2003

An empirical study of algorithms for the negative cost cycle detection problem.

Lisa L. Kovalchick
West Virginia University

An Empirical Study of Algorithms for the Negative Cost Cycle Detection Problem

Lisa L. Kovalchick

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

K. Subramani, Ph.D., Chair
Hany Ammar, Ph.D.
Bojan Cukic, Ph.D.
Hong-Jian Lai, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2003

Keywords: Negative Cycle, Vertex Contraction, Relaxation

ProQuest Information and Learning Company
300 North Zeeb Road
PO Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

An Empirical Study of Algorithms for the Negative Cost Cycle Detection Problem

Lisa L. Kovalchick

In this thesis, we develop a greedy algorithm for the problem of checking whether a directed graph with positive and negative costs on its edges has a negative cost cycle. Our approach is the first known greedy strategy for this problem; all known approaches in the literature are based on Dynamic Programming. It is well known that the Negative Cost Cycle Detection problem is equivalent to the problem of checking whether a system of linear difference constraints is feasible. Our algorithm exploits this equivalence and uses polyhedral projection on the polyhedron of difference constraints corresponding to the input graph to detect the presence of negative cost cycles. We use the Fourier-Motzkin elimination procedure to effect polyhedral projection and contrast the performance of our algorithm with the "standard" Bellman-Ford (BF) algorithm along with more sophisticated implementations of the BF algorithm, such as BFFI (BF with a FIFO queue), BFPR (BF with a predecessor array), BFFP (BF with a FIFO queue and a predecessor array), BFCT (BF with subtree disassembly) and GORC (the Goldberg-Radzik algorithm). Our experiments show that Vertex Contraction (VC) is always superior to BF, BFFI, and BFPR implementations, while it approaches the performance of the other algorithms in certain cases.

ACKNOWLEDGEMENTS

I wish to thank my graduate advisor, Dr. K. Subramani, who gave me the opportunity to work with him and lead me into the research of such an interesting problem. I wish to thank Dr. Francis Vanscoy for providing a computer on which I could implement the algorithms. Dr. Hany Ammar, Dr. Bojan Cukic, and Dr. Hong-Jian Lai are greatly appreciated for serving on my Graduate Committee. Dejan Desovski is appreciated for his helpful suggestions. I wish to thank Dr. Anthony Pyzdrowski for encouraging me to attend graduate school at West Virginia University and for his continued encouragement. Finally, I wish to thank my parents, James and Dorothy Kovalchick, my sister Amanda Kovalchick, and my boyfriend Steven Pettit for their support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

The Negative Cost Cycle Detection Problem

1.1 Introduction

Before defining the Negative Cost Cycle Detection problem, it is important that the reader be familiar with several definitions involving graphs. [CLR92] defines a directed graph G as a pair (V, E) where V is a finite set and E is a binary relation on V. They define a directed weighted graph as a directed graph for which each edge has an associated weight, typically given by a weight function $w : E \rightarrow Z$. Figure 1.1 is an example of a directed weighted graph.

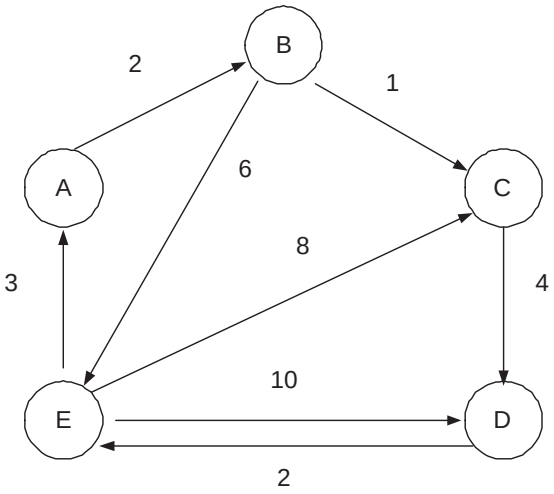

Figure 1.1: A directed weighted graph.

A cycle in a directed weighted graph is defined as a directed weighted path $< v_0, v_1, \ldots, v_k >$, where $v_0 = v_k$; a negative cost cycle is defined as a cycle in which the sum of the weights of each edge in the cycle is negative [CLR92]. Figure 1.2 is an example of a directed weighted graph containing a negative cost cycle.

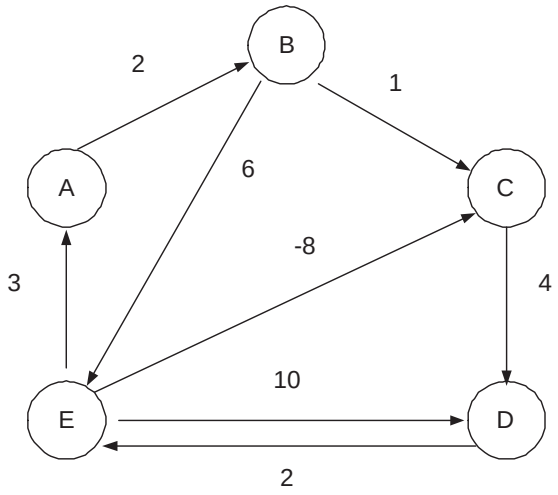

Figure 1.2: A directed weighted graph containing a negative cost cycle (the edges (E, C), (C, D), and (D, E) form the negative cost cycle).

Having stated the above definitions, the Negative Cost Cycle Detection problem can be defined as the problem of finding a negative cost cycle in a directed graph, or proving that there are none [Gol95]. Formally, the Negative Cost Cycle Detection problem is defined as:

Given a directed graph $\mathbf{G} =< \mathbf{V}, \mathbf{E}, \mathbf{c} >$*, where* $\mathbf{V} = \{v_1, v_2, v_3, \ldots, v_n\}$, $|\mathbf{V}| = n$, $\mathbf{E} = \{e_{ij} : v_i \rightsquigarrow v_j\}$, $|\mathbf{E}| = m$*, and a cost function* $\mathbf{c} : \mathbf{E} \rightarrow \mathbf{Z}$*, is there a negative cost cycle in* $\mathbf{G}$*?*

There are no restrictions on the edge costs, i.e., they can be arbitrary integers as opposed to small integers, as required by some scaling algorithms [Gol95]. We note that the problem, as specified, is a decision problem, in that all that is asked of an algorithm is to *detect* the presence of a negative cost cycle; however a simple modification of our algorithm using a stack, can be employed to produce the negative cost cycle if it exists.

In this thesis, we are concerned with the empirical validation of projection-based approaches for the problem of checking whether there exists a negative cost cycle in an arbitrary, directed graph with positive and negative costs on the edges.

1.2 Applications

The Negative Cost Cycle Detection problem has numerous applications in model verification, Compiler Construction, Software Engineering, VLSI design, scheduling, and image processing.

Constraint-based program analysis requires feasibility checking of constraint sets. Constraint graphs are often used to represent systems of difference constraints; an application of Farkas' Lemma shows that a system of difference constraints is feasible if and only if there does not exist a negative cost cycle in the corresponding constraint graph [CLR92, DMP91]. In other words, the Negative Cost Cycle Detection problem is equivalent to the problem of verifying whether a system of difference constraints is feasible. Constraint solving within the context of Compiler Construction is the basic thrust of [BGS00, QHV02] and [Pug92b, Pug92a, PW98]. In [BGS00], a simple and elegant algorithm called `ABCD` is developed for the purpose of checking array bounds; their algorithm uses a linear difference constraint solver that checks for negative cost cycles in the constraint graph. [QHV02] builds on the work in [BGS00] and is concerned with dynamic array bounds checking in Java. They introduce 3 different techniques for the problem, all of which employ linear difference constraint solvers. [Pug92b, Pug92a] and [PW98] use the Fourier-Motzkin elimination procedure to solve a system of constraints that arise in data dependency analysis.

In the design of VLSI circuits, it is required to isolate negative feedback loops. These negative feedback loops correspond to negative cost cycles in the amplifier-gain graph corresponding to the circuit [WE94]. The problem of checking whether a zero-clairvoyant scheduling system has a valid schedule can also be reduced to the problem of identifying negative cost cycles in the appropriate graph [Sub02]. Recent approaches to the image segmentation problem are also based on negative cost cycle detection [CRY96, CGGV95].

All other negative cost cycle detection algorithms are essentially Single Source shortest path algorithms, whereas ours is a local approach for the sole purpose of detecting negative cost cycles.

Our work establishes polyhedral projection as a viable, general-purpose technique to solve network optimization problems involving cycles; while we focus exclusively on the Negative Cost Cycle Detection problem, there are a number of related problems that can be attacked in a similar fashion. We use the Fourier-Motzkin elimination technique (see Section §A) as the backbone of our polyhedral projection algorithm. Although, other polyhedral projection schemes exist in the literature [Wei97, LW93], the Fourier-Motzkin procedure is our method of choice, on account of its simplicity and wide applicability.

All approaches to the Negative Cost Cycle Detection problem in the literature are based on dynamic programming or scaling [Gol95] ; our approach is the first and only greedy approach to this problem, that we know of.

Our experiments indicate that polyhedral projection is an effective alternative to the "standard" Bellman-Ford (BF) algorithm and more sophisticated implementations of the BF algorithm including BFFI and BFPR for the same problem; this is most surprising since in the case of sparse graphs, BF is provably superior to polyhedral projection. The principal conclusion that we draw from our study is that the asymptotic analysis of algorithms in the RAM model may not serve the purpose of predicting efficiency in an empirical setting. One of the issues that we need to consider in the design of algorithms is the nature of the underlying architecture. The presence of caches is almost universal in current architectures and an algorithm needs to exploit "locality of reference", in order to be competitive.

Chapter 2

Related Work

2.1 The "Standard" Bellman Ford Algorithm

One of the earliest and to date the (asymptotically) fastest algorithm for the Negative Cost Cycle Detection problem is the Bellman-Ford (BF) algorithm [CLR92]. The BF algorithm maintains a list of distance labels $d[i]$ for each vertex v_i. Initially, each vertex's distance label is set to ∞, with the exception of the source's distance label which is set to 0 (the source is the vertex where the relaxation begins). If the distance label of a vertex v_i is $d[i] = \infty$, this indicates that a path from the source vertex to vertex v_i has not yet been discovered. Otherwise, $d[i]$ contains the weight of the shortest path from the source to v_i that has been discovered. BF proceeds by relaxing all of the edges of the graph and updating the corresponding distance labels in $n - 1$ stages; at the termination of stage i, cycles of length $(i + 1)$ can be detected. Notice that each stage of the "standard" BF algorithm takes $O(m)$ time (where m is the number of edges in the graph). The number of stages that must be carried out is equal to one less than the number of vertices in the graph, thus, for a graph containing n vertices the "standard" BF algorithm runs in time $O(m \cdot n)$.

