Result Verification Algorithms for Optimization Problems

Himanshu Gupta*

May 12, 1995

Abstract

In this article we discuss the design of result verification algorithms for optimization problems. In particular, we design time-optimal result verification algorithms which verify the solution of all-pairs shortest paths, maximum-flow in a network, and matching problems. We prove that polynomial-time verification algorithms for NP-complete problems do not exist exist, unless $P = NP$. Result verification problems for most of the NP-hard problems are not believed to be in NP. We also consider verification algorithms for approximation algorithms for NP-complete and NP-hard problems.

1 Introduction

Consider a program A written to evaluate a function f. In this article we address the issue of being able to verify that the program A works correctly for all inputs *i.e.*, $\mathsf{A}(x) = f(x)$ for all inputs x. Most of the approaches to this problem fall under the following broad categories: formal verification, program testing/checking and result verification.

Formal verification of an algorithm or program requires a formal, rigorous proof of the correctness of the program, using the syntactic and semantic rules of the language in which the program is written. Formal verification has had limited success because even small

*Dept. of Computer Science, and Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana IL 61801. Email: `gupta@geisel.csl.uiuc.edu`. Supported by the National Science Foundation under Grant CCR-9315696.

programs are hard to verify formally. Because the technique of formal verification checks the correctness of the code as written on paper, formal verification doesn't prove the correctness of the executable code of the algorithm verified. The executable code may still be faulty because of errors in the compilation process, hardware faults, etc.

Blum *et al.* [3] introduced the theory of *self-testing/correcting*, where a *self-tester* for a program A estimates the probability that $\mathsf{A}(x) \neq f(x)$ for a random input x and a *self-corrector* for A computes $f(x)$ correctly, if the error probability of A is sufficiently low. Both self-tester and self-corrector can make calls to A, and they are considered *efficient* if their total time is linear in the running time of A. This approach has two drawbacks. Firstly, self-testers and self-correctors are both program specific. Secondly, they are allowed to make multiple calls to the original program. Hence, in most cases the total running time of the testor or corrector is more than the time of the original program itself.

Result verification, as the name suggests, involves checking the correctness of a proposed solution to an instance of the problem. A result verification algorithm for a problem Π can be used to test the output of any program solving the problem Π. Moreover, result verification algorithms do not make any calls to a program solving the original program.

In this manuscript, we discuss and design time-optimal result verification algorithms for some optimization problems. We discuss the existence of polynomial-time verification algorithms for *NP*-complete or *NP*-hard problems, and for approximation algorithms for *NP*-hard problems. The question of existence of polynomial verification algorithms becomes more interesting when we allow an algorithm to leave behind a trail of data, polynomial in length of the input, called *certification trail* [6], [7], in addition to its normal output. A result verification algorithm might use this additional information cleverly to verify the result in much less time.

The rest of the article is organized as follows. In the next section we formally define some terms and design time-optimal result verification algorithms which verify the solution of all-pairs shortes paths, maximum-flow in a network, and matching problems. In the third section we discuss the existence of polynomial-time verification algorithms for *NP*-complete and *NP*-

hard problems, and for approximation algorithms for NP-hard optimization problems. In the fourth section, we discuss result verification with the help of certification trails. Finally, we conclude in section 5.

2 Definitions

In this section we define some terms and present some examples.

Definition 2.1 A *problem* Π is formalized as a binary relation, *i.e.*, a set of ordered pairs. The domain of Π is the set of *instances*, and the range is the set of *solutions*. Thus, $\Pi \subseteq I \times S$, where I is the set of instances and S is the set of solutions for the problem. We say an *algorithm* A solves a problem Π if for all x in I, when x is input to A, a y in S is output such that $(x, y) \in \Pi$. In some cases, when each instance has a unique solution, it is possible to formalize the problem as a function.

Definition 2.2 Given a problem $\Pi \subseteq I \times S$, the *verification problem* for the problem Π is denoted by $\mathsf{V}(\Pi)$. The verification problem $\mathsf{V}(\Pi)$ is a subset of $(I \times S) \times \{\mathbf{0}, \mathbf{1}\}$ such that $((x, y), \mathbf{1}) \in \mathsf{V}(\Pi)$ if $(x, y) \in \Pi$ **and** $((x, y), \mathbf{0}) \in \mathsf{V}(\Pi)$ if $(x, y) \notin \Pi$. An algorithm solving the problem $\mathsf{V}(\Pi)$ is called a *verifier* or a *verification algorithm* for the problem Π.

Consider an algorithm A for a problem Π and a verification algorithm B solving the corresponding verification problem $\mathsf{V}(\Pi)$. By definition, if B takes as an input the ordered pair (x, y) where y is the output generated by A on the input x, then B outputs $\mathbf{1}$ if and only if A produced a correct output on the input x.

We illustrate the above definitions using the maximum clique size problem as an example. For the maximum clique size problem, Π, the set of instances is the set of graphs, and the set of solutions is the set of natural numbers; $(G, k) \in \Pi$ if k is the size of the maximum clique in G. For the verification problem, if k is the size of the largest clique in G, then $((G, k), \mathbf{1}) \in \mathsf{V}(\Pi)$; otherwise, $((G, k), \mathbf{0}) \in \mathsf{V}(\Pi)$ when k is not the size of the largest clique in G.

