Technical Report No. 2000-441

Real-Time Minimum Vertex Cover For Two-Terminal Series-Parallel Graphs[*]

Marius Nagy and Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada
Email: {marius,akl}@cs.queensu.ca

October 6, 2000

Abstract

Tree contraction is a powerful technique for solving a large number of graph problems on families of recursively definable graphs. The method is based on processing the parse tree associated with a member of such a family of graphs in a bottom-up fashion, such that the solution to the problem is obtained at the root of the tree. Sequentially, this can be done in linear time with respect to the size of the input graph. In parallel, efficient and even cost optimal tree contraction algorithms have also been developed. In this paper we show how the method can be applied to compute the cardinality of the minimum vertex cover of a two-terminal series-parallel graph. We then construct a real-time paradigm for this problem and show that in the new computational environment, a parallel algorithm is superior to the best possible sequential algorithm, in terms of the accuracy of the solution computed. Specifically, there are cases in which the solution produced by a parallel algorithm that uses p processors is better than the output of any sequential algorithm for this problem, by a factor superlinear in p.

1 Introduction

Many real-life applications can be abstracted to graph theoretical problems. That is why studying graph problems is of capital importance for the theory of computation. Unfortunately, many of these problems appear to be not solvable efficiently, as they were proved to be NP-complete. However, if the input to these problems is restricted to some particular

[*]This research was supported by the Natural Sciences and Engineering Research Council of Canada.

families of graphs, then a linear time solution (with respect to the size of the input graph) can be derived. Several independent results were obtained along these lines. Eventually, a theory was developed, that explained the linearity of the previously published algorithms, predicted the existence of several thousand new linear time algorithms of this kind and provided a methodology for constructing any one of them [24].

The problem addressed in this paper is that of finding the cardinality of a minimum vertex cover, in the particular case of a two-terminal series-parallel graph. Furthermore, we define the problem in a real-time environment and compare the performance of a parallel algorithm to that of a sequential one by evaluating the accuracy of the solutions computed in both cases. We begin by describing some previous results that will help us build our parallel algorithm for solving the above stated problem. This is followed by an outline of the results obtained in this paper.

1.1 Previous work

The framework elaborated in [24] can be applied to any family of graphs which can be defined recursively by certain rules of composition involving finite sets of terminals. The construction process of a member of such a recursive family of graphs can be modeled as a tree. The leaves of the tree represent base graphs of the family, while the internal nodes depict the composition rules applied. Such a structure is called a *parse tree*, or a *decomposition tree*.

In order to obtain a solution whose running time is linear in the size of the input graph, an algorithm must specify a finite set of recurrence relations for the problem being solved and proceed in a bottom-up fashion through the parse tree associated with the input graph. The solution to the problem addressed is obtained at the root of the decomposition tree, as a result of gradually reducing the tree to its root. For this reason the computational process is called *tree contraction* and its immediate application is the evaluation of arithmetic expressions encoded as binary trees. In fact, finding the solution to a graph problem, using the methodology developed in [24] corresponds to the evaluation of an expression in a particular kind of algebra.

In what follows, and throughout the paper, we assume that the input graph has N vertices. The number of edges of a two-terminal series-parallel graph is on the order of the number of its vertices. The parse tree of such a graph has as many leaves as there are edges in the original graph. As a consequence, the number of leaves in the parse tree is $O(N)$, and furthermore, the number of nodes in the parse tree is also on the order of N. We also make a distinction between *nodes* and *vertices* throughout the paper. We speak of nodes in the context of a parse tree, while vertices will always refer to the original series-parallel graph.

Because of its widespread applicability, developing an efficient parallel tree contraction algorithm was very important. Miller and Reif [19] describe a deterministic algorithm which runs in $O(\log N)$ time with $O(N)$ Exclusive-Read Exclusive-Write Parallel Random Access Machine (EREW PRAM) [2] processors. A single step of their algorithm uses two basic operations to reduce the current binary tree. The RAKE operation removes in parallel all leaves, while the COMPRESS operation is responsible for compressing maximal chains of nodes with only one child. These chains may appear as a consequence of applying the RAKE operation. It is shown in [19] that after $O(\log N)$ such steps a given tree is reduced to its root.