2.2 The BF Algorithm Implemented with Various Heuristics

[AMO93] presents several heuristics which can reduce the number of steps that are executed by the "standard" BF algorithm; these heuristics include:

1. A FIFO queue (BFFI),
2. A predecessor array (BFPR),
3. both a FIFO queue and a predecessor array (BFFP).

BFFI attempts to reduce the number of vertices that must be examined in each stage of the "standard" BF algorithm by making use of a FIFO queue to store the vertices whose distance labels were changed in the previous stage. Only vertices in the queue will be examined in the current stage and each vertex may be examined only once in each stage. In order to detect a negative cost cycle in a graph of n vertices, a counter is maintained for each vertex. This counter records the number of times that the vertex's distance label has been changed (i.e., the number of times that the vertex has been placed on the queue). Observe that since any vertex may be placed on the queue at most once per stage, each vertex's counter should be at most $n - 1$ if no negative cost cycles exist. There are two cases that can cause termination of the BFFI algorithm:

1. the queue is empty (this will occur only if the graph does not contain a negative cost cycle).
2. one vertex's counter is incremented to n (this indicates that a negative cost cycle exists).

Notice that the BFFI algorithm takes at most $O(m)$ time in each stage and that there are at most $n - 1$ stages in a graph with n vertices. Thus, the total running time for the BFFI algorithm is $O(m \cdot n)$.

BFPR attempts to reduce the number of steps that are carried out by the "standard" BF algorithm by making use of a predecessor array to store the parent ($parent_i$) of each vertex v_i, where $parent_i$ is the vertex that caused the most recent label change to v_i. Each distance label update to a vertex v_i causes the parent of v_i ($parent_i$) to be updated. This happens because the change of a distance label indicates that a shorter path from the source vertex to vertex v_i has been discovered, by using a different parent vertex for v_i. Notice that we can construct a tree by making use of the parent pointers, however if our graph contains a negative cost cycle, we will no longer have a tree because the parent of a vertex v_i in our graph will

be a vertex v_j, which is located at a level further from the root than vertex v_i, thereby creating a cycle and destroying the tree. BFPR detects a negative cost cycle by choosing a source vertex and designating it as labeled while all other vertices are designated as unlabeled, then each unlabeled vertex v_k is examined and assigned a label of v_k. We now trace the predecessors, starting with vertex v_k and assign a label of v_k to each of them until we come to a vertex v_l, which has already been labeled; if v_l's label is the same as v_k's label, then a negative cost cycle exists. Notice that each stage of the BFPR algorithm takes at most $O(m)$ time to complete and that there are at most $O(n)$ stages for a graph of n vertices. Thus the total running time for BFPR is $O(m \cdot n)$.

BFFP combines the techniques of BFFI and BFPR in order to reduce the number of steps taken by the "standard" Bellman Ford algorithm to detect negative cost cycles; like the other algorithms described, BFFP also runs in time $O(m \cdot n)$.

While BFFI, BFPR and BFFP are all $O(m \cdot n)$ algorithms, there exist inputs, which could cause them to execute $\Theta(m \cdot n)$ steps (for instance, a graph with no negative cost cycles).

Another variation of the BF algorithm is BFCT, which combines the "standard" BF algorithm with the FIFO queue idea of BFFI, the predecessor array idea of BFFP and the subtree disassembly cycle detection strategy due to Tarjan [CG96].

2.3 The Goldberg-Radzik Algorithm

The Goldberg-Radzik (GORC) algorithm maintains two sets of labeled vertices (sets A and B) with each vertex being in at most one set at any given time. At the start of the algorithm, the set A is empty and the set B contains only the source vertex (since this is the only vertex which is initially marked as labeled). At each stage of the algorithm, set B is used to compute the set A of vertices to be scanned in the next stage, and the set B becomes the empty set. During each pass, elements are removed from set A, scanned, and any newly labeled vertices are added to set B. A pass ends when the set A becomes empty and the algorithm terminates when the set B becomes empty. GORC is the fastest known empirical strategy for the Negative Cost Cycle Detection problem and is primarily designed to calculate the Single Source shortest paths in the input graph [Gol95]. The number of steps required by the GORC algorithm is $O(m \cdot n)$, although in practice, GORC

usually outperforms other variations of the BF algorithm [Gol95]. GORC is not a comparison based algorithm, whereas all the other algorithms discussed in this thesis use comparisons only. Note that GORC is not an effective strategy in the case that the edge weights are large integers. The principal problem with these algorithms is that they are functional in nature, i.e., they are designed to calculate the actual shortest paths in the graph (*optimization*). As a result, they can be forced to execute a pre-specified number of steps. Our problem is much simpler in the following sense; we are content with merely identifying the presence of a negative cost cycle (*feasibility*). It is therefore at least reasonable to expect that our problem has a faster strategy (even, if only from an empirical perspective) than the problem of finding the shortest paths in a directed graph with positive and negative edge costs.

Chapter 3

A Greedy Approach

3.1 The Vertex Contraction Algorithm

The *vertex contraction* procedure consists of eliminating a vertex from the input graph, by merging all its incoming and outgoing edges. Consider a vertex v_i with incoming edge e_{ki} and outgoing edge e_{ij}. When v_i is contracted, e_{ki} and e_{ij} are deleted and a single edge e'_{kj} is added with cost $c_{ki} + c_{ij}$. This process is repeated for each pair of incoming and outgoing edges. Consider the edge e'_{kj} that is created by the contraction; it falls into one of the following categories:

1. It is the first edge between vertex v_k and v_j. In this case, nothing more is to be done.

2. An edge e_{kj} already existed between v_k and v_j, prior to the contraction of v_i. In this case, if $c'_{kj} < c_{kj}$, keep the new edge and delete the previously existing edge (since it is redundant); otherwise delete the new edge (since it is redundant).

Algorithm (3.1.1) is a formal description of our technique.

Function NEGATIVE-COST-CYCLE($\mathbf{G}, n$)
1: **for** ($i = 1$ **to** n) **do**
2: VERTEX-CONTRACT($\mathbf{G}, v_i$)
3: **end for**
4: **return**(**false**)

Algorithm 3.1.1: Negative cost cycle detection

```
Function VERTEX-CONTRACT(G, v_i)
 1: for (k = 1 to n) do
 2:   for (j = 1 to n) do
 3:     if (e_ki and e_ij exist) then
 4:       {Let c_kj denote the cost of the existing edge between v_k and v_j; note that c_kj = ∞ if there does not exist such
          an edge}
 5:       Create edge e'_kj with cost c'_kj = c_ki + c_ij
 6:       Delete edges e_ki and e_ij from G
 7:       if (j = k) then
 8:         {A cycle has been detected}
 9:         if (c'_jj < 0) then
10:           return(true)
11:         else
12:           Delete edge e_jj
13:         end if
14:       else
15:         if (c'_kj < c_kj) then
16:           Replace existing edge e_kj with e'_kj in G
17:         else
18:           Delete edge e'_kj
19:         end if
20:       end if
21:     end if
22:   end for
23: end for
```

Algorithm 3.1.2: Vertex Contraction

3.2 Correctness and Analysis

The correctness of Algorithm (3.1.1) follows from the observation that vertex contraction is a path-cost preserving operation, i.e., if there is a path of cost c_1 from vertex v_a to vertex v_b, before the contraction of a vertex $v_c, c \neq a, b$, then there exists a path of cost at most c_1 from v_a to v_b, after the contraction of vertex v_c. Every cycle (including one of negative cost) is a path from some vertex to itself and hence negative cost cycles will eventually be detected as negative cost loops about some vertex.

Lemma: 3.2.1 *Algorithm (3.1.1) returns* **true** *if and only if there exists a negative cost cycle in the input graph* $\mathbf{G}$.

Proof: *The proof of Lemma 3.2.1 follows from the correctness of the Fourier-Motzkin elimination procedure.* □

Lemma: 3.2.2 *The running time of Algorithm (3.1.1) is at most* $O(n^3)$.

Proof: *Observe that a single* VERTEX-CONTRACT() *operation can be implemented in* $O(n^2)$ *time, since there are at most* $n-1$ *edges entering a vertex, which have to be combined with the at most* $n-1$ *edges leaving that vertex. It follows*

that contracting n vertices takes time at most $O(n^3)$. □

Thus, for dense graphs, Algorithm (3.1.1) is competitive with Bellman-Ford (BF); however for sparse graphs, the situation is not so sanguine. For instance, an adversary could provide the graph in Figure (3.1) as input.

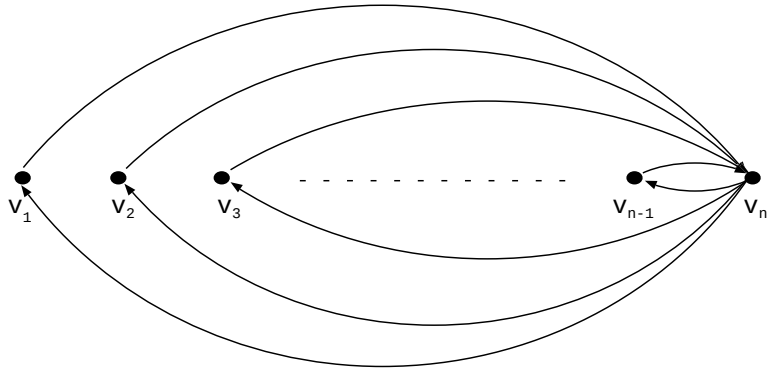

Figure 3.1: Sparse graph that becomes dense after vertex contraction

The above graph is sparse and has exactly $2 \cdot (n-1)$ edges. Observe that if vertex v_n is contracted first, the resultant graph is the complete graph on $n-1$ vertices and is therefore dense. *It follows that Algorithm (3.1.1) has a running time of $\Theta(n^3)$ on this graph, when it has no negative cost cycles.* We call this graph *the cruel adversary*; in our experiments, we made it a point to contrast the performance of all algorithms on this input. It is clear that any well-defined order of selecting the next vertex to be contracted is susceptible to attack by a malicious adversary; we could however choose the vertex to be contracted at random, without affecting the correctness of the algorithm. We have implemented the Vertex Contraction algorithm with 3 different strategies based on how the vertices to be contracted are chosen:

1. the vertex to be contracted is chosen in a well-defined order (VC),

2. the vertex to be contracted is the vertex with the smallest degree product (the degree product is calculated by multiplying the number of edges coming into a vertex v_i with the number of edges going out of the vertex); this is accomplished by using a heap (HVC),

3. the vertex to be contracted is chosen at random (RVC). Algorithm (3.2.1) is a formal description of the Random Vertex Contraction algorithm.

Function RANDOM-NEGATIVE-COST-CYCLE($\mathbf{G}, n$)

1: Generate a random permutation Π of the set $\{1, 2, 3 \ldots, n\}$.
2: **for** ($i = 1$ **to** n) **do**
3: VERTEX-CONTRACT($\mathbf{G}, v_{\Pi(i)}$)
4: **end for**
5: **return**(**false**)

Algorithm 3.2.1: Random negative cost cycle detection algorithm

Chapter 4

Comparison of Vertex Contraction with "Standard" Bellman Ford

4.1 Vertex Contraction versus "Standard" Bellman Ford

In industry, BF is the algorithm of choice on account of its simplicity, and hence from their perspective, it makes sense to compare our algorithm with the "standard" Bellman Ford algorithm. Our experiments are classified into various categories, based on the following criteria:

1. Type of input graph - Sparse with many small negative cost cycles (Type A), Sparse with a few long negative cost cycles (Type B), Dense with many small negative cost cycles (Type C), Dense with a few long negative cost cycles (Type D), and the Cruel Adversary (Type E).
2. Type of Algorithm - Bellman-Ford (BF), Vertex-Contraction (VC) or Random Vertex-Contraction (RVC).
3. Type of Graph Data Structure - Simple Pointer or Array of Pointer.

