the approach in [2] (experimentally) for changes made one at a time, and significantly outperforms their approach for the batch mode (this is true even for relatively small batch-sizes, as will be seen from the results). Also, at very large batch sizes, our algorithm reduces to the normal Bellman-Ford algorithm, starting from scratch, so we do not lose in performance.

3 The Modified Bellman-Ford Algorithm

In this section, we describe the modifications we have made to the Bellman-Ford algorithm to adapt it to the problem of dynamic negative cycle detection, and show the correctness of the modified version.

We first note that the problem of detecting negative cycles in a digraph is equivalent to finding whether or not a set of difference inequality constraints has a feasible solution. To see this, observe that if we have a set of difference constraints of the form

$$x_i - x_j \leq b_{ij}$$

we can construct a digraph with nodes corresponding to the x_i, and edges such that $length(e_{ij}) = b_{ij}$. Then, solving for shortest paths in this graph would yield a set of distances $dist$ which satisfy the constraints on x_i. This graph is henceforth referred to as the *constraint graph*.

The usual technique used to solve for *dist* is to introduce an imaginary vertex s_0 to act as a source, and edges from this vertex to each of the other vertices, with $length = 0$. In this way, we can use a single-source shortest paths algorithm to find *dist* from s_0, and any negative cycles (infeasible solution) will occur only in the original graph, since the new vertex and edges cannot create cycles. This graph is referred to as the *augmented graph* [2].

The algorithm presented in Alg. 1 is basically Tarjan's subtree disassembly method for negative cycle detection (the modifications are in lines 2-6). It works as follows: each time we find a *constraint violation* (i.e. an edge where $dist(v) - dist(u) > length(u \to v)$), we adjust $dist(v)$ so that equality is satisfied in the constraint, and make v a *child* of u. In this way, as we proceed, we will build a *tree* T_c of *critical edges*, rooted at s_0. Also, when we make v a child of u, we have now changed $dist(v)$, and so the subtree rooted at v no longer represents the correct *dist* values and critical edges at this stage. We therefore delete this subtree rooted at v. If, in doing so, we happen to remove u from the main tree rooted at s_0, it means that we have encountered a negative cycle.

The modifications in lines 2-6 accomplish two purposes. The first is to eliminate the need for explicitly introducing the augmenting vertex s_0. This can be done once we recognize that the only purpose served by this vertex is to initialize the other vertices to a set of *dist* values which can then be updated by the rest of the algorithm to obtain a correct solution. The other purpose is that, when combined with the idea of retaining *dist* values between calls to the routine, these lines have the effect of initializing the $Q1$ queue with only those nodes which have been modified, and hence have an effect on the new *dist* values.

3.1 Correctness of the method

The use of a shortest path routine to find a solution to a system of constraint equations is based on the following 2 theorems (for proof see [10]).

THEOREM 1 *A system of difference constraints is consistent if and only if its augmented constraint graph has no negative cycles if and only if its constraint graph has no negative cycles.*

THEOREM 2 *Let G be the augmented constraint graph of a consistent system of constraints $\langle V, C\rangle$. Then D is a feasible solution for $\langle V, C\rangle$, where*

$$D(u) = dist_G(s_0, u)$$

In the above theorem, the *constraint graph* is defined as in sec. 3 above, the *augmented graph* consists of this graph with an additional source vertex (s_0) which has 0-weight edges leading to all the other existing nodes, and *consistency* means that a set of x_i exist which satisfy all the constraints in the system.