Since the original paper of Miller and Reif [19], a series of improved tree contraction algorithms have been developed. Gazit et al. [10], Kosaraju and Delcher [15], and Abrahamson et al. [1] were all able to develop optimal, deterministic parallel algorithms for tree contraction on the EREW PRAM model, running in $O(\log N)$ time and using only $O(N/\log N)$ processors. These results provide efficient parallel algorithms for a large class of graph problems when the underlying graph is a member of a recursively definable family of graphs. He [11] applied the tree contraction scheme to solve the binary tree algebraic computation (BTAC) problem. It is then shown in [11] how several optimization problems for trees, such as for example, minimum covering set, maximum independent set and maximum matching, can be converted into instances of the BTAC problem. Abrahamson et al. [1] generalized the BTAC problem by defining a bottom-up algebraic tree computation (B-ATC) problem, as well as a top-down algebraic tree computation (T-ATC) problem. Under some restrictions (taking the form of some closure properties of certain classes of unary functions) both of these two problems can be solved within the cost of tree contraction. Abrahamson et al. [1] showed how a number of problems (e.g., largest clique and largest independent set) which are very difficult for general graphs, can be solved in logarithmic parallel time, if applied to a cograph represented as a parse tree, by reducing them to instances of a B-ATC or T-ATC problem. Lin and Olariu [17] developed an optimal parallel matching algorithm for cographs, a result that was extended for some other classes of graphs by Parfenoff [18].

He [12] investigated the possibility of developing optimal parallel algorithms for another family of graphs: two-terminal series-parallel graphs. Using the already well established methodology based on tree contraction, He was able to construct linear cost parallel algorithms for solving three problems: 3-coloring, depth-first spanning tree and breadth-first spanning tree, under the assumption that the input is given by the decomposition tree. Nevertheless, our intuition is that the parallel tree contraction technique can be successfully applied to the whole list of families of graphs and problems presented in [24].

In order to solve a graph problem using the tree contraction scheme, one must construct an algebra in such a way that the evaluation of the expression encoded in the parse tree of the input graph should lead to the solution of the problem. Usually, if the defined algebra does not have a finite carrier, then some conditions must be fulfilled in order to evaluate the parse tree by applying a parallel tree contraction algorithm. For more details about these conditions, see [1].

1.2 Contributions of this paper

Takamizawa et al. [21] showed in a unified manner that for many combinatorial problems on graphs, including minimum vertex cover, there exist algorithms whose running time is linear in the size of the input, if the graphs under consideration are restricted to the class of series-parallel graphs. In this paper we show how an appropriate algebra can be defined for finding the cardinality of the minimum vertex cover of a two-terminal series-parallel graph. Using a tree contraction algorithm next, the problem can be solved in linear sequential time, or in logarithmic parallel time if $O(N/\log N)$ EREW PRAM processors are employed.

We are also interested in comparing the performances of a parallel algorithm with the

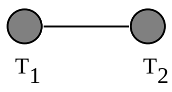

Figure 1: The base graph of the TTSP family of graphs.

best possible sequential one in a *real-time environment*. For this purpose, we define a real-time paradigm for computing the cardinality of the minimum vertex cover in which the input data, represented by the parse tree of a two-terminal series-parallel graph, is divided into chunks and fed to the algorithm at regular time intervals. We also impose deadlines on when the newly arrived data should be processed and incorporated into the solution computed so far. The kind of real-time paradigm constructed in this paper is similar to the one defined in [4] for another optimization problem in graph theory, namely the minimum spanning tree.

The parallel algorithm will always be able to compute the exact cardinality of the minimum vertex cover for the input graph. By contrast, any sequential algorithm is forced, due to the limited time, to compute an approximate solution. We show that in the worst case, the ratio of the cardinality of the minimum vertex cover obtained sequentially to the one computed in parallel can be superlinear in the number of processors used by the parallel algorithm.

The remainder of this paper is organized as follows. Section 2 is intended to make the reader familiar with the class of two-terminal series-parallel (TTSP) graphs. Section 3 states the variant of the minimum vertex cover problem addressed in this paper and shows how it can be solved using the general framework developed in [24]. In section 4 we construct a real-time environment for the problem and derive two algorithms, one sequential and one parallel, for its solution. An analysis of the performances of the two algorithms is described in section 5. Finally, some conclusions and suggestions for future investigations are presented in section 6.