2.1 Verifying Network Flow Problems

In this section we discuss verification algorithms for maximum-flow and minimum-cost flow problems in networks, and bipartite and general matching problems.

2.1.1 Verifier for the Maximum-Flow Problem

Consider a directed graph G with two distinguished vertices s, the source, and t, the sink. Let the vertex set and edge set of G be $V(G)$ and $E(G)$ respectively. Let $m = |E(G)|$ and let $c(e)$ be the nonnegative maximum capacity of an edge e in $E(G)$. Let the flow in arc (x, y) be denoted by $f(x, y)$. Then an s–t flow of value v is defined by the following constraints:

$$\sum_{(u,w)\in E(G)} f(u,w) = \sum_{(w,u)\in E(G)} f(w,u), \qquad \text{for each node } u \neq s,t$$

$$\sum_{(s,w)\in E(G)} f(s,w) = \sum_{(w,s)\in E(G)} f(w,s) - v$$

$$\sum_{(t,w)\in E(G)} f(t,w) = \sum_{(w,t)\in E(G)} f(w,t) + v$$

$$f \leq c$$

$$f \geq 0$$

where $f,\ c \in \Re^m$ are the flow and the capacity vectors, respectively. A *feasible flow* is a function f on vertex pairs which satisfies the above constraints. The value of a flow f is v, the flow out of the source s. The *maximum-flow problem* is that of finding a feasible flow f of maximum value v. This problem has a rich and elegant theory and many applications both in operations research and in combinatorics [13], [12].

We need to define a few terms. A *residual capacity* for a flow f is the function on vertex pairs given by $res(v, w) = c(v, w) - f(v, w)$. The *residual graph* R_f for a flow f is the graph with vertex set $V(G)$, source s, sink t, and an edge (v, w) of capacity $res(v, w)$ for every pair v, w such that $res(v, w) > 0$. An *augmenting path* for f is a path p from s to t in R_f.

We state the following lemma without proof [12].

Lemma 2.1 *A flow f is a maximum flow if and only if there is no augmenting path for f.*

Let MF be the maximum flow problem. For the corresponding verification problem $\mathsf{V}(\mathsf{MF})$, the set of instances is $\Gamma \times F$, where Γ is the set of graphs with edge capacities, and F is the set of flows on graphs. An algorithm for $\mathsf{V}(\mathsf{MF})$ takes a graph G with a capacity vector c and a flow f as an input. It outputs $\mathbf{1}$ if and only if f is a flow with maximum value in G. According to Theorem 2.1, the algorithm outputs $\mathbf{1}$ if and only if there is no augmenting path for f. The existence of an augmenting path for f in G can be easily checked in $\mathrm{O}(m)$ time, because there exists an augmenting path in G if and only if there exists an (s,t) path in the residual graph R_f, which can be constructed in $O(m)$ time.

Theorem 2.1 *The maximum-flow result verification problem can be solved by an $O(m)$ time verification algorithm.*

2.1.2 Verifiers for Minimum-Cost Flow Problem

Let G be a network such that each edge (v,w) has a cost per unit of flow, $cost(v,w)$, in addition to a capacity $c(v,w)$. Let v_0 be a positive flow value given. The min-cost problem is to find a feasible s–t flow of value v_0 that has minimum cost, where the *cost* of a flow f is $cost(f) = \sum_{f(v,w)>0} cost(v,w)f(v,w)$.

We define the *residual graph* R_f for a flow f exactly as we did in the section 2.2.1, with the extension that $cost(v,w)$ is the same on R_f as on G.

We state the following lemma [11] without proof.

Lemma 2.2 *An s-t flow f in a network is an optimal min-cost flow if and only if there are no negative cost cycles in the residual graph R_f.*

From the above theorem, verifying the optimality of a given flow f on a given network G amounts to checking for a negative cost cycle in the residual graph R_f. Negative cycles in a weighted graph can be detected in $O(n^3)$ time, where n is the number of vertices in the graph [1].

Theorem 2.2 *There is an $O(n^3)$ verification algorithm for the minimum-cost flow verification problem.*

2.1.3 Verifiers for Matching

A *matching* M of a graph $G = (V, E)$ is a subset of edges with the property that no two edges of M share the same node. The *unweighted matching problem* is to find a maximum size matching M of an unweighted graph G. Edges in M are called *matched* edges; the other edges are *free*. Nodes that are not incident upon any matched edge are called *exposed*. A path $p = [u_1, u_2, u_3, ...u_k]$ is called *alternating* if the edges $[u_1, u_2], [u_3, u_4], ...[u_{2j-1}, u_{2j}], ...$ are free, whereas $[u_2, u_3], [u_4, u_5], ...[u_{2j}, u_{2j+1}], ...$ are matched. The alternating path p is called *augmenting* if both u_1 and u_k are exposed vertices.

Lemma 2.3 *A matching M in an unweighted graph G is maximum if and only if there is no augmenting path in G with respect to M.*

See [11] for the proof.

One is tempted to devise the analog of the max-flow algorithm for matching: Start with any matching, and repeatedly discover augmenting paths. Indeed, all known algorithms for matching are based on exactly this idea.