4.1.1 Machine Characteristics

Table 4.1: Implementation System

Machine Model	Silicon Graphics Onyx2
Processors	IR2/R10 250 Mhz
Cache	8 MB
Memory	2 GB
Operating System	IRIX 6.5.15
Language	C
Software	gcc

4.1.2 Graph Data Structures

Two different types of graph data structures were used for the experiments. We implemented BF, VC and RVC with an array of pointer structure and a simple pointer structure.

The array of pointer structure is our own original representation, requiring quadratic space. This representation makes use of an array of n pointers, one for each of the n vertices of the graph. Each pointer points to an n element array, which corresponds to the n vertices of the graph. Initially all entries of the array are assigned an undefined value. For a vertex v_i, if there exists an edge from v_i to another vertex v_j, position v_j of the array that v_i points to is assigned the cost of the edge between v_i and v_j. It should be noted that this representation is different from the adjacency-matrix representation [CLR92]. Assuming that we are contracting vertex v_i, the vertex contraction operation for the array of pointer structure is performed as follows.

Table 4.2: Time required to perform Vertex Contraction using an Array of Pointer data structure

Step	Time to Execute
1)For each vertex v_x check to see if there is an edge from v_x to v_i	$O(n)$
2) If an edge exists	$O(1)$
3) For each vertex v_y check to see if there is an edge from v_i to v_y	$O(n)$
4) If an edge exists	$O(1)$
5) Add the weight of edge (v_x, v_i) and the weight of (v_i, v_y) and store the result in w	$O(1)$
6) If the weight of edge (v_x, v_y) is greater than w, replace the weight of the edge with w	$O(1)$
7) If a negative cost edge of the form (v_x, v_x) is created, the algorithm terminates with a negative cost cycle being detected	$O(1)$

The time required to contract vertex v_i by the array of pointer implementation is: $O(n \cdot n) = O(n^2)$.

The simple pointer data structure is also known as the adjacency-list representation [CLR92]. This representation makes use of an array of n lists, one for each of the n vertices of the graph. For each vertex v_i, we store the in-degree d_i^{in} (the number of edges going into the vertex), the out-degree d_i^{out} (the number of edges going out of the vertex), and a singly linked list of edges going out from the vertex along with their weights. The linked lists of each vertex are sorted based on the destination vertex of the edge. Assuming that we are contracting vertex v_i, the vertex contraction operation for the simple pointer data structure is performed as follows.

Table 4.3: Time required to perform Vertex Contraction using a Simple Pointer data structure

Step	Time to Execute
1) Scan all edges looking for edges with destination vertex v_i : (v_x, v_i)	$O(m)$
2) For every edge (v_x, v_i) found:	$O(d_i^{in})$
3) Remove the edge from the adjacency list of vertex v_x	$O(1)$
4) Merge v_i's list with v_x's list by adding those edges which are not present in v_x's list and updating those that are already there	$O(d_i^{out} + d_x^{out})$
5) If a negative cost edge of the form (v_x, v_x) is created, the algorithm terminates with the negative cost cycle being detected	$O(1)$

The time required to contract vertex v_i by the simple pointer implementation is:

$O(m + d_i^{in} \cdot d_i^{out} + \sum_x^{(v_x, v_i) \in E} d_x^{out})$. In the worst case when the graph is dense: $m = O(n^2)$, $d_i^{in} = d_i^{out} = O(n)$, and the time complexity of the vertex contraction operation is $O(n^2)$.

4.1.3 Experimental Setup for Sparse Graphs

Sparse graphs were generated using the generator developed by Andrew Goldberg [CG96], which generates multiple edges between two vertices. Sparse graphs are defined as graphs with $o(n \cdot \log n)$ edges. We generated each graph 5 times using 5 different seeds for the random number generator. The times recorded were averaged over 5 executions of each implementation.

Graphs of Type A and B were tested, with the number of vertices ranging from 500 to 5,500 in increments of 250. We define a small negative cost cycle as one consisting of at most $\frac{n}{100}$ vertices. We define a long negative cost cycle as one consisting of $\Omega(\frac{n}{2})$ vertices. The number of long negative cost cycles in the input graph was set to 4.

4.1.4 Observations

	Array of Pointer (Time in Seconds)	
n	VC	BF
500	0.15351	2.80657
750	0.50453	9.37442
1,000	1.58202	27.9044
1,250	2.23023	54.0744
1,500	4.74535	105.143
1,750	5.55235	156.953
2,000	12.7588	257.852
2,250	19.5588	337.136
2,500	13.9183	514.046
2,750	24.3229	624.652
3,000	30.4645	883.024
3,250	34.8372	1034.04
3,500	49.6497	1400.41
3,750	48.4852	1606.85
4,000	88.8478	2104.50
4,250	70.4305	2319.88
4,500	132.506	3094.30
4,750	82.0854	3180.42
5,000	108.178	4229.53
5,250	116.699	4377.80
5,500	133.606	5453.65

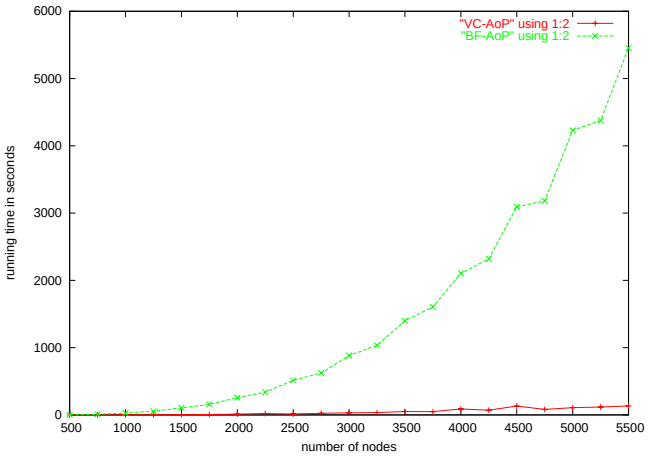

Figure 4.1: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Array of Pointer (AoP) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type A graphs.

n	Simple Pointer (Time in Seconds) VC	BF
500	0.003933	1.65399
750	0.007623	5.19749
1,000	0.009573	11.8637
1,250	0.023780	22.5836
1,500	0.013797	38.9001
1,750	0.013525	64.8949
2,000	0.022071	103.797
2,250	0.022178	155.955
2,500	0.025375	222.823
2,750	0.030861	304.137
3,000	0.040336	403.182
3,250	0.046731	519.489
3,500	0.047264	656.995
3,750	0.071233	814.206
4,000	0.063790	995.579
4,250	0.073681	1199.38
4,500	0.101693	1433.95
4,750	0.083590	1688.46
5,000	0.124874	1981.08
5,250	0.084357	2295.98
5,500	0.087477	2650.91

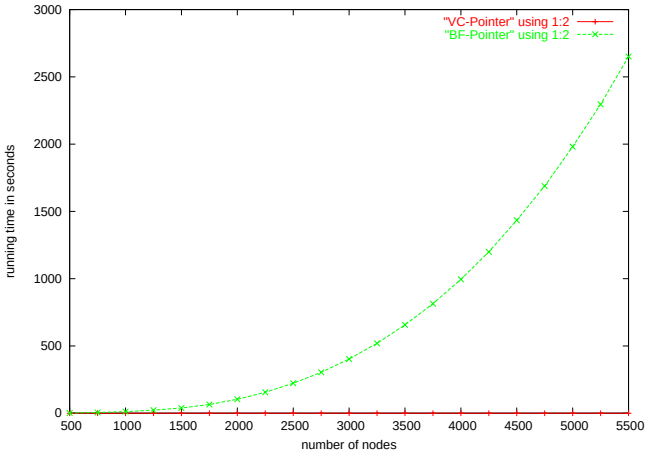

Figure 4.2: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Simple Pointer implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type A graphs.

n	Array of Pointer (Time in Seconds) VC	BF
500	0.22578	2.80053
750	0.51732	9.40496
1,000	1.63598	25.2701
1,250	2.76916	52.7881
1,500	4.03085	103.765
1,750	4.58197	152.435
2,000	11.8345	253.484
2,250	20.4233	330.726
2,500	13.9014	502.027
2,750	23.9882	607.284
3,000	27.1102	875.921
3,250	35.6303	995.875
3,500	49.7201	1383.19
3,750	48.6051	1577.46
4,000	77.5183	2071.36
4,250	65.2763	2307.49
4,500	120.153	2978.97
4,750	83.6087	3209.21
5,000	92.0987	4076.75
5,250	151.066	4376.84
5,500	130.703	5408.41

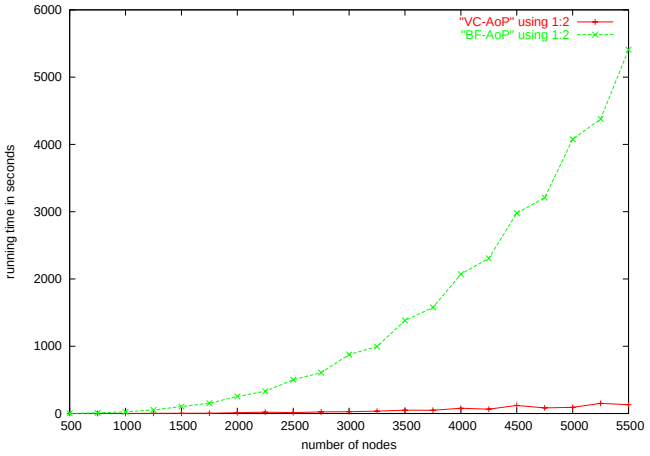

Figure 4.3: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Array of Pointer (AoP) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type B graphs.

n	Simple Pointer (Time in Seconds) VC	BF
500	0.004119	1.65394
750	0.008052	5.20284
1,000	0.011301	11.8367
1,250	0.022105	22.5755
1,500	0.099097	38.8326
1,750	0.022232	64.7921
2,000	0.021255	103.191
2,250	0.037886	154.749
2,500	0.026206	222.234
2,750	0.030613	303.657
3,000	0.037332	403.146
3,250	0.050565	518.724
3,500	0.047130	655.168
3,750	0.078139	813.916
4,000	0.188993	993.696
4,250	0.078959	1199.66
4,500	0.106838	1432.46
4,750	0.059203	1690.80
5,000	0.128170	1977.29
5,250	0.096562	2293.64
5,500	0.114865	2646.47

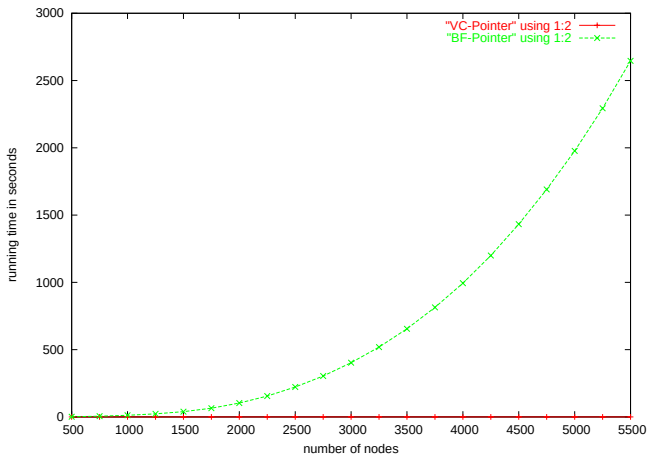

Figure 4.4: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Simple Pointer implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type B graphs.