2 Two-terminal series-parallel (TTSP) graphs

The class of such graphs, which is a subclass of planar graphs, is a well-known model of series-parallel electrical networks and various scheduling problems. A two-terminal series-parallel graph can be constructed from a given graph by recursively applying *series* and *parallel* connections.

Each graph belonging to this recursive family of graphs has two vertices that play a special role in the composition rules, and therefore they are called *terminal* vertices. The family has only one *base graph*, consisting of two vertices connected by an edge (Figure 1).

Starting from this base graph, any member of the family can be constructed recursively from two smaller members using one of the composition rules described below.

Let G_1 and G_2 be two TTSP graphs.

Rule 1 (*Series connection*): Identify a terminal of G_1 with a terminal of G_2 and choose the other terminals of G_1 and G_2 as the new terminals (Figure 2).

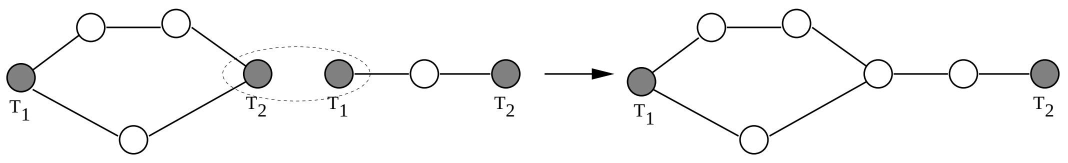

Figure 2: Example of a series connection.

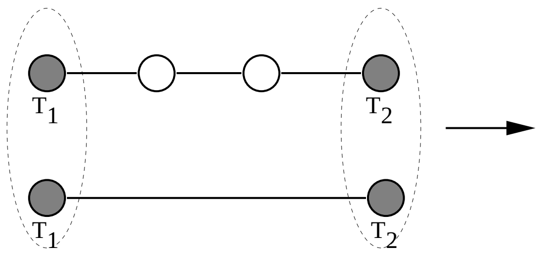

Figure 3: Example of a parallel connection.

Rule 2 (*Parallel connection*): Identify one terminal of G_1 with one terminal of G_2 and also identify the other pair of terminals. Use the resulting two vertices as the terminals and delete any redundant edges (Figures 3 and 4).

The parse tree of a series parallel graph gives us an intuitive image of how the graph was constructed and therefore it is also called a decomposition tree. In Figure 5 a TTSP graph is shown together with its parse tree. This is a binary tree in which the leaves represent distinct instances of the base graph (BG) and the internal nodes denote either a series (S) or a parallel (P) connection.

Both sequential and parallel algorithms for recognizing a series-parallel graph and obtaining its decomposition tree exist in the literature [13, 23].

3 The minimum vertex cover problem

Let $G = (V, E)$ be a graph under the usual notation convention, where V represents the set of vertices and E the set of edges. A *covering set* of G is a subset $C \subseteq V$ such that for any edge $(i, j) \in E$, $\{i, j\} \cap C \neq \emptyset$. The *minimum vertex cover* problem, also known as

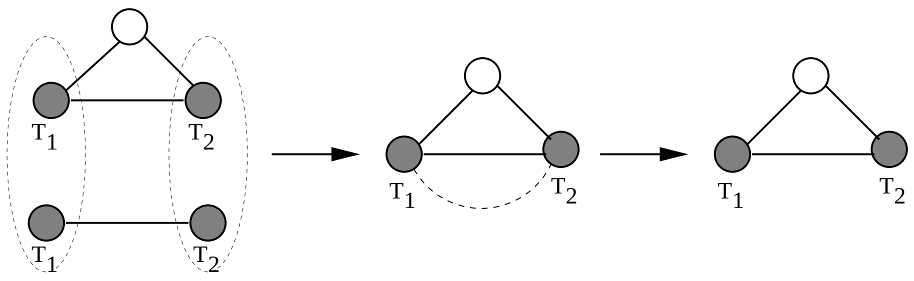

Figure 4: Redundant edge deleted after a parallel connection.