The modifications that we have made can be seen to be correct if we relax the definition of the augmented graph so that the augmenting edges (from s_0) do not have 0 weight. In other words:

THEOREM 3 *Consider a constraint graph augmented with a source vertex s_0, and edges from this vertex to every other vertex v, such that these augmenting* *edges have arbitrary weight $length(s_0 \to v)$. The system of constraints is consistent if and only if the augmenting graph defined above has no negative cycles if and only if its constraint graph has no negative cycles.*

Proof: This theorem, which deals with the consistency of the system, is essentially the same as Theorem 1. Clearly, since s_0 does not have any in-edges, no cycles can pass through it. So any cycles, negative or otherwise, which are detected in the augmented graph, must have come from the original constraint graph, which in turn would happen only if the constraint system was inconsistent (by theorem 1). Also, any inconsistency in the original system would manifest as a negative cycle in the constraint graph, and the above augmentation cannot remove any such cycle. $\square$

Now for the validity of the solution:

Theorem 4 *If G' is the augmented graph with arbitrary weights as defined above, and $D(u) = dist_{G'}(s_0, u)$ (single source shortest paths from s_0), then*

1. *D is a solution to $\langle V, C\rangle$*
2. *Any solution to $\langle V, C\rangle$ can be converted into a solution to the constraint system represented by G' by adding a constant to each $D(u)$.*

Proof: The first part is obvious, by the definition of shortest paths.

Now we need to show that by augmenting the graph with arbitrary weight edges, we do not prevent certain solutions from being found. To see this, first note that any solution to a difference constraint system is modifiable by a constant. *i.e.* we can add or subtract a constant to all the $D(u)$ without changing the validity of the solution.

In our case, if we have a solution to the constraint system which does not satisfy the constraints posed by our augmented graph, it is clear that the constraint violation can only be on one of the augmenting edges (since the underlying constraint graph is the same). Therefore, if we define

$$l_{max} = \max_{augmenting\ edges} length(e)$$

and

$$D'(u) = D(u) - l_{max}$$

we ensure that D' satisfies all the constraints of the original graph, as well as all the constraints on the augmenting edges. $\square$

So an augmented graph with arbitrary weights on the augmenting edges can also be used to find a feasible solution to a constraint system. This means that, once we have found *a* solution to the constraint system, we can now change the augmented graph so that the weights on the edges are $dist(snk(e))$. Now even if we change the underlying constraint graph in any way, we can use the new augmented graph to run shortest paths and test the feasibility of the solution.

Note that now T_c (defined in section 3) may no longer be a tree rooted at s_0, but rather, it becomes a forest of trees, each rooted at some node. This does not affect the correctness of the algorithm, since the criterion for negative cycles remains the same in this case as well.

In Alg. 1, lines 1-6 implement the first step of the above process: they insert only those vertices into $Q1$ which are affected as a result of the modifications we

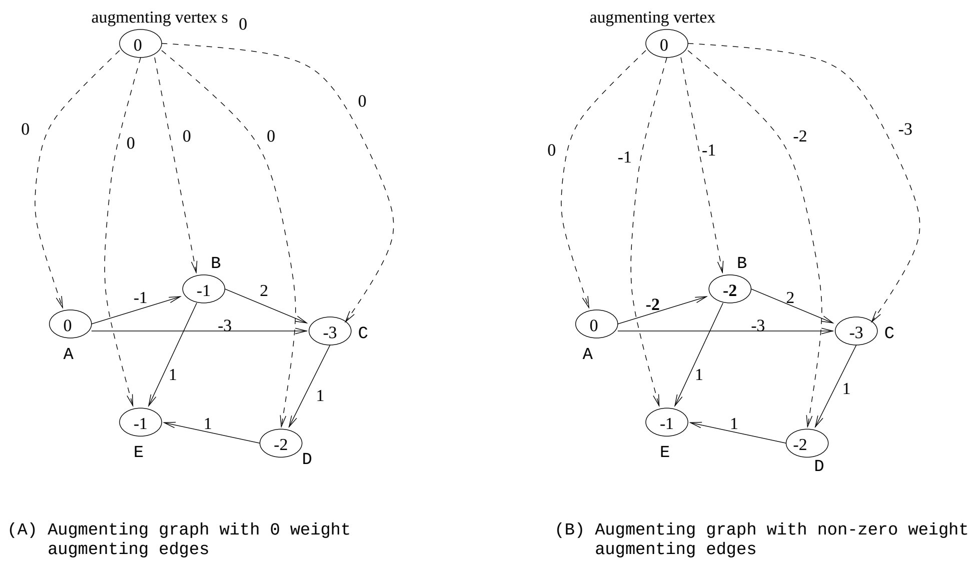
augmenting vertex s
0
0
0
0
0
0
0
B
-1
2
0
A
-1
-3
-3
C
1
1
-1
E
1
-2
D
(A) Augmenting graph with 0 weight augmenting edges
augmenting vertex
0
0
-1
-1
-2
-3
B
-2
2
0
A
-2
-3
-3
C
1
1
-1
E
1
-2
D
(B) Augmenting graph with non-zero weight augmenting edges