It is easy to see from the tables and graphs that VC outperforms BF using either data structure; this is true for both types of sparse graphs that were tested. We conclude that VC is far superior to BF for sparse graphs.

4.1.5 Experimental Setup for Dense Graphs

Dense graphs were generated using the generator developed by Andrew Goldberg [CG96]. Dense graphs were defined as those with $\Omega(\frac{n^2}{8})$ edges. We generated each graph 5 times using 5 different seeds for the random number generator. The times recorded were averaged over 5 executions of each implementation.

Graphs of Type C and D were tested, with the number of vertices ranging from 125 to 1,875 in increments of 125, with small negative cost cycles and long negative cost cycles defined as in Section §4.1.3.

4.1.6 Observations

n	Array of Pointer (Time in Seconds) VC	BF
125	0.03830	0.07012
250	0.05997	0.52512
375	0.15247	1.74020
500	0.28095	4.08695
625	0.46288	7.98574
750	0.65885	13.7706
875	1.48780	21.9978
1,000	1.55311	34.5631
1,125	2.97252	51.5897
1,250	3.37425	74.6079
1,375	4.37236	100.182
1,500	7.30295	132.641
1,625	6.22661	168.864
1,750	8.63348	210.632
1,875	8.00939	260.838

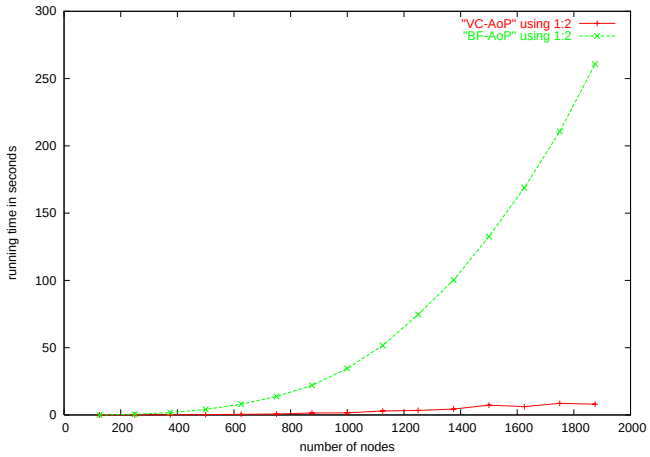

Figure 4.5: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Array of Pointer (AoP) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type C graphs.

n	Simple Pointer (Time in Seconds) VC	BF
125	0.00048	0.09019
250	0.00194	1.02020
375	0.00303	4.50625
500	0.00675	13.2450
625	0.00750	30.9083
750	0.01562	62.0953
875	0.03498	123.824
1,000	0.09200	293.591
1,125	0.11334	672.256
1,250	0.19100	1350.14
1,375	0.25657	2447.76
1,500	0.42457	4079.35
1,625	0.41033	6346.27
1,750	0.65944	9445.69
1,875	0.99798	13480.9

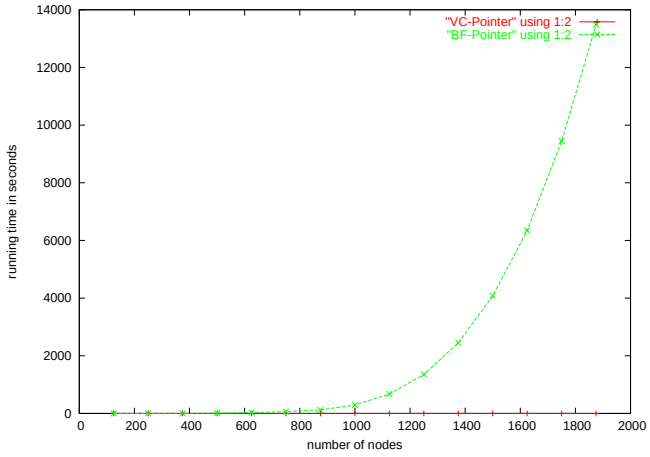

Figure 4.6: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Simple Pointer implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type C graphs.

n	Array of Pointer (Time in Seconds)	
	VC	BF
125	0.00146	0.06872
250	0.06294	0.52747
375	0.21069	1.73264
500	0.30008	4.09149
625	0.65476	7.98092
750	0.74009	13.7807
875	2.03858	21.9476
1,000	1.50686	35.0864
1,125	3.64647	51.5824
1,250	3.17255	72.8195
1,375	6.92824	101.460
1,500	6.88772	133.293
1,625	6.67336	167.244
1,750	7.48221	212.698
1,875	15.9974	263.763

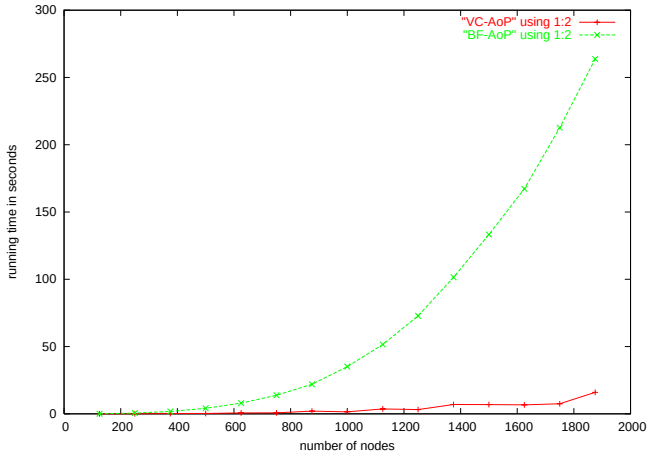

Figure 4.7: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Array of Pointer (AoP) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type D graphs.

n	Simple Pointer (Time in Seconds)	
	VC	BF
125	0.00069	0.08752
250	0.00185	1.02023
375	0.00488	4.44321
500	0.00706	13.2395
625	0.01379	30.6152
750	0.01765	62.0956
875	0.05033	122.713
1,000	0.07456	293.617
1,125	0.20491	664.932
1,250	0.22301	1348.96
1,375	0.41339	2426.95
1,500	0.39452	4079.71
1,625	1.00850	6299.06
1,750	0.59050	9447.84
1,875	1.43688	13418.7

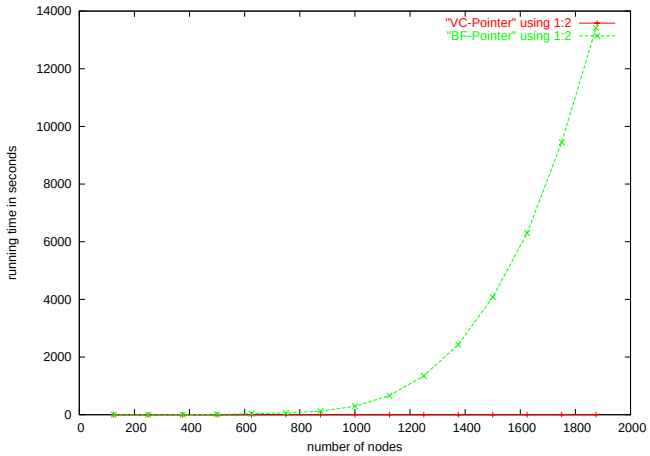

Figure 4.8: Comparison of Vertex Contraction (VC), and Bellman-Ford (BF) Simple Pointer implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type D graphs.

It is easy to see from the tables and graphs that VC outperforms BF using either data structure; this is true for both types of dense graphs that were tested. We conclude that VC is far superior to BF for dense graphs.

An asymptotic analysis would indicate that BF is superior to VC for dense graphs, although, our experiments contradict this indication.

4.1.7 Experimental Setup for Cruel Adversary Graphs

The cruel adversary is generated by specifying the number of vertices in the graph and the maximum cost of any edge. For our experiments we generated graphs with vertices ranging from 125 to 1,875 in increments of 125. We generated each graph 5 times; the times recorded were averaged over 5 executions of each implementation.

4.1.8 Observations

n	Array of Pointer (Time in Seconds)		
	VC	RVC	BF
125	0.02168	0.05125	0.04738
250	0.16735	0.40114	0.34556
375	0.55377	1.35226	1.13898
500	1.29808	3.08334	2.66123
625	2.54713	6.22039	5.19088
750	4.38616	10.7309	9.02499
875	7.01832	15.6965	14.3931
1,000	10.9170	25.7463	23.6483
1,125	15.7832	37.1957	35.7989
1,250	21.9961	52.2092	54.5719
1,375	30.1900	71.4197	72.7322
1,500	41.3305	93.0058	94.8282
1,625	53.3580	66.8170	121.354
1,750	66.5882	147.186	152.360
1,875	83.2914	171.411	188.266

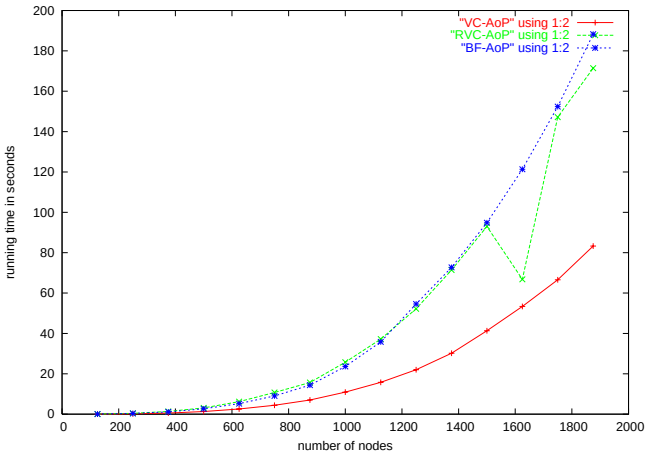

Figure 4.9: Comparison of Vertex Contraction (VC), Random Vertex Contraction (RVC) and Bellman-Ford (BF) Array of Pointer (AoP) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type E graphs.

n	Simple Pointer (Time in Seconds)		
	VC	RVC	BF
125	0.06657	0.00407	0.04572
250	0.48752	0.05800	0.33976
375	1.61851	0.05450	1.11752
500	4.55655	4.51124	2.61450
625	10.2743	10.3570	5.06600
750	19.7552	12.2059	8.70514
875	33.6204	40.3862	13.7683
1,000	52.5211	25.9912	20.4959
1,125	76.4936	50.4126	29.0852
1,250	106.248	107.372	39.9211
1,375	144.869	120.122	53.0680
1,500	187.058	206.512	69.8548
1,625	244.296	200.013	91.4537
1,750	304.338	399.716	118.606
1,875	379.341	306.991	151.796

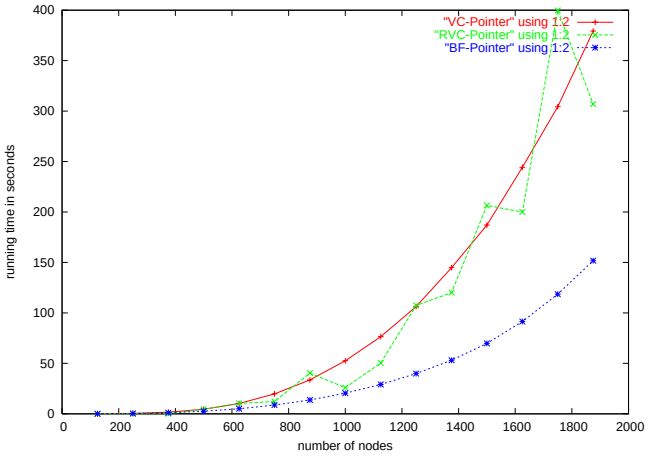

Figure 4.10: Comparison of Vertex Contraction (VC), Random Vertex Contraction (RVC) and Bellman-Ford (BF) Simple Pointer implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type E graphs.