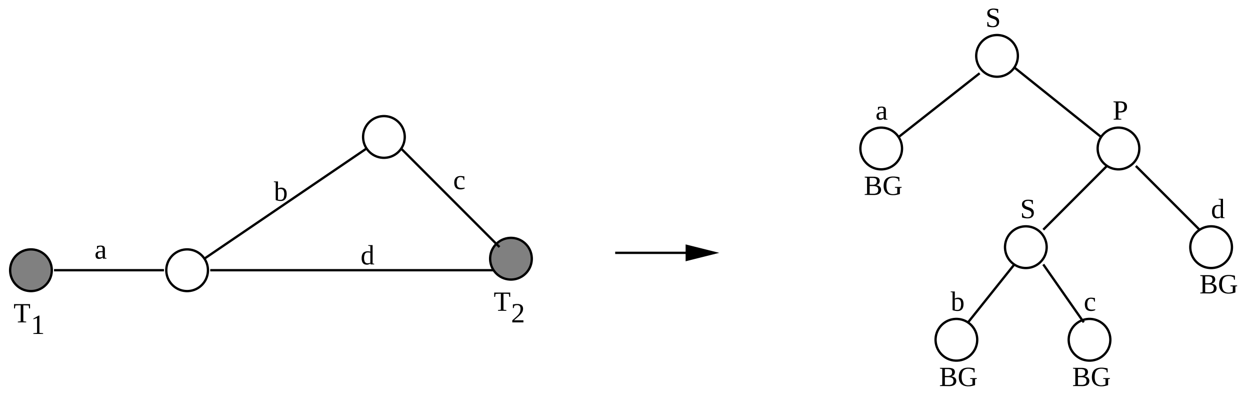

Figure 5: A TTSP graph and its decomposition tree.

the minimum covering set problem, is to find a covering set of minimum cardinality. For an arbitrary graph, this is an NP-complete problem [9]. In our case, however, the input to the problem is a TTSP graph and we are only interested to find the cardinality of the minimum covering set, or the best possible approximation of it. Under these restrictions the problem can be solved in sequential linear time (see [21] for example).

In our variant, a binary tree, representing the decomposition tree of a TTSP graph, is the input to the problem. Obtaining the decomposition tree from the original TTSP graph is not the task of the real-time algorithm. This is obtained by a preprocessing step, using any one of the existing algorithms [13, 23].

Let $G = (V, E)$ be a TTSP graph with terminals T_1 and T_2. The cardinality of the minimum covering set of G is computed at the root of the parse tree, after processing the tree in a bottom-up fashion. The result of this computation is a four-element integer vector X, whose components are detailed in the following.

$X(1)$ = cardinality of the minimum vertex cover of G containing both terminals.
$X(2)$ = cardinality of the minimum vertex cover of G containing T_1, but not T_2.
$X(3)$ = cardinality of the minimum vertex cover of G containing T_2, but not T_1.
$X(4)$ = cardinality of the minimum vertex cover of G containing none of the terminals.

Obviously, one of these four values must be the cardinality of the minimum covering set of G. If we were able to compute the values of the four elements forming the vector associated with the root of the parse tree corresponding to G, then their minimum would represent the cardinality of the minimum vertex cover of the original TTSP graph G.

In a similar way, a four-integer vector is associated with each of the nodes in the decomposition tree. For an arbitrary node v, the four integers of the associated vector refer to the subgraph of G whose decomposition tree is rooted at v. If v is a leaf in the parse tree, then the values of the four components of the associated vector X can be immediately specified (as detailed at the end of this section) because they refer to the base graph of the TTSP family (Figure 1). On the other hand, if v is an internal node, with children u_1 and u_2 and associated vectors X_1 and X_2 respectively, then the vector X of v is computed as a function of X_1 and X_2. The function differs, depending on whether the node v represents a series or

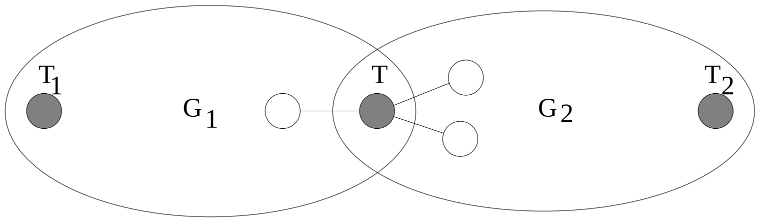

Figure 6: A series connection between G_1 and G_2.

a parallel connection.