Figure 1: Constraint graph

make to the graph. After this, the algorithm proceeds exactly along the lines of the normal Bellman-Ford algorithm.

Figure 1 helps to illustrate the concepts that are explained in the previous paragraphs. In part (B) of the figure, there is a change in the weight of one vertex. But as we can see from the augmented graph, this will result in only the single update to the affected vertex itself, and all the other vertices will get their constraint satisfying values directly from the previous iteration.

In the rest of the paper, we refer to this algorithm as the "Batch Bellman-Ford algorithm" or BBF algorithm, to stress the fact that it is particularly efficient at handling batches of changes made to the constraint system.

4 Comparison against Other Incremental Algorithms

We compare the `BBF` algorithm against (a) the incremental algorithm developed in [2] for maintaining a solution to a set of difference constraints, and (b) a straightforward modification of Howard's algorithm [5], since it appears to be the fastest algorithm to compute the cycle mean, and hence can also be used to check for feasibility of a system.

We have implemented all the algorithms under the LEDA [11] framework for uniformity. The tests were run on random graphs, with several random variations performed on them thereafter. We kept the number of nodes constant and changed only the edges. These changes were precomputed to ensure that they did not add to the measurement of running time. They are of 3 types:

- *Edge insertion:* An edge is inserted into the graph, ensuring that multiple edges between vertices do not occur.
- *Edge deletions:* An edge is chosen at random and deleted from the graph. Note that, in general, this cannot cause any violations of constraints.

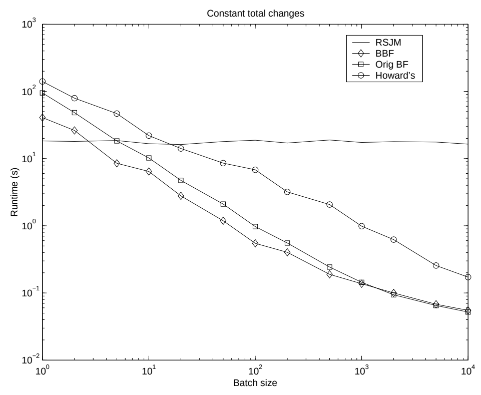

Figure 2: Comparison of algorithms as batch size varies

- *Edge weight change:* An edge is chosen at random and its weight is changed to another random number.

Figure 2 shows a comparison of the running time of the 3 algorithms on random graphs. The graphs in question were randomly generated, had 1000 nodes and 2000 edges each, and a sequence of 10000 edge change operations (as above) were applied to them. The X-axis shows the "granularity" of the changes. That is, at one extreme, we apply the changes one at a time, and at the other, we apply all the changes at once and then compute the correctness of the result. Note that the batch nature is not used by algorithm RSJM, which uses the fact that only one change occurs per test to look for negative cycles. As can be seen, the algorithms which use the batch mode benefit greatly as the batch size is increased, and even among these, the BBF algorithm far outperforms the Howard algorithm, because the latter actually recomputes most of the cycle-mean, which is far more than necessary.