VC does considerably better than both RVC and BF, as observed from the table and graph of the Array of Pointer implementation on Type E graphs. The results of RVC and BF are similar with RVC doing better in most instances.

VC does very poorly, as observed from the table and graph of the Simple Pointer implementation on Type E graphs. RVC does much better than VC and outperforms BF by a large margin on some instances. On other instances, RVC is either comparable to BF or much worse. One conclusion that can be drawn from the data is that the time taken by RVC varies greatly depending on the random sequence of vertices chosen.

Remark: 4.1.1 *The code for the above implementations can be downloaded from :*

`http://www.csee.wvu.edu/~lynn/msthesis/bf.html`

Chapter 5

Comparison of Vertex Contraction with More Sophisticated Algorithms

5.1 Vertex Contraction versus More Sophisticated Algorithms

In this chapter, we compare the Vertex Contraction algorithm with the Bellman Ford algorithm implemented with various heuristics and the Goldberg-Radzik algorithm. Our experiments are classified into various categories, based on the following criteria:

1. Type of input graph - Sparse with many small negative cost cycles (Type A), Sparse with a few long negative cost cycles (Type B), Dense with many small negative cost cycles (Type C), Dense with a few long negative cost cycles (Type D), the Cruel Adversary (Type E), Square Grid with many small negative cost cycles (Type F), and Square Grid with no negative cost cycles (Type G).

2. Type of Algorithm - Simple Vertex Contraction (VC), Vertex Contraction using a heap (HVC), Random Vertex Contraction (RVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC).

3. Type of Graph Data Structure - Simple Pointer or Advanced Pointer.

5.1.1 Graph Data Structures

Two different types of graph data structures were used for the experiments. We implemented VC with a simple pointer data structure and an advanced pointer data structure. The simple pointer data structure is defined the same as in Chapter 4 Section §4.1.2.

In order to decrease the amount of time taken by the simple pointer data structure, we introduced the advanced pointer data structure. For each vertex v_i, we store the in-degree d_i^{in}, the out-degree d_i^{out}, and two doubly linked lists representing the edges. Each vertex has an out list, for the edges going out of the vertex, and an in list, for the edges coming into the vertex. The out lists of each vertex are sorted based on the destination vertex of the edge. Assuming that we are contracting vertex v_i, the vertex contraction operation for the advanced pointer data structure is performed as follows.

Table 5.1: Time required to perform Vertex Contraction using an Advanced Pointer data structure.

Step	Time to Execute
1) For every edge (v_x, v_i) in v_i's in list	$O(d_i^{in})$
2) Merge v_i's *out* list with v_x's *out* list by adding those edges which are not present in v_x's *out* list and updating those that are already there, and also updating the *in* lists of the vertices appropriately	$O(d_i^{out} + d_x^{out})$
3) If a negative cost edge (v_x, v_x) is created, the algorithm terminates with the negative cost cycle being detected	$O(1)$

The time required to contract vertex v_i by the advanced pointer implementation is: $O(d_i^{in} \cdot d_i^{out} + \sum_x^{(v_x,v_i)\in E} d_x^{out})$. In the event that there exists some constant c such that, for all v_x, $d_x^{out} \leq c \cdot d_i^{out}$, then this time bound simplifies to: $O(d_i^{in} \cdot d_i^{out})$. In the worst case, $d_i^{in} = d_i^{out} = O(n)$ and hence the time complexity of the vertex contraction operation takes $O(n^2)$ time.

5.1.2 Experimental Setup for Sparse Graphs

Sparse graphs were generated using the generator developed by Andrew Goldberg [CG96], which generates multiple edges between two vertices. Sparse graphs are defined as graphs with $o(n \cdot \log n)$ edges. We generated each graph 5 times using 5 different seeds for the random number generator. The times recorded are the medians over 5 executions of each imple-

mentation.

Graphs of Type A and B were tested, with the number of vertices ranging from 500 to $10,000$ in increments of 500. We define a small negative cost cycle as one consisting of at most $\frac{n}{100}$ vertices. We define a long negative cost cycle as one consisting of $\Omega(\frac{n}{2})$ vertices. The number of long negative cost cycles in the input graph was set to 4.

5.1.3 Observations

Table 5.2: Comparison of simple Vertex Contraction (VC), Vertex Contraction using a heap (HVC), Random Vertex Contraction (RVC), Advanced Pointer Vertex Contraction (AVC), Advanced Pointer Heap Vertex Contraction (AHVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type A graphs.

	Time in Seconds									
	Simple Pointer									
n	VC	HVC	RVC	AVC	AHVC	BFFI	BFPR	BFFP	BFCT	GORC
500	0.00308	0.01090	0.00986	0.00065	0.00144	0.09281	0.05488	0.00015	0.00015	0.00003
1000	0.01007	0.10589	0.02657	0.00185	0.00874	0.41574	0.24478	0.00033	0.00037	0.00006
1500	0.01310	0.17956	0.05966	0.00167	0.01012	0.98833	0.59417	0.00054	0.00031	0.00008
2000	0.01324	0.25038	0.14488	0.00099	0.01087	1.84291	1.11845	0.00075	0.00096	0.00011
2500	0.01898	0.41555	0.24571	0.00134	0.01541	2.97744	1.81666	0.00096	0.00041	0.00014
3000	0.03527	0.84691	0.42555	0.00306	0.03322	4.40603	2.70737	0.00116	0.00101	0.00017
3500	0.04564	0.49018	0.27029	0.00381	0.01197	6.13858	3.78800	0.00138	0.00040	0.00019
4000	0.05749	0.48917	0.32360	0.00479	0.01091	8.15256	5.07085	0.00161	0.00076	0.00022
4500	0.07181	2.43284	0.56815	0.00628	0.08171	10.46086	6.55479	0.00184	0.00158	0.00023
5000	0.12887	0.46291	0.71178	0.01372	0.00945	13.05920	8.23724	0.00205	0.00139	0.00027
5500	0.08047	1.29981	1.51941	0.00554	0.02816	16.01527	10.12093	0.00231	0.00122	0.00029
6000	0.10032	3.79661	1.17647	0.00712	0.09261	19.17140	12.20301	0.00255	0.00237	0.00032
6500	0.14622	4.03073	1.26679	0.01220	0.07833	22.67825	14.50322	0.00271	0.00237	0.00035
7000	0.20369	1.47323	2.69333	0.01703	0.02535	26.49276	17.03199	0.00292	0.00217	0.00037
7500	0.17082	5.26773	1.39002	0.01070	0.08704	30.67584	19.78572	0.00320	0.00093	0.00040
8000	0.24320	4.86814	1.68994	0.01806	0.07082	35.05484	22.68988	0.00337	0.00238	0.00042
8500	0.33361	15.45075	2.26408	0.01577	0.11820	39.88285	25.84949	0.00359	0.00278	0.00044
9000	0.40964	6.90554	2.93421	0.01972	0.06642	44.99560	28.21223	0.00385	0.00178	0.00048
9500	0.48629	16.51808	5.93532	0.01994	0.13526	50.31443	30.49039	0.00403	0.00210	0.00052
10000	0.37623	28.16053	8.26182	0.00962	0.10003	55.93022	31.93657	0.00426	0.00196	0.00055

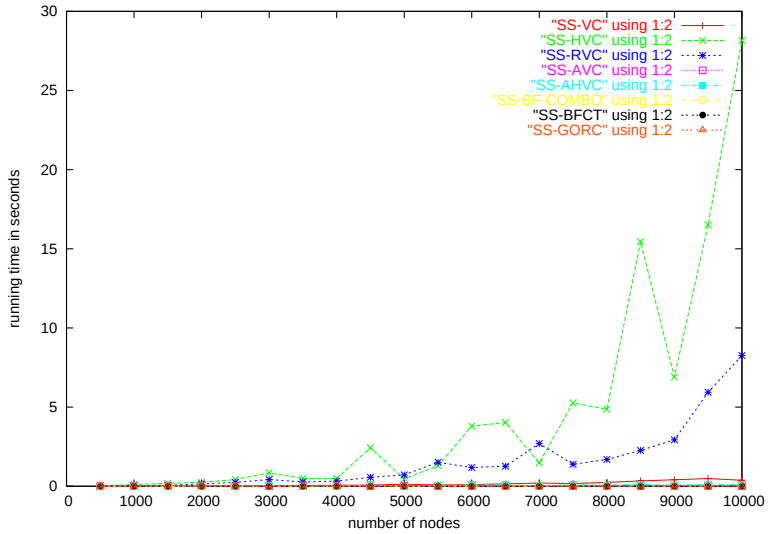

Figure 5.1: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type A graphs. Bellman Ford using a FIFO queue (BFFI) and Bellman Ford using a predecessor array (BFPR) have been omitted from the graph, since the times for these two algorithms are much worse than any of the other algorithms tested.

Table 5.3: Comparison of simple Vertex Contraction (VC), Vertex Contraction using a heap (HVC), Random Vertex Contraction (RVC), Advanced Pointer Vertex Contraction (AVC), Advanced Pointer Heap Vertex Contraction (AHVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type B graphs.

	Time in Seconds									
	Simple Pointer									
n	VC	HVC	RVC	AVC	AHVC	BFFI	BFPR	BFFP	BFCT	GORC
500	0.00307	0.01091	0.00942	0.00064	0.00144	0.09333	0.05486	0.00015	0.00010	0.00003
1000	0.01101	0.09380	0.04286	0.00228	0.00833	0.41697	0.24540	0.00033	0.00042	0.00005
1500	0.02229	0.21677	0.10963	0.00388	0.01361	0.98994	0.59466	0.00108	0.00036	0.00008
2000	0.01325	0.12521	0.14429	0.00100	0.00483	1.83873	1.11689	0.00075	0.00067	0.00011
2500	0.01878	0.41785	0.25055	0.00156	0.01533	2.97601	1.82160	0.00096	0.00044	0.00014
3000	0.03530	0.56390	0.42365	0.00303	0.01729	4.40830	2.70939	0.00117	0.00109	0.00017
3500	0.04570	0.47927	0.16870	0.00380	0.01176	6.13508	3.79725	0.00139	0.00070	0.00019
4000	0.07762	0.51539	0.52118	0.00644	0.01122	8.16938	5.07259	0.00162	0.00091	0.00023
4500	0.09704	2.58792	0.96013	0.00950	0.08610	10.45832	6.55218	0.00183	0.00168	0.00024
5000	0.12886	0.60525	0.72379	0.01363	0.01185	13.04893	8.24072	0.00205	0.00096	0.00027
5500	0.09346	1.69849	1.21267	0.00634	0.04033	15.99360	10.11421	0.00231	0.00120	0.00030
6000	0.06240	3.54915	1.05404	0.00348	0.09232	19.21575	12.21483	0.00253	0.00142	0.00033
6500	0.14794	3.17623	1.09860	0.01222	0.07068	22.68709	14.51948	0.00273	0.00212	0.00035
7000	0.19948	1.40736	2.71933	0.01782	0.02631	26.54950	17.05229	0.00291	0.00160	0.00037
7500	0.17002	7.34657	1.41677	0.01065	0.11172	30.70571	19.76453	0.00322	0.00093	0.00040
8000	0.08164	4.01609	3.75630	0.00308	0.05681	35.08038	22.72728	0.00337	0.00238	0.00042
8500	0.24130	9.53096	6.19234	0.01076	0.08516	39.92924	25.87346	0.00359	0.00375	0.00044
9000	0.42653	7.00237	2.13453	0.02033	0.06667	44.96553	28.20060	0.00384	0.00176	0.00049
9500	0.59369	11.38686	6.52555	0.02554	0.10993	50.27517	30.50643	0.00402	0.00255	0.00053
10000	0.33428	21.61870	4.07723	0.00862	0.10041	55.91048	31.93979	0.00427	0.00147	0.00056