Bipartite matching problems can be viewed as special cases of network flow problems [13], [11].

For non-bipartite unweighted matching problems, verification of a solution involves checking for an augmenting path in the graph with respect to the matching presented. Tarjan [12] describes an algorithm to detect an augmenting path in O(m) time.

For non-bipartite weighted matching problem, the following lemma is of great import [12]. We state the theorem without proof. Here, the weight of an augmenting path is the sum of the weights of the edges on the augmenting path.

Lemma 2.4 *Let M be a matching of maximum weight among matchings of size $|M|$, let p be an augmenting path for M of maximum weight, and let M' be the matching formed by* *augmenting M using p. Then M' is of maximum weight among matching of size $|M| + 1$.*

The above theorem implies that the augmenting path method will compute maximum weight matchings of all possible sizes if we always augment using a maximum weight augmenting path. All known algorithms, use exactly this idea to solve the general weighted matching problem. Essentially, all algorithms work in stages where each stage searches for a maximum weight augmenting path. For verification purposes, a stage of such an algorithm solving the general weighted matching problem suffices to check for correctness of the matching presented. The best known algorithm for solving the weighted nonbipartite matching is by Galil, Micali, and Gabow [14] which runs in $O(nm \log n)$ time. Each result verification stage of the algorithm runs in $O(m \log n)$ time.

Theorem 2.3 *The unweighted general matching problem has a result verification algorithm which runs in $O(m)$ time. The result verification algorithm for the weighted general matching problem runs in $O(m \log n)$ time.*

2.2 Verifying All-Pairs Shortest Paths

Let G be a directed graph with n vertices and m edges, with nonnegative weights on edges. The all-pairs shortest paths problem is to find a shortest path, *i.e.*, the path with minimum weight, between each pair of vertices in G. Here, the *weight of a path* is defined as the sum of the weights of its edges. The *length of a path* is the number of edges in the path. The most widely known algorithms for the all-pairs shortest paths problem are those of Dijkstra [9] and Floyd [10]. Dijkstra's algorithm has a running time of $\Theta(mn + n^2 \log n)$ when implemented with a heap. Floyd's algorithm runs in $\Theta(n^3)$ time. Bellman and Ford [1] also developed an algorithm which runs in $\Theta(n^3)$ time and handles graphs with negative weights also.

Many algorithms for the shortest-paths problem use edge weights only to compute and compare the weights of paths. We therefore define a version [8] of the decision tree model that captures this behavior.

Definition 2.3 A *path-comparison-based* algorithm A solving the all-pairs shortest paths problem accepts as input a graph G and a weight function. The algorithm A can perform all standard operations. However, the only way it can access the edge weights is to compute or compare the weights of paths in the graph.

A verification algorithm for the all-pairs shortest paths problem takes as an input a directed graph G on n vertices, a weight function, and $n(n-1)$ paths corresponding to the pairs of vertices in G; the algorithm outputs **1** if and only if for each pair of vertices, the path presented in the input is actually a shortest path between that pair of vertices in G. Karger *et al.* [8] show that any path-comparison-based algorithm for verification of all-pairs shortest paths for *directed* graphs requires $\Omega(mn)$ path-weight comparisons. We present a path-comparison-based verification algorithm for the all-pairs shortest paths problem which runs in $O(mn)$ time. This is an improvement over the program checker of Rubinfeld [2], which runs in $O(n^3)$ time.

There are graphs with $\Theta(n^2)$ pairs of vertices whose connecting paths have lengths $\Theta(n)$ each. Hence, we cannot verify all-pairs shortest paths in $O(mn)$ time if the verification algorithm is given as an input all those shortest paths explicitly, because the size of the input itself could be $\Omega(n^3)$. Thus the verification algorithm takes as an input only a data structure from which shortest connecting paths can be constructed in time proportional to their lengths. This data structure is the *predecessor matrix* C, where for each vertex pair $i \neq j$ the entry $C[i,j]$ is a vertex k such that the edge (k,j) lies on the shortest path from i to j. In addition to C, the verification algorithm also takes the *distance matrix* D as an input, where each entry $D[i,j]$ is the weight of the shortest path from i to j. By definition, $D[i,i] = 0$.

We explain the algorithm briefly before presenting the pseudo code. The algorithm takes a weighted directed graph, and an arbitrary solution to the all-pairs shortest paths problem, *i.e.*, matrices C and D, as an input. It outputs **0** if the given solution is correct; otherwise, it outputs **1**. The weight of the edge (i,j) is denoted by $w(i,j)$. The first part of the algorithm determines for each entry $C[i,j]$ of the matrix C whether $D[i,j] = D[i,C[i,j]] + w(C[i,j],j)$. If not, then an error is detected and **0** is output. The second part of the algorithm checks for each source i and each edge $(u, v) \in E(G)$ whether using an edge (u, v) could improve the given shortest path from the source i to v. If $D[i, u] + w(u, v) < D[i, v]$, then the path from i to u with the edge (u, v) yields a shorter path from i and v and hence, an error is detected. Following the algorithm, we prove that the algorithm outputs **1** if and only if the matrices C and D presented to it are correct.