Figure 3 shows a plot of what happens when we apply 1000 iterations to the graph, but change the batch size, so that the total number of changes actually varies from 1000 to 100,000. As expected, RSJM takes total time linear in the number of changes. But the other algorithms take nearly constant time as the batch size varies, which provides the benefit.

From the figures, we see, as expected, that the RSJM algorithm takes time linear in the total number of changes. Howard's algorithm also appears to take more time when the number of changes increases. Figure 2 allows us to estimate at what batch size each of the other algorithms becomes more efficient than the RSJM algorithm. Note that the scale on this figure is logarithmic.

For large batch sizes, the BBF algorithm essentially reduces to the standard algorithm. This can be seen in figure 2, where the two curves meet when the batch size approaches the total number of changes.

Another point to note with regard to these experiments is that they represent the relative behavior for graphs with 1000 nodes and 2000 edges. These numbers were chosen to obtain reasonable run-times on the experiments. The effect of changing the graph size is that for larger graphs, the advantage of the BBF algorithm appears to be more pronounced.

In particular, for larger graphs, the "break-even" point, where the BBF algorithm becomes more efficient than the RSJM algorithm, seems to shift downwards, so that at 10000 nodes and 20000 edges, even a batch size of 2 is faster using

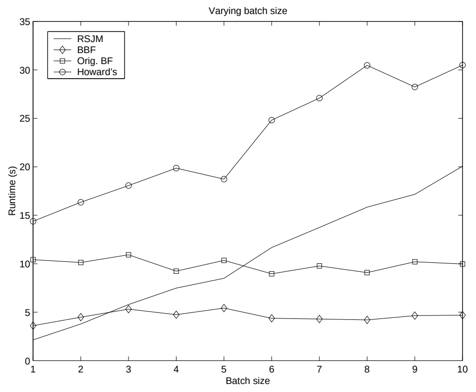

Figure 3: Constant number of iterations at different batch sizes

the `BBF` algorithm.

5 Applications

5.1 Maximum Cycle Mean computation

The first application we consider is computation of the Maximum Cycle-Mean of a weighted digraph. This is defined as the maximum over all directed cycles of the sum of the arc weights divided by the number of "delay" elements on the arcs. This measure plays an important role in discrete systems and embedded systems [3], since it represents the greatest throughput that can be extracted from the system.

To model this application, the edge weights on our graph consist of the value

$$length(u \to v) = delay(e) \times P - exec\_time(u)$$

where $length(e)$ refers to the weight of the edge, $delay(e)$ refers to the number of delay elements (flip-flops) on the edge, $exec\_time(u)$ is the propagation delay of the circuit element that is the source of the vertex, and P is the desired clock period that we are testing the system for. In other words, if the graph with weights as mentioned above does not have negative cycles, then P is a feasible clock for the system. We can then perform a binary search in order to compute P to any precision we require.

It is clear that the best performance bound that can be placed on the algorithm as it stands is $O(nm \log T)$ where T is the maximum value of P that we examine in the search procedure. However, we find that experimentally it performs significantly faster than would be expected by this bound. One point to note is that since we are doing a binary search on T, we are forced to set a limit on the precision to which we compute our answer. In our experiments, we have used 0.001, or 3 decimal places (with $length(e) \sim O(10^2)$). Each additional digit would require about a 10% increase in run-time. On the other hand, this also gives us the freedom to work at lower precision when we are far away from the exact solution, and increase precision as we get closer to an exact solution.

For an experimental study, we build on the work by Dasdan and Gupta ([3]), where the authors have conducted a fairly extensive study of algorithms for this problem. They conclude that Howard's algorithm ([5]) appears to be the fastest experimentally, even though no theoretical time bounds indicate this. As will be seen, our algorithm performs almost as well as Howard's algorithm on several useful sized graphs, and especially on the ISCAS 89 benchmarks, where it typically performs better.