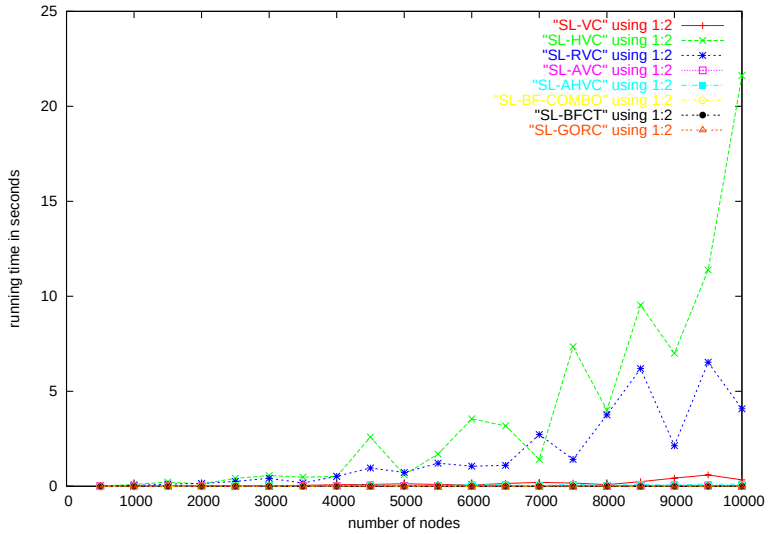

Figure 5.2: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type B graphs. Bellman Ford using a FIFO queue (BFFI) and Bellman Ford using a predecessor array (BFPR) have been omitted from the graph, since the times for these two algorithms are much worse than any of the other algorithms tested.

It is easy to see from the tables and graphs that GORC outperforms all other implementations; this is true for both types of sparse graphs that were tested. BFCT and BFFP are comparable to GORC on most instances. AVC and AHVC are far superior to BFFI and BFPR; they also outperform HVC and RVC on most instances.

5.1.4 Experimental Setup for Dense Graphs

Dense graphs were generated using the generator developed by Andrew Goldberg [CG96], which generates multiple edges between two vertices. Dense graphs were defined as those with $\Omega(\frac{n^2}{8})$ edges. We generated each graph 5 times using 5 different seeds for the random number generator. The times recorded are the medians over 5 executions of each implementation.

Graphs of Type C and D were tested, with the number of vertices ranging from 500 to $10,000$ in increments of 500, with small negative cost cycles and long negative cost cycles defined as in Section §5.1.2.

5.1.5 Observations

Table 5.4: Comparison of simple Vertex Contraction (VC), Random Vertex Contraction (RVC), Advanced Pointer Vertex Contraction (AVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type C graphs.

	Time in Seconds							
	Simple Pointer							
n	VC	RVC	AVC	BFFI	BFPR	BFFP	BFCT	GORC
500	0.00509	0.00529	0.00175	0.47843	0.37888	0.00060	0.00048	0.00003
1000	0.07894	0.18720	0.03053	3.55661	3.03432	0.00209	0.00133	0.00007
1500	0.43782	0.43558	0.11267	12.09485	10.46653	0.00474	0.00298	0.00010
2000	0.68231	0.47620	0.16673	29.69264	26.77249	0.01145	0.00398	0.00015
2500	1.53555	1.86814	0.35029	87.43734	62.06683	0.01872	0.00842	0.00025
3000	3.07647	5.02095	0.63071	170.11099	116.16429	0.02933	0.01386	0.00029
3500	2.32360	3.68301	0.51454	290.44085	194.80290	0.04222	0.01989	0.00034
4000	3.15280	8.53318	0.74046	438.88519	298.32998	0.05582	0.02406	0.00042
4500	5.64083	4.38189	1.29922	627.74945	408.21356	0.07093	0.01378	0.00044
5000	8.52792	6.05178	1.81399	832.19946	516.30401	0.08527	0.02026	0.00054
5500	12.53881	7.21655	2.69415	1,110.79003	662.60742	0.10302	0.03446	0.00058
6000	10.39215	23.57904	2.64794	1,446.03076	834.68829	0.12299	0.04281	0.00059
6500	14.29223	9.71380	3.24841	1,916.80932	1,068.52075	0.14371	0.02983	0.00067
7000	10.68957	25.26721	4.75992	2,137.90882	1,115.24783	0.17399	0.07362	0.00072
7500	24.50317	17.72326	5.09112	3,055.12915	1,595.58618	0.22184	0.09180	0.00077
8000	19.35974	24.49890	5.99822	3,559.66845	1,855.45031	0.24109	0.05681	0.00080
8500	15.33935	17.81176	7.05091	4,271.92236	2,213.34252	0.27080	0.08865	0.00087
9000	33.03027	45.71020	8.56347	5,299.29736	2,619.88574	0.28969	0.05823	0.00089
9500	22.96044	17.94311	9.99568	5,973.37793	3,055.50073	0.33669	0.04222	0.00095
10000	25.59668	35.28002	10.56971	7,275.05615	3,674.93335	0.37928	0.07394	0.00102

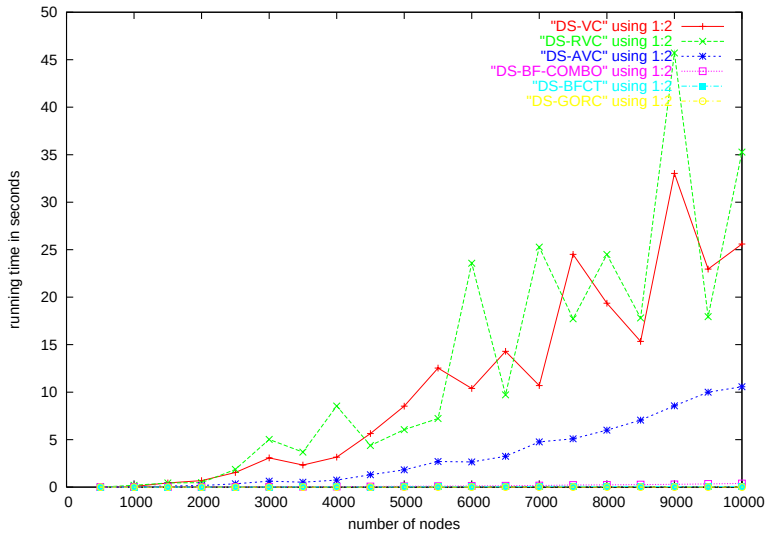

Figure 5.3: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type C graphs. Bellman Ford using a FIFO queue (BFFI) and Bellman Ford using a predecessor array (BFPR) have been omitted from the graph, since the times for these two algorithms are much worse than any of the other algorithms tested.

Table 5.5: Comparison of simple Vertex Contraction (VC), Random Vertex Contraction (RVC), Advanced Pointer Vertex Contraction (AVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type D graphs.

	Time in Seconds							
	Simple Pointer							
n	VC	RVC	AVC	BFFI	BFPR	BFFP	BFCT	GORC
500	0.00534	0.00527	0.00251	0.49948	0.31060	0.00063	0.00048	0.00003
1000	0.07196	0.19826	0.03144	3.68215	2.44459	0.00230	0.00104	0.00006
1500	0.40496	0.60226	0.09929	12.54405	8.70501	0.00527	0.00297	0.00010
2000	0.67679	0.47633	0.17866	30.88633	25.64597	0.01249	0.00497	0.00015
2500	1.55956	3.11202	0.35760	92.73985	55.91687	0.02021	0.00876	0.00026
3000	3.40649	3.01038	0.82488	181.17565	107.50185	0.03261	0.01285	0.00029
3500	3.90751	3.67845	1.05140	300.66351	176.91734	0.04514	0.00849	0.00035
4000	3.46441	7.48689	0.89778	454.59231	274.72247	0.06114	0.02418	0.00039
4500	5.96737	8.92617	1.22677	650.54907	383.21737	0.07463	0.01521	0.00045
5000	8.51730	6.32275	1.78509	894.84710	507.70788	0.09263	0.03931	0.00054
5500	12.54193	4.96325	2.99926	1,195.28356	665.74780	0.11142	0.03399	0.00058
6000	12.33178	24.99017	2.73803	1,494.68908	819.44702	0.13181	0.02126	0.00059
6500	14.24517	9.54516	3.29602	1,900.31359	1,055.81933	0.16023	0.02954	0.00066
7000	10.70251	23.92993	2.59326	2,379.41650	1,265.38305	0.18648	0.06536	0.00072
7500	20.80163	20.91272	5.23535	3,053.38183	1,597.56579	0.22259	0.09168	0.00076
8000	19.58166	19.91662	4.87426	3,707.58227	1,862.34826	0.24146	0.07080	0.00080
8500	16.08593	16.35754	3.86572	4,272.79101	2,215.66796	0.27797	0.10061	0.00087
9000	34.73608	62.04028	9.51879	5,074.77685	2,629.83178	0.29002	0.06702	0.00090
9500	45.18774	28.57104	12.35291	5,970.22851	3,170.64526	0.33432	0.04251	0.00096
10000	17.58203	37.22973	14.88489	7,271.08398	3,674.56469	0.37950	0.07865	0.00109

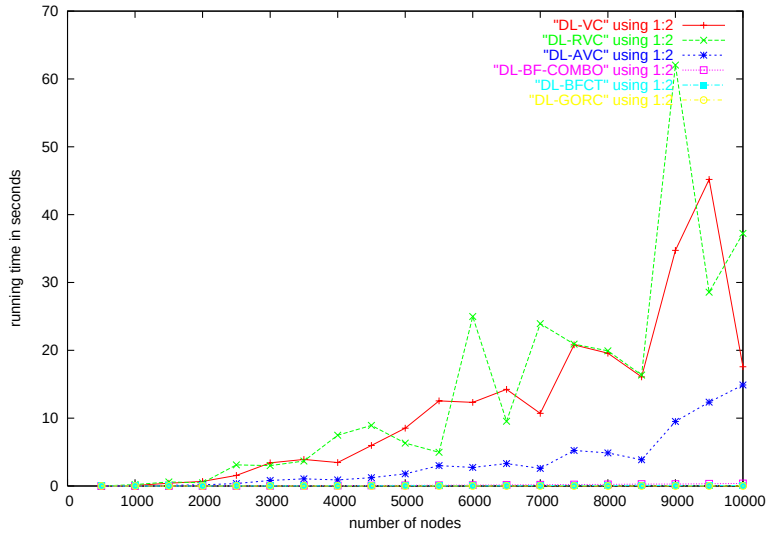

Figure 5.4: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type D graphs. Bellman Ford using a FIFO queue (BFFI) and Bellman Ford using a predecessor array (BFPR) have been omitted from the graph, since the times for these two algorithms are much worse than any of the other algorithms tested.