Let $\Re^+$ denote the set of nonnegative real numbers and let $w(i, j)$ denote the value of the weight function $w : E(G) \rightarrow \Re^+$ at (i, j).

<u>Algorithm 1: *Verification algorithm for all-pairs shortest paths*</u>

Input: Directed graph G, a weight function $w : E(G) \rightarrow \Re^+$, the predecessor matrix C, and the distance matrix D.

```
for i = 1 to n              /* for each source */
   for j = 1 to n           /* for each destination */
      if (j = i and D[i,j] ≠ 0)
         Return(0)
      elseif (j ≠ i)
         k := C[i,j]
         if (D[i,j] ≠ D[i,k] + w(k,j))
             /* From the definition of the predecessor matrix, the edge (k,j) */
             /* lies on the given shortest path from i to j. Hence, if C and D */
             /* are correct, then D[i,j] should equal D[i,k] + w(k,j). */
            Return(0)
         endif
       endif
    endfor
    for each edge (u,v) ∈ E(G)
```

/* Check whether the edge (u, v) could be used to improve the */
/* given shortest path from i to v */
13. **if** $(D[i,u] + w(u,v) < D[i,v])$
14. Return(**0**)
15. **endif**
16. **endfor**
17. **endfor**
18. Return(**1**)

Proof of Correctness:

Because the algorithm outputs **0** only if there is an obvious error in the input matrices C or D, Algorithm 1 outputs **1** if the input matrices C and D are correct. Conversely, we prove that if Algorithm 1 outputs **1** then the input matrices C and D are correct.

The verification algorithm verifies single-source shortest paths for source i, in the i^{th} iteration of the outermost **for** loop. Define, for each i, the function $p_i : [1..n] \rightarrow [1..n]$ as $p_i(j) = C[i,j]$. Interpretation of $p_i(j)$ as the parent of j gives a tree T_i for each i.

We claim that if for some i, j, the entry $D[i,j]$ is not equal to the distance of the vertex j from the root i in T_i, then lines 3–10 of Algorithm 1 detect an error, and it returns **0**. If some entry in the i^{th} row of the matrix D is incorrect, then there exists a vertex j such that $D[i,j]$ is not the distance of j from i in T_i. If there is more than one such vertex in T_i, then consider the one whose path from i in T_i has the fewest edges. Let that vertex be j. If $k = C[i,j]$, then the path from i to k has fewer edges than the path from i to j in T_i, and hence, by the selection of j, the entry $D[i,k]$ equals the distance of k from i in T_i. Thus, the distance from i to j in T_i should be $D[i,k] + w(k,j)$. Because the entry $D[i,j]$ is incorrect, $D[i,j] \neq D[i,k] + w(k,j)$ and hence, Line 8 returns **0**.

Finally, lines 12–16 check whether the tree T_i correctly represents the shortest path tree for the source i. We prove that if Algorithm 1 outputs **1**, then D is correct.

We prove that each row i of D is correct by showing that for every source i, the distance of a vertex j from i in T_i is the correct shortest distance of j from i, for all j. Note that by preceding discussion, $D[i,j]$ is the correct distance of the vertex j from the root i in T_i if the algorithm outputs **1**. Let $T_i^{'}$ be the correct shortest path tree for the source i. We denote the distance of a vertex j from the root i in $T_i^{'}$ by $D^{'}[i,j]$. We show that the distance of a vertex j in $V(G)$ from the root is the same in T_i and $T_i^{'}$, *i.e.,* $D[i,j] = D^{'}[i,j]$ for every j. We prove this by induction on the level of the vertex j in $T_i^{'}$, where level of a vertex in a tree is defined as the number of edges on the path from the root of the tree to that vertex. By convention, the level of the root is 0.

Basis. $Level(j) = 0$. Here, $j = i$ and the claim is obviously true.

Induction Step. $Level(j) = l$. Suppose the predecessor of j in $T_i^{'}$ is k. Because the level of the vertex k is $l-1$, the induction hypothesis implies $D^{'}[i,j] = D^{'}[i,k] + w(k,j) = D[i,k] + w(k,j)$. Because of the check made in line 13 for the edge (k,j) and the algorithm outputs **1**, $D[i,j] \leq D[i,k] + w(k,j)$. Also, because $D^{'}[i,j]$ is the correct shortest distance, $D^{'}[i,j] \leq D[i,j]$. Therefore, $D^{'}[i,j] = D[i,j]$.

Therefore, all the entries of the row i of D are correct. This implies that T_i is a correct shortest tree path for the source i.

Hence, the matrix D and the predecessor matrix C, which defines the trees, are correct.

Thus, if Algorithm 1 outputs **1**, the input matrices D and C are correct. □

It is easy to see that Algorithm 1 is a path-based-comparison algorithm and it runs in O(mn) time for a connected graph G. The algorithm can easily be parallelized to run on O(mn) processors in $O(1)$ time on an EREW PRAM machine.

Theorem 2.4 *Algorithm 1 is a* $\Theta(mn)$ *time path-based-comparison verification algorithm for the all-pairs shortest paths problem.*

3 Verifiers for Intractable Problems

In this section we discuss the existence of polynomial-time verification algorithms for *NP*-complete, and *NP*-hard problems.