The remainder of this section is dedicated to deriving and proving correct the function used to compute X. This will allow the vector associated with the root of the parse tree to be computed from the vectors associated with the leaves, in a bottom-up fashion, by applying the appropriate function to each internal node of the parse tree. Finally, choosing the minimum among the four components of the vector at the root of the parse tree will produce the cardinality of the minimum covering set of the original TTSP graph G.

For the ease of presentation, we adopt the following notation:

$mcs(G)$ = the minimum covering set of graph G (although it might not be unique, we will stick with this notation for convenience).

$|mcs(G)|$ = the cardinality of the minimum covering set of graph G.

$mcs^{+v}(G)$ = the minimum covering set of graph G which includes vertex v, $v \in V$.

$mcs^{-v}(G)$ = the minimum covering set of graph G which does not include vertex v, $v \in V$.

$mcs^{+v_1,-v_2}(G)$ = the minimum covering set of graph G which includes vertex v_1 but does not include vertex v_2, $v_1, v_2 \in V$.

Now suppose $G = (V, E)$ was constructed from $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ through a series or parallel connection. Let X_1 and X_2 be the labels associated with the root of the decomposition tree of G_1 and G_2, respectively. The problem is to compute the four elements of the vector associated with the root of the parse tree of G, as a function of X_1 and X_2, in the case of a series and of a parallel connection.

3.1 Series connection

G_1 and G_2 are connected through a single vertex T (Figure 6). Therefore T is called an articulation point.

Proposition 1 *$mcs(G) \bigcap V_1$ is a covering set for G_1, and similarly, $mcs(G) \bigcap V_2$ is a covering set for G_2.*

Proof
Suppose that $mcs(G) \bigcap V_1$ is not a covering set for G_1. Then there must be an edge $e = (v_i, v_j) \in E_1$ such that

$$\{v_i, v_j\} \bigcap (mcs(G) \bigcap V_1) = \emptyset. \tag{1}$$

But $v_i \in V_1$ and $v_j \in V_1$, so the last equality can be rewritten as

$$\{v_i, v_j\} \bigcap mcs(G) = \emptyset. \tag{2}$$

On the other hand, $E_1 \subset E$, which means that

$$(v_i, v_j) \in E. \tag{3}$$

From equations 2 and 3 we can conclude that $mcs(G)$ is not a covering set for G, which is obviously a contradiction. This means that our hypothesis is false and that $mcs(G) \bigcap V_1$ is indeed a covering set for G_1. □

In our effort to construct $mcs(G)$ from $mcs(G_1)$ and $mcs(G_2)$, the articulation point T plays a key role. There are only two possibilities: either T belongs to $mcs(G)$ or it does not. Therefore, we compute the cardinality of $mcs(G)$ in both of these cases and then choose the minimum of the two:

$$|mcs(G)| = min(mcs^{+T}(G), mcs^{-T}(G)). \tag{4}$$

Proposition 2 *$mcs^{+T}(G) \bigcap V_1 = mcs^{+T}(G_1)$, and similarly $mcs^{+T}(G) \bigcap V_2 = mcs^{+T}(G_2)$.*

Proof
According to Proposition 1, $mcs^{+T}(G) \bigcap V_1$ is a covering set for G_1. It remains to show that this covering set has the smallest cardinality among all covering sets containing T.

The only vertex belonging to both G_1 and G_2 is T. Therefore $mcs^{+T}(G)$ can be expressed as the union of two disjoint sets as follows.

$$mcs^{+T}(G) = S_1 \bigcup S_2,$$

where $S_1 = mcs^{+T}(G) \bigcap V_1$, and $S_2 = mcs^{+T}(G) \bigcap V_2 - \{T\}$.