With regard to the ISCAS benchmarks, note that there is a slight ambiguity in translating the netlists into graphs. This arises from the fact that a DFF (D-type flip-flop) can either be treated as a single edge with a delay, with the fanout proceeding from the sink of this edge, or as k separate edges with unit delay emanating from the source node. In the former treatment, it makes more sense to talk about the $|D|/|V|$ ratio ($|D|$ being the number of D flip-flops), as opposed to the $|D|/|E|$ ratio that we use in the experiments with random graphs. However, the difference between the two treatments is not significant and can be safely ignored.

For comparison purposes, we implemented our algorithm in the C programming language, and compared it against the implementation provided by the authors of [5]. Although the authors do not claim their implementation is the fastest possible, it appears to be a reasonably efficient implementation, and we could not find any obvious ways of improving it.

We also vary the number of edges with delays on them. For this, we need to exercise care, since we may introduce cycles without delays on them. To avoid this, we follow the policy of treating edges with delays as "back-edges" in an otherwise acyclic graph [12]. This view is inspired by the structure of circuits, where a delay element usually figures in the feedback portion of the system. Unfortunately, one effect of this is that when we have low number of delay edges, the resulting graph tends to have an asymmetric shape: it is like an almost acyclic graph with only a few edges in the reverse direction. It is not clear how to get around this problem in a fashion which does not destroy the symmetry of the graph, since this requires solving the feedback arc set problem, which is NP-hard.

Intuitively, however, for the above situation, we would expect our algorithm to perform better. This is because, for the MCM problem, a change in the value of P for which we are testing the system will cause changes in the weights of those edges which have delays on them. If these are fewer, then we would expect that fewer operations would be required overall when we retain information across iterations. This is borne out by the experiments. The resulting graphs are Fig. 4 for 20,000 edges and Fig. 5 for 200,000 edges, with the feedback edge ratio varied from 0.1 to 0.9.

Note: For figures 6,7 and 8, we used the random graph generators `SPRAND` from [1]. This assumes unit delays on all edges. Note that unit delays on all edges decreases the benefit from using the incremental algorithm. We can understand why this happens as follows: The algorithm performs a binary search over several values of the iteration period. At each step, the weights of edges in the graph are set to $length(e) = delay(e) \times P - w(e)$, where $w(e)$ was the original weight. So any time we decrease P, all edges that had delays on them will get a new weight, and need to be re-analyzed. Because of this, having fewer edges with delays is more helpful to the incremental algorithm. Since this is normally the situation encountered in physical systems, we consider this a useful behavior.

We note the following features from the experiments:

- If all edges have unit delay, our modifications do not provide much benefit, as expected.

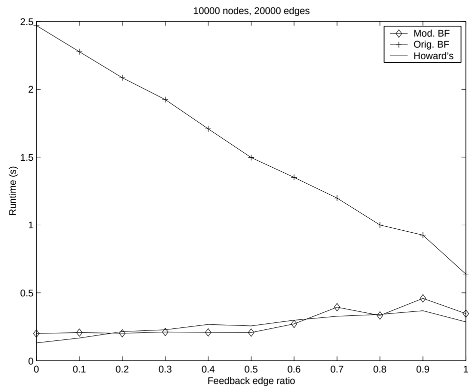

Figure 4: Comparison with Howard's algorithm for 10000 nodes, 20000 edges

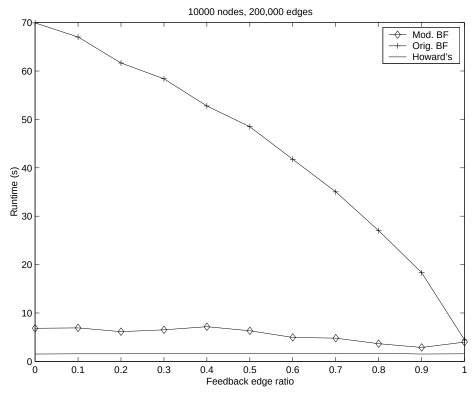

Figure 5: Comparison with Howard's algorithm for 10000 nodes, 200000 edges

- When we vary the number of feedback edges, the benefit of the modifications becomes clear at low feedback ratios.
- From examining practical examples like the ISCAS benchmarks, we can see that all of the circuits have $|E|/|V| < 2$, and $|D|/|V| < 0.1$, ($|D|$ is number of flip-flops, $|V|$ is total number of circuit elements, and $|E|$ is number of edges). In this range of parameters, our algorithm performs very well, even better than Howard's algorithm in several cases (also see Table 1 for the ISCAS results).