3.1 Verification Problems for *NP*-complete Problems

Consider a decision problem Π: $I \to \{\mathbf{0},\ \mathbf{1}\}$. The verification problem $\mathsf{V}(\Pi)$ for the problem Π is the relation $\mathsf{V}(\Pi) \subseteq (I \times \{\mathbf{0}, \mathbf{1}\}) \times \{\mathbf{0}, \mathbf{1}\}$ such that $((x, y), \mathbf{1}) \in \mathsf{V}(\Pi)$ if and only if $(x, y) \in \Pi$. It is important to observe here that an algorithm A for a decision problem Π can be interpreted as a Turing machine accepting the language $\{x|(x, \mathbf{1}) \in \Pi\}$. The *language accepted* by an algorithm A for a decision problem Π is denoted by $L(\Pi)$.

Because there is a one-to-one correspondence between decision problems and languages accepted, we define a *problem* Π to be *NP-complete* if $L(\Pi)$, the language accepted by an algorithm for Π, is *NP*-complete. We now show that the verification problem for any *NP*-complete problem is *NP*-complete. We show this by proving that for every decision problem Π, $L(\Pi)$ is Karp-reducible to the language $L(\mathsf{V}(\Pi))$.

Reducing $L(\Pi)$ to $L(\mathsf{V}(\Pi))$

Define $f : I \to J$, where J is $I \times \{\mathbf{0}, \mathbf{1}\}$, as $f(x) = (x, 1)$. Its easy to see that $f(x) \in L(\mathsf{V}(\Pi))$ if and only if $x \in L(\Pi)$. Because f is computable in polynomial time, $L(\Pi)$ is Karp-reducible to $L(\mathsf{V}(\Pi))$.

Also, $L(\mathsf{V}(\Pi)) \in NP$, for any $L(\Pi)$ in *NP*.

Thus, we have proved the following result.

Theorem 3.1 *The verification problem* $\mathsf{V}(\Pi)$ *for any NP-complete problem* Π *is NP-complete.*

3.2 Verification for *NP*-hard Problems

Let $\aleph$ be the set of natural numbers. Consider an *NP*-hard problem $\mathsf{H} \subseteq I \times \aleph$ satisfying the following property: If $(x, y) \in H$, then $y \leq f(x)$, where the integer **value** of $f(x)$ is polynomial in size of x and the function f is computable in polynomial time. The value $f(x)$ is essentially the upper bound on the integer value of a solution of x. Let B be a verification algorithm solving the problem $\mathsf{V(H)}$. We show that a polynomial number of calls to a verification algorithm B is sufficient to solve H. Consequently, if $P \neq NP$, then a polynomial time algorithm for $\mathsf{V(H)}$ does not exist.

An algorithm for solving H *using* B

Input: x

Compute $f(x)$

for $k = 1$ **to** $f(x)$

if (B outputs **1** on the input (x, k))

Return(k)

endif

endfor.

Because the value of $f(x)$ is polynomial in the length of the input x, the *NP*-hard problem H is Cook-reducible to the verification problem $\mathsf{V(H)}$. For many graph optimization problems — minimum vertex cover size, maximum clique size, minimum clique cover size, chromatic number, maximum cycle length size, etc. — the value of the solution y for a given instance graph x is bounded above by the number of vertices in the graph. Hence, each of these *NP*-hard graph optimization problems is Cook-reducible to its corresponding verification problem.

We state the above result in the form of the following theorem.

Theorem 3.2 *Consider an NP-hard problem* $\mathsf{H} \subseteq I \times \aleph$*, such that if* $(x, y) \in \mathsf{H}$ *then* $y \leq f(x)$*, where* f *is computable in polynomial time. Also,* $f(x)$ *is polynomial in size of* x*.*

A polynomial time verification algorithm for $V(H)$ does not exist, unless $P = NP$.

Let $\Re$ be the set of real numbers. Consider an *NP*-hard optimization problem $\mathsf{H} \subseteq I \times \Re$. For each $x \in I$, x has a certain set of real *candidate solutions*. We illustrate the term, real canditate solutions, through the following examples. For the maximum clique size problem, y is a candidate solution of a graph G if G has a clique of size y. For the traveling salesman problem, y is a candidate solution of an instance G if there exists a tour of cost y in G. In contrast, a feasible solution of an instance x is a combinatorial structure or vector which is a valid solution to the problem. For example, a feasible solution of the minimum vertex coloring problem is an actual valid coloring of the vertices of the input graph. Similary, a feasible solution of the maximum clique size problem is the set of vertices which induce a clique in the input graph. Intuitively, a candidate solution is the cost of a feasible solution, and the optimization problem is to find the optimum candidate solution.

For some optimization problems, it is possible to compute the optimum feasible solution of an instance in polynomial time using an oracle which computes the optimum cost of any given instance. Such optimization problems are known as *self-reducible* problems. For example, it is an easy to design a polynomial time bounded graph coloring algorithm A, which colors a given graph with minimum number of colors (*chromatic number*), using an algorithm which returns the chromatic number of any given graph. Similarly, finding the maximum clique in a graph is also a self-reducible problem because it is easy to find a maximum clique in a graph, using an algorithm which returns the maximum clique size of any given graph. Note that for these optimization problems, it may not possible to compute the optimum solution of an instance I in polynomial time, if we are given only the optimum cost of the instance I. We should be able to query the optimum costs of the "subinstances" of I also in constant time.