Suppose there exist a covering set for G_1, containing T, which has a smaller cardinality than S_1. Let us call this set $mcs^*(G_1)$. Our claim is that $mcs^*(G_1) \bigcup S_2$ is a covering set for G. This is quite obvious, since $mcs^*(G_1)$ covers all the edges in E_1, and $S_2 \bigcup \{T\}$ covers all the edges in E_2. But since $|mcs^*(G_1)| < |S_1|$, it follows that

$$|mcs^*(G_1) \bigcup S_2| < |S_1 \bigcup S_2|,$$

because $mcs^*(G_1) \bigcap S_2 = \emptyset$ and $S_1 \bigcap S_2 = \emptyset$. The last inequality can be rewritten as

$$|mcs^*(G_1) \bigcup S_2| < |mcs^{+T}(G)|,$$

meaning that $mcs^*(G_1) \bigcup S_2$ is a covering set for G, which contains T, and is smaller than $mcs^{+T}(G)$. But this is absurd, and consequently our assumption must have been false. Therefore $mcs^{+T}(G) \bigcap V_1$ is a minimum covering set of G_1 containing T. □

The immediate consequence of this proposition is that

$$|mcs^{+T}(G)| = |mcs^{+T}(G_1)| + |mcs^{+T}(G_2)| - 1. \tag{5}$$

due to the fact that $mcs^{+T}(G_1) \bigcap mcs^{+T}(G_2) = \{T\}$.

In a very similar way, we can obtain the following result.

Proposition 3 *$mcs^{-T}(G) \bigcap V_1 = mcs^{-T}(G_1)$, and $mcs^{-T}(G) \bigcap V_2 = mcs^{-T}(G_2)$.*

From Proposition 3 and the observation that $mcs^{-T}(G_1) \bigcap mcs^{-T}(G_2) = \emptyset$ we can immediately deduce that

$$|mcs^{-T}(G)| = |mcs^{-T}(G_1)| + |mcs^{-T}(G_2)|. \tag{6}$$

Substituting equations 5 and 6 in 4 we obtain

$$|mcs(G)| = min(|mcs^{+T}(G_1)| + |mcs^{+T}(G_2)| - 1, |mcs^{-T}(G_1)| + |mcs^{-T}(G_2)|).$$

Finally, if we rewrite this general formula in each of the four particular cases representing the components of the vector, we obtain the function associated with a series connection, namely:

$X(1) = min(X_1(1) + X_2(1) - 1, X_1(2) + X_2(3)),$
$X(2) = min(X_1(1) + X_2(2) - 1, X_1(2) + X_2(4)),$
$X(3) = min(X_1(3) + X_2(1) - 1, X_1(4) + X_2(3)),$
$X(4) = min(X_1(3) + X_2(2) - 1, X_1(4) + X_2(4)).$

3.2 Parallel connection

In the case of a parallel connection (Figure 7), T_1 and T_2 form an articulation set. Due to the fact that the articulation set is composed of the two terminal vertices of the newly formed graph G, the recurrence formulas will be simpler than those involved in a series connection. In deriving these formulas, the following four propositions, one for each component of the vector, are very useful. The proofs for these propositions are omitted because of their obvious similarity with Proposition 2 and 3.

If the minimum vertex cover of G contains both terminals, T_1 and T_2, we have:

Proposition 4 *$mcs^{+T_1,+T_2}(G) \bigcap V_1 = mcs^{+T_1,+T_2}(G_1)$,
and $mcs^{+T_1,+T_2}(G) \bigcap V_2 = mcs^{+T_1,+T_2}(G_2)$.*

If the minimum vertex cover of G contains T_1 but not T_2, we have:

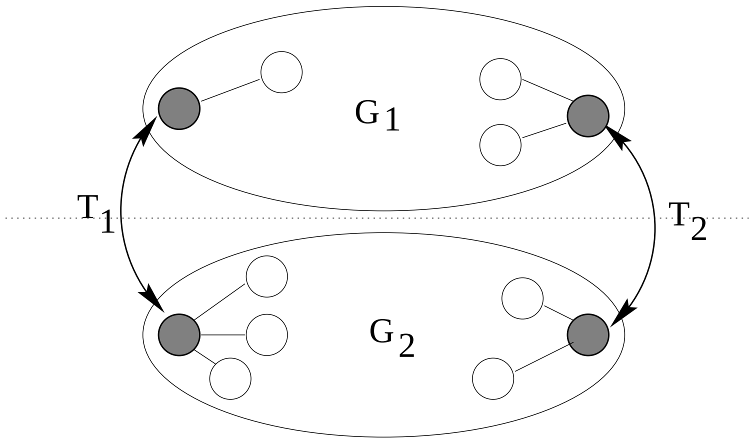

Figure 7: A parallel connection between G_1 and G_2.