5.2 Local search for Scheduling

We also touch upon another application of our technique: for efficient searching of schedules for iterative DFGs. The basic idea is that for scheduling an IDFG, we need to (a) assign nodes to processors and (b) assign relative positions to the nodes within each processor (for sharing). Once these two aspects are done, the schedule for a given time constraint is determined by finding a feasible solution to the constraint equations, which we do using the technique specified above.

The approach we have taken for the schedule search is:

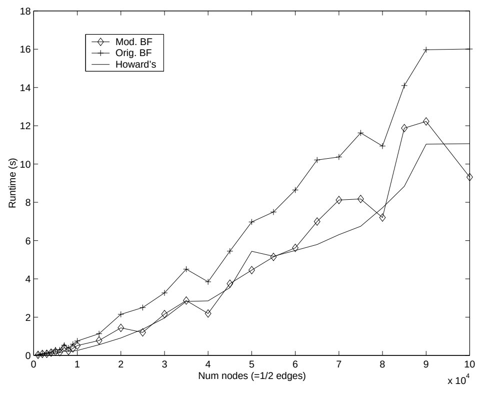

Figure 6: Comparison with Howard's algorithm for $E/V = 2$ and $D/V = 0.1$

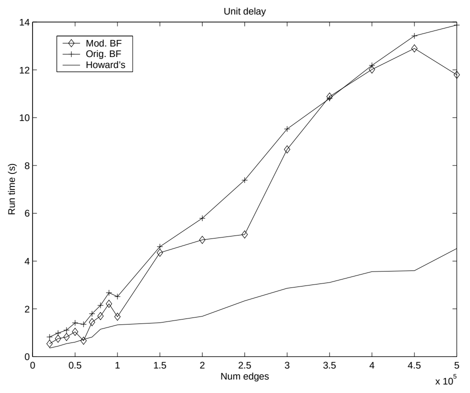

Figure 7: Unit delay: varying edges

- Start with each node on its own processor, find a feasible solution on the fastest possible processor
- Examine each node in turn, and try to find a place for it on another processor (implementing sharing). In doing so, we are making a small number of changes to the constraint system, and need to recompute a feasible solution.
- In choosing the new position, choose one which has minimum power (or area, or whatever cost we want to optimize).
- Additional "moves" that can be made include inserting a new processor type and moving as many nodes onto it as possible, moving nodes in batches from one processor to another etc.
- The technique also lends itself very well to application in schemes using evolutionary improvement.

Each such "move" or modification that we make to the graph can be treated as a set of edge-changes in the precedence/processor constraint graph, and a feasible schedule would be found if the system does not have negative cycles. In addition, the $dist(v)$ values that are obtained from applying the algorithm directly give us the starting times that will meet the schedule requirements.