H is an *NP*-hard maximization problem if for each $(x, y) \in \mathsf{H}$, y is the largest candidate solution of x. Also we define the corresponding decision problem, $\mathsf{N_H}$, for a maximization problem H as: $((x, y), \mathbf{1}) \in \mathsf{N_H}$ if $(x, z) \in \mathsf{H}$ for some $z \geq y$. Suppose H is an *NP*-hard maximization problem and the corresponding decision problem $\mathsf{N_H}$ is *NP*-complete (as is the case with many NP-hard optimization problems).

<u>Observations</u>

- $\mathsf{N_H} \subseteq (I \times \Re) \times \{\mathbf{0},\ \mathbf{1}\}$. $((x, y),\ \mathbf{1}) \in \mathsf{N_H}$ if and only if x has a candidate solution z such that $z \geq y$.
- $\mathsf{V(H)} \subseteq (I \times \Re) \times \{\mathbf{0},\ \mathbf{1}\}$. $((x, y),\ \mathbf{1}) \in \mathsf{V(H)}$ if and only if y is the maximum candidate solution of x.

Let Π be the maximum clique size problem, which is an NP-hard maximization problem. If $(G, k) \in \Pi$, then the **value** of k is bounded by the number of vertices in G. Hence, for the problem Π, $f(G) = |V(G)|$, which is polynomial in the size of G. For a graph G, y is a candidate solution of G if and only if G has a clique of size y. An algorithm solving Π outputs the maximum candidate solution of x. The corresponding decision problem for Π is known as the *Clique-Size* decision problem where $(G, k) \in$ *Clique-Size* if the graph G has a clique of size greater than or equal to k. Moreover, $(G, k) \in \mathsf{V}(\Pi)$, the verification problem for Π, if and only if k is the size of the largest clique in the graph G.

Definition 3.1 A language L is in the class DP if and only if there are two languages $L_1 \in NP$ and $L_2 \in coNP$ such that $L = L_1 \cap L_2$ [4], [5]. Obviously, $NP \subseteq DP$ and all DP-complete languages are NP-hard.

$\mathsf{V(H)}$ is the "exact cost" version of the NP-complete optimization problem $\mathsf{N_H}$. The "exact cost" versions of many of the NP-complete optimization problems known (independent set, knapsack, max-cut, max-sat, maximum clique size, etc.) have been shown to be DP-complete [4], [5]. Also, if $\mathsf{V(H)}$ is DP-complete, then $\mathsf{V(H)} \notin NP$ unless $DP = NP$.

Theorem 3.3 *Consider an NP-hard problem* H *such that the "exact-cost" version of its corresponding decision problem* $\mathsf{N_H}$ *is DP-complete. The verification problem* $\mathsf{V(H)}$ *for* H *is DP-complete and* $\mathsf{V(H)} \notin NP$ *unless* $DP = NP$.

3.3 Approximate Verifiers

Consider an *NP*-hard **maximization** problem, $\mathsf{H} \subseteq I \times \Re$, where, as defined in the previous section, $(x, z) \in \mathsf{H}$ if z is the largest candidate solution of the instance x. An algorithm for H is an (a, c) *approximation algorithm* if on input x in I, it outputs y in $\Re$ such that y is a candidate solution of x, *and* if $(x, z) \in \mathsf{H}$, then $z \leq a \cdot y + c$, where $a, c \in \Re$. Similarly, an (a, c) *approximation verifier* for the problem H is defined as the problem $\mathsf{AH} \subseteq (I \times \Re) \times \{\mathbf{0}, \mathbf{1}\}$, such that $((x, y), \mathbf{1}) \in \mathsf{AH}$ if $y \leq z \leq a \cdot y + c$, where z is such that $(x, z) \in \mathsf{H}$. Note that y need not be a candidate solution of x.

An (a, c) approximation algorithm for the maximum clique size problem would take a graph G as an input and would output y such that G has a clique of size y, and if k is the size of the largest clique of G, then $k \leq a \cdot y + c$.

We show that if there exists a polynomial-time (a, c) approximation algorithm for an *NP*-hard maximization problem H, then $L(\mathsf{N_H})$, the language accepted by an algorithm for the *NP*-complete decision problem $\mathsf{N_H}$ corresponding to H (as defined in 3.2), is Cook-reducible to $L(\mathsf{AH})$, the language accepted by the (a, c) approximation verifier for the problem H. Let AA be a polynomial-time (a, c) approximation algorithm for the problem H.

Observations

- $L(\mathsf{N_H}) \subseteq I \times \Re$. $(x, y) \in L(\mathsf{N_H})$ if and only if x has a candidate solution z such that $z \geq y$.
- $L(\mathsf{AH}) \subseteq I \times \Re$. $(x, y) \in L(\mathsf{AH})$ if and only if $y \leq z \leq a \cdot y + c$ where z is the maximum candidate solution of x.