It is easy to see from the tables and graphs that GORC outperforms all other implementations; this is true with both types of dense graphs that were tested. BFCT performs slightly better than BFFP, although, both are comparable to GORC. AVC performs the best among the Vertex Contraction algorithms; although AVC, VC, and RVC are all far superior to BFFI and BFPR.

5.1.6 Experimental Setup for Cruel Adversary Graphs

The cruel adversary is generated by specifying the number of vertices in the graph, the maximum cost for any edge, and a seed for the random number generator.

For our experiments we generated graphs with vertices ranging from 125 to $1,875$ in increments of 125. We generated each graph 5 times; the times recorded are the medians over 5 executions of each implementation.

5.1.7 Observations

Table 5.6: Comparison of Vertex Contraction using a heap (HVC), Random Vertex Contraction (RVC), Advanced Pointer Heap Vertex Contraction (AHVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type E graphs.

	Time in Seconds							
	Simple Pointer							
n	HVC	RVC	AHVC	BFFI	BFPR	BFFP	BFCT	GORC
125	0.00219	0.00438	0.00086	0.00197	0.00113	0.00009	0.00005	0.00016
250	0.00797	0.00684	0.00181	0.00751	0.00461	0.00018	0.00010	0.00025
375	0.01749	0.01712	0.00007	0.01808	0.01046	0.00010	0.00001	0.00026
500	0.03370	0.02470	0.00626	0.04371	0.01896	0.00012	0.00001	0.00008
625	0.00016	0.03878	0.00006	0.08831	0.03007	0.00016	0.00001	0.00010
750	0.00019	0.06264	0.00006	0.13278	0.04398	0.00019	0.00002	0.00013
875	0.00022	0.08362	0.00007	0.18969	0.06315	0.00022	0.00002	0.00016
1000	0.00025	0.09760	0.00008	0.25736	0.08613	0.00025	0.00003	0.00018
1125	0.00028	0.12958	0.00009	0.33244	0.11071	0.00029	0.00003	0.00022
1250	0.00030	0.11690	0.00010	0.41327	0.13836	0.00033	0.00004	0.00025
1375	0.00034	0.14274	0.00011	0.52451	0.17114	0.00036	0.00004	0.00028
1500	0.00037	0.17026	0.00012	0.62402	0.20968	0.00042	0.00005	0.00032
1625	0.00040	0.10800	0.00013	0.74202	0.24621	0.00043	0.00005	0.00034
1750	0.00042	0.21480	0.00014	0.87620	0.28605	0.00049	0.00006	0.00037
1875	0.00111	0.26748	0.00015	1.00852	0.33392	0.00053	0.00006	0.00039

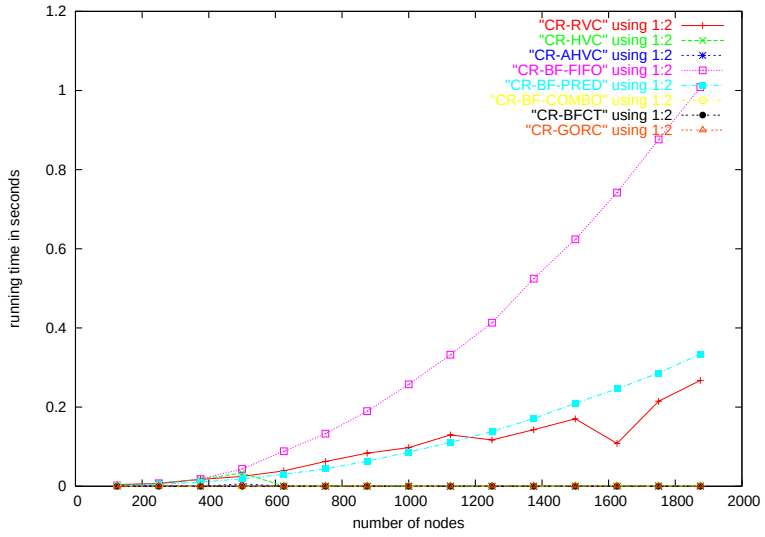

Figure 5.5: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type E graphs.

As depicted in the table and graph above, BFCT outperforms all other implementations. AHVC does very well, outperforming GORC and BFFP on most instances, although all three of these algorithms are comparable to BFCT. HVC outperforms both BFFI and BFPR.

5.1.8 Experimental Setup for Square Grid Graphs

Square Grid graphs are graphs with $\sqrt{n}$ rows and $\sqrt{n}$ columns (where n is the total number of vertices in the graph). Each vertex has at most 8 edges (4 incoming edges and 4 outgoing edges). Square Grid graphs are generated by specifying the number of vertices in each row. Square Grids with many small negative cost cycles are defined as those containing at most $\frac{n}{4}$ negative cost cycles, each consisting of 4 edges. For our experiments we generated graphs of Type F and G with vertices ranging from 100 to $11,025$. We generated each graph once in order to obtain an execution time for each implementation.

5.1.9 Observations

Table 5.7: Comparison of Simple Pointer Heap Vertex Contraction (HVC), Advanced Pointer Heap Vertex Contraction (AHVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type F graphs.

	Time in Seconds						
n	HVC	AHVC	BFFI	BFPR	BFFP	BFCT	GORC
100	0.00004	0.00001	0.00306	0.00103	0.00002	0.00001	0.00001
225	0.00007	0.00002	0.01623	0.00534	0.00005	0.00001	0.00001
400	0.00041	0.00002	0.05325	0.01732	0.00008	0.00001	0.00002
625	0.00017	0.00001	0.13380	0.04306	0.00013	0.00001	0.00003
900	0.00024	0.00001	0.30169	0.09944	0.00019	0.00002	0.00005
1225	0.00162	0.00005	0.59112	0.18502	0.00027	0.00004	0.00006
1600	0.00092	0.00002	1.01408	0.31273	0.00080	0.00005	0.00008
2025	0.00266	0.00004	1.62710	0.52423	0.00059	0.00007	0.00011
2500	0.00197	0.00003	2.46406	0.76924	0.00059	0.00073	0.00013
3025	0.00505	0.00008	3.56335	1.13797	0.00073	0.00011	0.00016
3600	0.00399	0.00003	5.16170	1.60292	0.00089	0.00013	0.00018
4225	0.00555	0.00006	7.13701	2.20785	0.00106	0.00016	0.00022
4900	0.00236	0.00002	9.57028	3.06150	0.00121	0.00017	0.00025
5625	0.00153	0.00002	12.79214	3.83176	0.00138	0.00021	0.00029
6400	0.00174	0.00002	16.26835	5.05592	0.00156	0.00024	0.00032
7225	0.00203	0.00002	20.98035	6.53380	0.00183	0.00027	0.00037
8100	0.00432	0.00003	26.09143	8.19707	0.00202	0.00030	0.00041
9025	0.01194	0.00008	32.65708	9.96410	0.00223	0.00034	0.00045
10000	0.00278	0.00003	39.98170	12.25010	0.00248	0.00036	0.00050
11025	0.01181	0.00006	48.06971	15.21093	0.00281	0.00041	0.00055

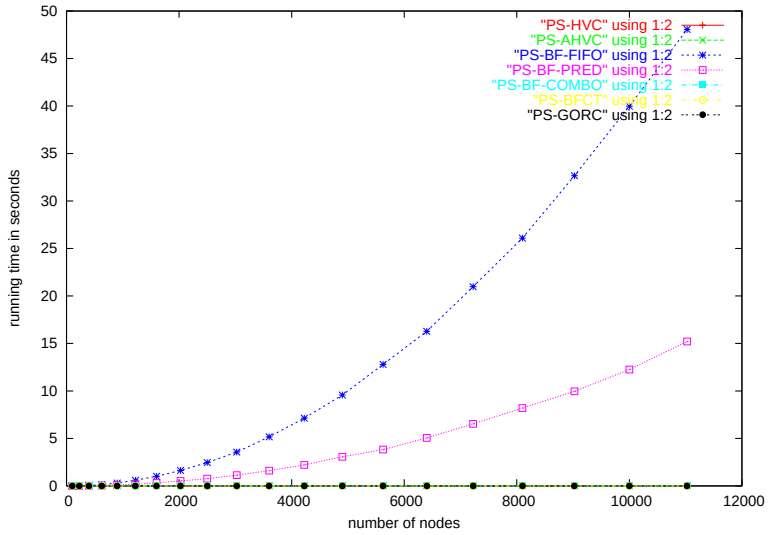

Figure 5.6: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type F graphs.

Table 5.8: Comparison of Simple Pointer Heap Vertex Contraction (HVC), Advanced Pointer Heap Vertex Contraction (AHVC), Bellman-Ford using a FIFO queue (BFFI), Bellman-Ford using a predecessor array (BFPR), Bellman-Ford using both a FIFO queue and a predecessor array (BFFP), Bellman-Ford using subtree disassembly (BFCT), and Goldberg-Radzik (GORC) implementation execution times (seconds) required to solve the Negative Cost Cycle Detection problem for Type G graphs.

	Time in Seconds						
n	HVC	AHVC	BFFI	BFPR	BFFP	BFCT	GORC
100	0.00328	0.00200	0.00004	0.00112	0.00006	0.00005	0.00011
225	0.01278	0.00643	0.00008	0.00560	0.00014	0.00011	0.00023
400	0.04615	0.01605	0.00018	0.01775	0.00031	0.00019	0.00043
625	0.11565	0.03148	0.00047	0.04340	0.00080	0.00039	0.00074
900	0.24542	0.05206	0.00063	0.09394	0.00103	0.00052	0.00105
1225	0.44884	0.08812	0.00073	0.17525	0.00117	0.00071	0.00142
1600	0.75910	0.13482	0.00155	0.30142	0.00283	0.00108	0.00198
2025	1.22930	0.19401	0.00157	0.47973	0.00301	0.00134	0.00242
2500	1.86103	0.26592	0.00301	0.73522	0.00585	0.00159	0.00292
3025	2.78646	0.35256	0.00270	1.07030	0.00486	0.00193	0.00359
3600	3.96165	0.45089	0.00380	1.52849	0.00734	0.00233	0.00424
4225	5.47559	0.64028	0.00508	2.08382	0.00916	0.00298	0.00500
4900	7.45970	1.03076	0.00549	2.83794	0.01130	0.00316	0.00655
5625	9.71863	1.33153	0.00509	3.69429	0.00913	0.00356	0.00662
6400	12.50163	2.13848	0.00937	4.83852	0.01728	0.00437	0.00767
7225	16.55281	2.80086	0.01304	6.11459	0.02366	0.00503	0.00878
8100	22.51027	4.09970	0.00967	7.74870	0.02677	0.00578	0.01476
9025	31.96791	4.88142	0.01671	9.53503	0.02971	0.00640	0.01098
10000	41.95098	6.43501	0.01704	11.82319	0.03064	0.00721	0.01216
11025	56.72657	7.65005	0.01390	14.26746	0.03356	0.00802	0.01902

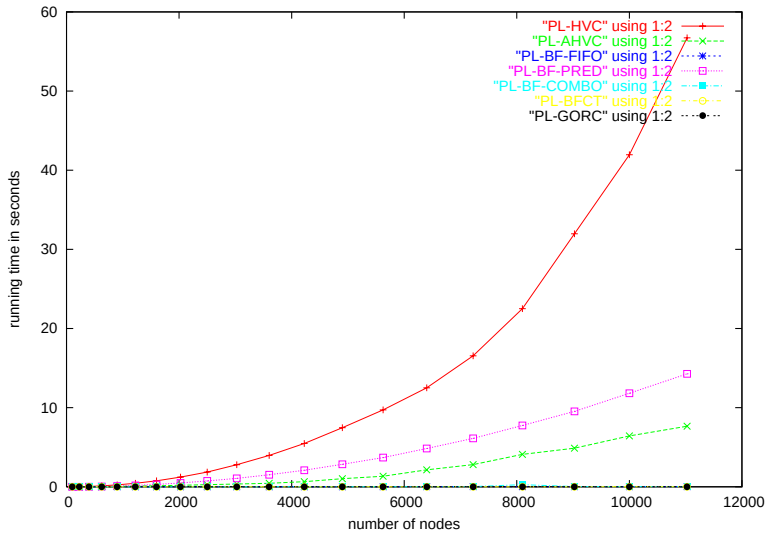

Figure 5.7: Implementation execution times required to solve the Negative Cost Cycle Detection problem for Type G graphs.