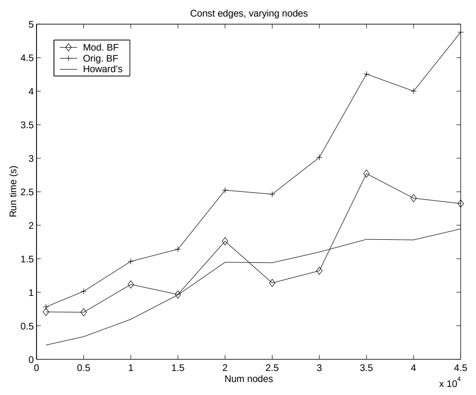

Figure 8: Unit delay: varying nodes

Benchmark	$\lvert E\rvert/\lvert V\rvert$	$\lvert D\rvert/\lvert V\rvert$	orig BF	mod BF	Howard
s38417	1.416	0.069	2.71	0.29	0.66
s38584	1.665	0.069	2.66	0.63	0.59
s35932	1.701	0.097	1.79	0.37	0.09
s15850	1.380	0.057	1.47	0.18	0.36
s13207	1.382	0.077	0.73	0.12	0.35
s9234	1.408	0.039	0.57	0.06	0.11
s6669	1.657	0.070	0.74	0.07	0.04
s4863	1.688	0.042	0.27	0.04	0.03
s3330	1.541	0.067	0.11	0.02	0.01
s1423	1.662	0.099	0.07	0.01	0.01

Table 1: Results for the 10 largest ISCAS 89/93 benchmarks

We have applied this technique to attack the multiple-voltage scheduling problem addressed in [13]. The problem here is to find a schedule for the given DFG that minimizes the overall power consumption, subject to fixed constraints on the iteration period bound, and on the total number of resources available. It is clear that the power savings are basically obtained through scheduling as many adders as possible on 3.3V adders instead of 5V adders. We have used only the basic resource types mentioned in the table to comare our results against those in [13]. There is really no limit imposed by the algorithm itself on the number of different kinds of resources that we can consider.

In tackling this problem, we have used only the most basic method, namely moving nodes onto another existing processor. Already, the results match and even outperform that obtained in [13]. In addition, the method has the benefit that it can handle any number of voltages/processors, and can also easily be extended to other problems, such as homogeneous processor scheduling etc. Table 2 shows the power-savings that were obtained using this technique. S and R power saving indicates the power savings (assuming 25 units for 5V devices and 10.89 units for 3.3V devices) obtained by [13], while BBF power savings refers to the results obtained using our algorithm. T is the overall timing constraint (the maximum iteration period bound that we are aiming for).

We have also tested the application of this technique to homogeneous processor scheduling. This problem is also NP-complete, and several heuristics exist

Benchmark	Resource constraint	T	S and R power saving	BBF power saving
Fifth-order	{(2,+,5V), (2,+,3.3V), (2,*,5V)}	25	268.1(31.54%)	253.98(29.88%)
elliptic filter	{(2,+,5V), (1,+,3.3V), (2,*,5V)}	25	155.21(18.26%)	141.1(16.6%)
	{(2,+,5V), (2,+,3.3V), (2,*,5V)}	22	197.54(23.24%)	211.65(24.9%)
	{(2,+,5V), (1,+,3.3V), (2,*,5V)}	21	112.88(13.28%)	112.88(13.28%)
FIR filter	{(1,+,5V),(2,+,3.3V),(1,*,5V)}	15	169.32(29.45%)	197.54(34.36%)
	{(1,+,5V),(2,+,3.3V),(2,*,5V)}	10	98.77(17.18%)	141.1(24.54%)

Table 2: Comparison against Sarrafzadeh and Raje (ISCAS 99)

for it. One of the best known is Range-chart scheduling ([12]).

We used the inputs used by [12] and tried the above technique, but this time instead of using power, we use area (equiv. number of processors) as our cost criterion. The processors are uniform, with + taking 1 unit of time on them, and $\times$ taking 2 units of time. The results are shown in table 3. It is clear that, while it does not achieve the best possible results, it is very close. Further improvements in terms of moves as mentioned above can be tried to further improve the results.

6 Conclusions

We have introduced a batch-processing approach (the BBF algorithm) to negative cycle detection in dynamically changing graphs. Our technique explicitly addresses the common, practical scenario in which negative cycle detection must be periodically performed after intervals in which a small number of changes are made to the graph. We have also shown how our batch-processing approach can be exploited to compute the maximum cycle mean of a weighted digraph, which is a relevant metric for many problems in the design and analysis of circuits and systems. We have compared our BBF technique, and BBF-based MCM computation technique against the best known related work in the literature, and have observed favorable performance.