Cook-reduction of $L(\mathsf{N_H})$ to $L(\mathsf{AH})$

Input: (x, y)

1. Call AA with the input x. Let the output be m.
2. /* Let z be the maximum candidate solution of x. Then $m \leq z \leq a \cdot m + c$ */

3. **if** $(y \leq m)$ Return($\mathbf{1}$) /* Because $y \leq m \leq z$ */

4. **if** $(y > m)$

5. **if** $(y > a \cdot m + c)$ Return($\mathbf{0}$) /* Because $z \leq a \cdot m + c < y$ */

6. **else**

7. Use the oracle for language $L(\mathsf{AH})$ to determine whether $(x, y) \in L(\mathsf{AH})$.

8. **if** $((x, y) \in L(\mathsf{AH}))$ Return($\mathbf{1}$) /* $(x, y) \in L(\mathsf{AH}) \Rightarrow y \leq z \leq a \cdot y + c.$ */

9. **else** Return($\mathbf{0}$)

10. /* $y \geq m \Rightarrow a \cdot y + c \geq a \cdot m + c$. Because z is not between y and $a \cdot y + c$ */

11. /* but is less than $a \cdot m + c$ (line 2.), $m \leq z \leq y.$ */

12. **endif**

13. **endif**

14. **endif**

The above reduction returns **1** if and only if z, the maximum candidate solution of the instance x, is greater than or equal to y. A similar reduction can be shown for *NP*-hard minimization problems.

Thus, we have proved the following result.

Theorem 3.4 *Let* H *be an NP-hard optimization problem whose corresponding decision problem,* $\mathsf{N_H}$*, is NP-complete. Also, let* H *have a polynomial-time* (a, c) *approximation algorithm. Then there is no polynomial-time algorithm for an* (a, c) *approximate verifier of* H*, unless* $P \neq NP$.

It should be noted here that given a polynomial time (a, c) approximation algorithm A, there always exists a polynomial time verification algorithm which verifies the correctness of A, *i.e.*, given an ordered pair (x, y), it checks if y is the output to be produced by fault-free A when run on the input x. The fault-free algorithm A itself would suffice as such a verification algorithm. Such verification algorithms are algorithm specific, in contrast to the algorithms solving the (a, c) approximation verifier problem. For example, for bin-packing

NP-hard minimization problem there exists a polynomial time approximation algorithm which implements the "first-fit" hueristic and is guaranteed to return a solution of value no more than $\frac{17}{10} \cdot OPT + 1$, where OPT is the optimal solution. Now, given an input x and an *arbitrary* candidate solution y, it is easy to verify whether y could be output by the "first-fit" hueristic algorithm on input x, but it is not easy to verify whether $y \leq \frac{17}{10} \cdot OPT + 1$.

4 Verification using Certification Trails

Sullivan and Masson [6, 7] introduced a conceptually novel and powerful technique to achieve fault tolerance in software systems. We consider using this technique in the context of verifiers, as defined in this article.

Let us consider a problem $\Pi \subseteq I \times S$ and an algorithm A solving Π. The algorithm A takes $x \in I$ as an input, and outputs $y \in S$ such that $(x, y) \in \Pi$. The verification problem $\mathsf{V}(\Pi)$ is a relation on $(I \times S) \times \{\mathbf{0}, \mathbf{1}\}$. A verification algorithm verifying the correctness of A takes the ordered pair (x, y) as an input, where y is the output produced by A on the input x, and outputs $\mathbf{1}$ if and only if A produced a correct output on the input x, *i.e.*, $(x, y) \in \Pi$. To make the task easier for the verification algorithm, we modify the algorithm A to $\mathsf{A_c}$ so that it now produces a trail of data, polynomial in the length of the input, which we call *certification trail*, in addition to its normal output. The verification algorithm would try to use this additional information to verify the result more quickly.

Hence, we define a certification-trail verification algorithm $\mathsf{VA_c}$ for the modified algorithm $\mathsf{A_c}$ as one which takes (x, y, c) as an input, where y and c are the output and certification trail respectively produced by the algorithm $\mathsf{A_c}$ on the input x, and outputs $\mathbf{1}$ if y and c are the correct output and certification trail generated by $\mathsf{A_c}$; otherwise, it outputs $\mathbf{0}$. Note that $\mathsf{VA_c}$ is expected to output $\mathbf{0}$ even when $\mathsf{A_c}$ outputs both y and c incorrectly on the input x. Hence, this definition differs slightly from the one proposed in [6], which allows the verification algorithm to behave erratically when both y and c are incorrect.

We illustrate the concept of certification-trail verification algorithm with the help of an example. Let us consider the single-pair shortest path problem SP. Let $\mathsf{A_{SP}}$ be an algorithm solving SP, and $\mathsf{VA_{SP}}$ be a verification algorithm for SP. The input to $\mathsf{A_{SP}}$ is a weighted graph G and a pair of vertices (s, t). The output of the algorithm is the shortest path from s to t in G. The verification algorithm $\mathsf{VA_{SP}}$ takes a weighted graph G and a path between a pair of vertices (s, t) in G as an input and it checks whether the path given is actually the shortest path from s to t. Now, if $\mathsf{A_{SP}}$ also calculates shortest paths from s to every other vertex in G (most of the known algorithms solving SP work this way), then $\mathsf{A_{SP}}$ could provide the verification algorithm $\mathsf{VA_{SP}}$ with some additional information. In this case, we require $\mathsf{A_{SP}}$ to output for each other vertex, $u \in V(G)$, its predecessor in the shortest path from s to u and the weight of the shortest path from s to u. This additional information is output as a certification trail. The verification algorithm $\mathsf{VA_{SP}}$, hence, takes a weighted graph G, the shortest path between s and t, and the certification trail consisting of predecessor and shortest distance for each other vertex, as the input. As shown in section 2.2, this information can be used to detect an error in $O(m)$ time, where m is the number of edges in G.