As depicted from the table and graph for Square Grid graphs with many small negative cost cycles, AHVC outperforms all other algorithms on most instances. BFCT and GORC are comparable to AHVC, while BFFI and BFPR are much worse. As depicted from the table and graph for Square Grid graphs with no negative cost cycles, BFCT outperforms all

other algorithms. GORC, BFFP, and BFFI are comparable to BFCT; AHVC outperforms both BFPR and HVC.

Remark: 5.1.1 *The code for the above implementations can be downloaded from :*

`http://www.csee.wvu.edu/~lynn/msthesis/adv.html`

Chapter 6

Conclusion

In this thesis, we designed and analyzed a greedy algorithm called the Vertex Contraction algorithm for the Negative Cost Cycle Detection problem and compared it to existing algorithms which solve the same problem.

Although vertex contraction is asymptotically inferior to the “standard” Bellman Ford algorithm on sparse graphs, empirically, it is vastly superior. Our experiments indicate that the VC algorithm outperforms the BF algorithm for both sparse graphs with many small negative cost cycles and sparse graphs with a few long negative cost cycles when implemented with either an array of pointer or simple pointer data structure. When compared with more sophisticated algorithms for both types of sparse graphs, GORC outperforms VC and all Bellman Ford algorithms implemented with various heuristics. VC implemented with an advanced pointer data structure and an advanced pointer heap data structure both outperformed BFFI and BFPR by a large margin, although, they were not comparable to GORC.

The VC algorithm is asymptotically comparable to the “standard” BF algorithm on dense graphs, although empirically it is superior. Our experiments indicate that VC outperforms the BF algorithm for both dense graphs with many small negative cost cycles and dense graphs with a few long negative cost cycles when implemented with either an array of pointer or a simple pointer data structure. When compared with more sophisticated algorithms for both types of dense graphs, GORC outperforms VC and all Bellman Ford algorithms implemented with various heuristics. VC implemented with an advanced

pointer data structure outperforms both BFFI and BFPR by a large margin, although it is inferior to GORC.

In the case of the cruel adversary graphs, the VC algorithm would be expected to be inferior to the "standard" BF algorithm, although our experiments indicate that VC outperforms BF on this graph using an array of pointer data structure. Using a simple pointer data structure, BF outperforms VC by a large margin, although instead of selecting vertices to be contracted in a well-defined order, VC can choose the next vertex to be contracted randomly without affecting the correctness of the algorithm. We call this algorithm RVC; using RVC we can achieve a better running time for cruel adversary graphs using the simple pointer data structure, although these times are still inferior to that of BF in most cases (the running time of RVC depends upon the random sequence of vertices chosen). When compared with more sophisticated algorithms for the cruel adversary graph, BFCT outperforms VC, GORC, and all other Bellman Ford algorithms implemented with various heuristics. VC implemented with an advanced pointer heap data structure is comparable to BFCT on cruel adversary graphs, outperforming GORC and BFFP on most instances.

VC implemented with an advanced pointer heap data structure outperforms the more sophisticated algorithms, for square grid graphs with many small negative cost cycles, although GORC and BFCT are comparable. BFCT outperforms VC, and all the other more sophisticated algorithms in the case of square grid graphs with no negative cost cycles.

When deciding upon an algorithm to implement, one must consider the complexity of the implementation. Clearly, VC and the "standard" BF algorithm are much simpler to implement and do not require the sophisticated data structures demanded by the more sophisticated algorithms.

When settling for a simple algorithm to implement and deciding whether to use the VC algorithm or the "standard" BF algorithm to detect negative cost cycles in a given graph, clairvoyance helps. For instance, if one knows that they are likely to be testing a large amount of graphs which fall into the cruel adversary graph category, they may choose to implement either the VC algorithm with an array of pointer data structure, or the BF algorithm with a simple pointer data structure. If space is at a minimum, the simple pointer data structure would be the one of choice, since it usually uses less space as

opposed to the quadratic space required by the array of pointer data structure. On the other hand, if one has no intuition into the types of graphs which will be tested, it would be best to choose the VC algorithm implemented with either the array of pointer or simple pointer data structure. Once again, if space is an issue, one may be inclined to choose the simple pointer data structure, otherwise the array of pointer data structure may be a better choice in case many of the graphs turn out to be cruel adversary graphs.

When running time is more important than the complexity of the implementation, clairvoyance also helps in deciding between VC and more sophisticated algorithms. For instance, if one knows that they are likely to be testing cruel adversary graphs, they may be inclined to implement either BFCT or VC with an advanced pointer heap data structure. On the other hand, if one has no intuition into the types of graphs which will be tested, it may be most beneficial to implement the GORC algorithm on account of it's overall performance on the various types of graphs. It is important to note that vertex contraction is an extremely simple strategy and does not require the sophisticated data structures demanded in implementations such as [Gol95]. When the effort to implement is considered, the VC algorithm may be viewed as superior to the more sophisticated algorithms.

Appendix A

Fourier-Motzkin Elimination

The Fourier-Motzkin Elimination procedure is an elegant, syntactic, variable elimination scheme to solve constraint systems that are comprised of linear inequalities. It was discovered initially by Fourier [Fou24] and later by Motzkin [DE73], who used it to solve general purpose linear programs.

The key idea in the elimination procedure is that a constraint system in n variables (i.e., $\Re^n$), can be projected onto a space of $n-1$ variables (i.e., $\Re^{n-1}$), without altering the solution space. In other words, polyhedral projection of a constraint set is solution preserving. This idea is applied recursively, until we are left with a single variable (say x_1). If we have $a \leq x_1 \leq b, a \leq b$, then the system is consistent, for any value of x_1 in the interval $[a, b]$. Working backwards, we can deduce the values of all the variables $x_2, \ldots, x_n$. If $a > b$, we conclude that the system is infeasible.

Algorithm (A.0.1) is a formal description of the above procedure.

Function FOURIER-MOTZKIN ELIMINATION $(\mathbf{A}, \vec{\mathbf{b}})$

1: **for** $(i = n$ **down to** $2)$ **do**
2: Let $\mathbf{I}^+ = \{$ set of constraints that can be written in the form $x_i \geq ()\}$
3: Let $\mathbf{I}^- = \{$ set of constraints that can be written in the form $x_i \leq ()\}$
4: **for** (each constraint $k \in \mathbf{I}^+$) **do**
5: **for** (each constraint $l \in \mathbf{I}^-$) **do**
6: Add $k \leq l$ to the original constraints
7: **end for**
8: **end for**
9: Delete all constraints containing x_i
10: **end for**
11: **if** $(a \leq x_1 \leq b, \quad a, b \geq 0)$ **then**
12: `Linear program is consistent`
13: **return**
14: **else**
15: `Linear program is inconsistent`
16: **return**
17: **end if**

Algorithm A.0.1: The Fourier-Motzkin Elimination procedure

Though elegant, this syntactic procedure suffers from an exponential growth in the constraint set, as it progresses. This growth has been observed both in theory [Sch87] and in practice [HLL90, LM91]. By appropriately choosing the constraint matrix $\mathbf{A}$, it can be shown that eliminating k variables causes the size of the constraint set to increase from m to $O(m^{2^k})$ [Sch87]. Algorithm (A.0.1) remains useful though as a tool for proving theorems on polyhedral spaces [VR99]. [Sch87] gives a detailed exposition of this procedure.

Bibliography

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, 1993.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. *ACM SIGPLAN Notices*, 35(5):321–333, May 2000.

[CG96] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In Josep Díaz and Maria Serna, editors, *Algorithms—ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 349–363, Barcelona, Spain, 25–27 September 1996. Springer.

[CGGV95] L. A. Costa, D. Geiger, A. Gupta, and J. Vlontozos. Dynamic programming for detecting, tracking and matching elastic contours. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1995.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 1st edition, 1992.

[CRY96] I.J. Cox, S.B. Rao, and Y.Zhong. Ration regions: A technique for image segmentation. In *Proceedings of the International Conference on Pattern Recognition*, pages 557–564. IEEE, August 1996.

[DE73] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[Fou24] J. B. J. Fourier. *Reported in: Analyse de travaux de l'Academie Royale des Sciences, pendant l'annee 1824, Partie Mathematique, Historyde l'Academie Royale de Sciences de l'Institue de France 7 (1827) xlvii-lv. (Partial English translation in: D.A. Kohler, Translation of a Report by Fourier on his work on Linear Inequalities. Opsearch 10 (1973) 38-42.)*. Academic Press, 1824.

[Gol95] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, June 1995.

[HLL90] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Fourier Algorithm Revisited. In Hélène Kirchner and W. Wechler, editors, *Proceedings Second International Conference on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 117–131, Nancy, France, October 1990. Springer-Verlag.

[LM91] Jean-Louis Lassez and Michael Maher. On fourier's algorithm for linear constraints. *Journal of Automated Reasoning, to appear*, 1991.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.

[Pug92a] W. Pugh. The definition of dependence distance. Technical Report CS-TR-2292, Dept. of Computer Science, Univ. of Maryland, College Park, November 1992.

[Pug92b] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, 35(8):102–114, August 1992.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.

[QHV02] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. *Lecture Notes in Computer Science*, 2304:325–??, 2002.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1987.

[Sub02] K. Subramani. An analysis of zero-clairvoyant scheduling. In *Proceedings of the* 8^{th} *International Conference on Tools and Algorithms for the construction of Systems*, Lecture Notes in Computer Science. Springer-Verlag, April 2002.

[VR99] V.Chandru and M.R. Rao. Linear programming. In *Algorithms and Theory of Computation Handbook, CRC Press, 1999*. CRC Press, 1999.

[WE94] Neil H. Weste and Kamran Eshragian. *Principles of CMOS VLSI Design*. Addison Wesley, 1994.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997.