The algorithms can be further extended to attack a scheduling problem, by repeatedly trying different combinations of processor serialization, since the speed of our algorithms allows a quick re-examination of the feasibility of the system after each change. This greatly increases the number of possibilities that we can consider within a given time, and can form the basis of effective local search techniques, including genetic algorithm based techniques.

Since computing power is cheaply available now, it is increasingly worthwhile to employ extensive search techniques for solving NP-hard analysis and design problems such as scheduling.The availability of an efficient batch-processing algorithm for negative cycle detection can make this process much more efficient. We have demonstrated this concretely by employing our BBF algorithm within the framework of a local search strategy for multiple voltage scheduling.

References

[1] B.Cherkassky and A.V.Goldberg, “Negative cycle detection algorithms,” Tech. Rep. tr-96-029, NEC Research Institute, Inc., March 1996.

Example name	T_0	P_{rcgs}	P_{mcm}
Second-order	3	4	*5*
section	4	3	3
	6	2	2
	12	1	1
Jaumann	16	3	3
filter	17	2	*3*
	21	2	2
	33	1	1
All-pole	14	3	*4*
filter	15	3	3
	16	2	2
	31	1	1
16-point	2	16	16
FIR filter	3	11	11
	4	8	8
	5	7	7
	6	6	6
	7	5	5
	8	4	4
	11	3	3
	16	2	2
	31	1	1
Fifth-order	16	4	4
elliptic	17	3	3
filter	22	2	2
	42	1	1

Table 3: Homogeneous processor scheduling, compared against Range-chart guided scheduling

[2] G.Ramalingam, J.Song, L.Joskowicz, and R.E.Miller, “Solving systems of difference constraints incrementally,” *Algorithmica*, vol. 23, pp. 261–275, 1999.

[3] A. Dasdan, S. S. Irani, and R. K. Gupta, “Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems,” in *36th Design Automation Conference*, pp. 37–42, ACM/IEEE, 1999.

[4] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, “Fully dynamic shortest paths and negative cycle detection on digraphs with arbitrary arc weights,” in *ESA98*, vol. 1461 of *Lecture Notes in Computer Science*, (Venice, Italy), pp. 320–331, Springer, August 1998.

[5] J.Cochet-Terrasson, G.Cohen, S.Gaubert, M.McGettrick, and J.-P.Quadrat, “Numerical computation of spectral elements in max-plus algebra,” in *Proc. IFAC Conf. on Syst. Structure and Control*, 1998.

[6] G.Ramalingam, *Bounded Incremental Computation.* PhD thesis, University of Wisconsin, Madison, August 1993. Revised version published by Springer Verlag (1996) as Lecture Notes in Computer Science 1089.

[7] B.Alpern, R.Hoover, B.K.Rosen, P.F.Sweeney, and F.K.Zadeck, “Incremental evaluation of computational circuits,” in *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 32–42, 1990.

[8] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-paths problem," *Journal of Algorithms*, vol. 21, pp. 267–305, 1996.

[9] D.Frigioni, M.Ioffreda, U.Nanni, and G.Pasqualone, "Experimental analysis of dynamic algorithms for single source shortest paths problem," in *Proc. Workshop on Algorithm Engineering (WAE '97)*, 1997.

[10] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.

[11] K. Mehlhorn and S. Näher, "`LEDA`: A platform for combinatorial and geometric computing," *Communications of the ACM*, vol. 38, no. 1, pp. 96–102, 1995.

[12] S. M. H. de Groot, S. H. Gerez, and O. E. Herrmann, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Transactions on Circuits and Systems - I*, vol. 39, pp. 351–364, May 1992.

[13] S. Raje and M. Sarrafzadeh, "Scheduling with multiple voltages under resource constraints," in *Proc. ISCAS 99*, 1999.