Next we claim that if the certification-trail verification algorithm $\mathsf{VA_c}$ runs in polynomial time, then the original verification problem $\mathsf{V}(\Pi) \in NP$. Consequently, for an NP-hard problem H whose corresponding decision problem is NP-complete, the verification problem for H does not have even a certification-trail verification algorithm which runs in polynomial time, because $\mathsf{V}(\mathsf{H})$ for most of the well known "exact cost" version optimization problems is not known (or believed) to be in NP (Refer Theorem 3.3).

To show that $\mathsf{V}(\Pi) \in NP$, we use the algorithm $\mathsf{VA_c}$ to develop a polynomial time non-deterministic algorithm $\mathsf{ND_1}$ that accepts $L(\mathsf{V}(\Pi))$. On input $(x, y), \mathsf{ND_1}$ guesses a string c, polynomial in the length of (x, y), and calls $\mathsf{VA_c}$ with the input (x, y, c). If $\mathsf{VA_c}$ outputs $\mathbf{0}$, then $\mathsf{ND_1}$ loops forever; otherwise, $\mathsf{ND_1}$ outputs $\mathbf{1}$ when $\mathsf{VA_c}$ outputs $\mathbf{1}$. To see that $\mathsf{ND_1}$ accepts $L(\mathsf{V}(\Pi))$, we observe that $\mathsf{ND_1}$ accepts an input (x, y) if and only if there exists a string c such that $\mathsf{VA_c}$ outputs $\mathbf{1}$ on (x, y, c) if and only if $(x, y) \in \mathsf{V}(\Pi)$. Hence, $\mathsf{V}(\Pi) \in NP$.

Thus, we have proved the following theorem.

Theorem 4.1 *If there exists a polynomial time certification-trail verification algorithm VA_c for an algorithm solving the problem Π, then the verification problem $V(\Pi) \in NP$.*

5 Conclusions

We have shown that verification problems for NP-complete problems are NP-complete. Verification problems for many the well known NP-hard maximization problems whose corresponding decision problems are NP-complete have been proved to be DP-complete and NP-hard. Furthermore, (a, c) approximate verifiers of NP-hard languages having polynomial time (a, c) approximation algorithms cannot have polynomial time algorithms unless $P = NP$.

We have also discussed the existence of polynomial time verification algorithms when they are presented with some additional information (certification trail). We proved that if a certification-trail verification algorithm $\mathsf{VA_c}$ exists for an algorithm A solving the problem Π, then $\mathsf{VA_c} \in NP$.

The verification algorithms discussed in the section 2.1 belong to the class of primal-dual algorithms [11], which is a general framework for solving linear programming. A primal-dual algorithm works in stages, where each stages checks for the optimality of the current solution. Hence, each stage of a primal-dual algorithm is essentially a result verification stage. This leaves us with the following open question. Is there a broader characterization of algorithms consisting of stages whose each stage can be used as a result verification stage.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press. Mc Graw Hill Book Company.

[2] R. A. Rubinfeld. *A Mathematical Theory of Self-Checking, Self-Testing, and Self Correcting Programs.* Ph.D. Thesis, Computer Science Department, University of California, Berkeley. 1990.

[3] M. Blum, M. Luby, and R. A. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Sciences*, **47**, 549–595, 1993.

[4] C. H. Papadimitriou and M. Yannakakis. The Complexity of Facets (and some facets of complexity). *Journal of Computer Sciences and Systems*, **28**, 244–259, 1984.

[5] C. H. Papadimitriou and D. Wolfe. The Complexity of Facets Resolved. *Journal of Computer Sciences and Systems*, **37**, 2–13, 1987.

[6] G. F. Sullivan and G. M. Masson. Using Certification Trails to Achieve Software Fault Tolerance. *Digest of the 1990 Fault Tolerant Computing Symposium*, 423–431, 1990.

[7] G. F. Sullivan and G. M. Masson. Certification Trails for Data Structures. *Digest of the 1991 Fault Tolerant Computing Symposium*, 240–247, 1991.

[8] D. R. Karger, D. Koller, and S. J. Phillips. Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths. *SIAM Journal of Computing*, **22**, 1199–1217, December, 1993.

[9] E. W. Dijkstra. A Note on Two Problems in Connextion with Graphs. *Numerische Mathematik*, **1**, 269–271, 1959.

[10] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, **5**, 345, 1962.

[11] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Printice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[12] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, Pennsylvania, 1983.

[13] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[14] Z. Galil, S. Micali, and H. Gabow. Maximal weighted matching on general graphs. *Proceedings of 23rd Annual IEEE Symposium on Foundations of Computer Science*, 255–261, 1982.