Proposition 5 *$mcs^{+T_1,-T_2}(G) \bigcap V_1 = mcs^{+T_1,-T_2}(G_1)$,
and $mcs^{+T_1,-T_2}(G) \bigcap V_2 = mcs^{+T_1,-T_2}(G_2)$.*

Similarly, if the minimum vertex cover of G contains T_2, but not T_1, we have:

Proposition 6 *$mcs^{-T_1,+T_2}(G) \bigcap V_1 = mcs^{-T_1,+T_2}(G_1)$,
and $mcs^{-T_1,+T_2}(G) \bigcap V_2 = mcs^{-T_1,+T_2}(G_2)$.*

Finally, if the minimum vertex cover of G contains neither T_1 nor T_2, we have:

Proposition 7 *$mcs^{-T_1,-T_2}(G) \bigcap V_1 = mcs^{-T_1,-T_2}(G_1)$,
and $mcs^{-T_1,-T_2}(G) \bigcap V_2 = mcs^{-T_1,-T_2}(G_2)$.*

Based on these four propositions, the recurrence function associated with a parallel connection, can be expressed as follows:

$X(1) = X_1(1) + X_2(1) - 2$,
$X(2) = X_1(2) + X_2(2) - 1$,
$X(3) = X_1(3) + X_2(3) - 1$,
$X(4) = X_1(4) + X_2(4)$.

Now that the functions associated with a series or parallel connection have been specified, the bottom-up flow of the computation in the parse tree is ensured. It remains to provide a start for the computational process by specifying the values of the vector associated with the leaves of the parse tree. As we recall, a leaf in the parse tree of a TTSP graph represents the base graph of the family (Figure 1). The four components of the vector for the base graph of the TTSP family of graphs are:

$$X(BG) = (2, 1, 1, \otimes).$$

where $\otimes$ represents the undefined element. The impact of the undefined element over the subsequent evaluation of the parse tree can be summarized in the following evaluation rules:

(i) $k + \otimes = \otimes + k = \otimes$,
(ii) $\otimes - k = \otimes$,
(iii) $min(k, \otimes) = min(\otimes, k) = k$,

for any positive integer k.

4 Computing the cardinality of the minimum covering set (MCS) in real-time

In a traditional off-line paradigm, an algorithm wishing to compute the cardinality of the MCS of some graph, has access to the whole data structure representing the input graph before the computation begins. This is not the case in a real-time paradigm, where the data arrive as the algorithm proceeds. This is the basic idea underlying all real-time computations, regardless of the various particular characteristics each of them has [3, 4, 5, 6, 7, 8, 16, 22].

In our formulation of the paradigm, a subset of the input data is given at the beginning of the computation, and the rest arrive at regular intervals. The algorithm must then deal with the newly arrived data and incorporate them into the solution computed so far, before a new chunk of data arrives. Because we impose a deadline on when a partial solution (or an approximation of it) must be produced, we will call such an algorithm a *real-time* algorithm (as opposed to an *on-line* algorithm, where no deadlines are imposed [14]).

Not the whole data structure of the binary tree representing the decomposition tree of the original TTSP graph is available to the real-time algorithm at the outset. Initially, only a subtree with $O(n)$ nodes ($n < N$) is given as input to the algorithm. We assume that time is divided into intervals of cn^ϵ time units, where c is a positive constant and $0 < \epsilon < 1$. At the beginning of each time interval, a new subtree of the parse tree, with $O(n)$ nodes, is made available to the algorithm.

The division of time into fixed-length intervals and a constant data-arrival rate are two common features of real-time computations [3, 4, 5, 6]. However, we must make a distinction between two real-time settings. In the first variant, a new problem is to be solved by the algorithm during each time interval [3, 5, 6]. In the second, the computations that take place in one time interval are strongly connected to (and actually depend on) the previous computations, eventually leading to the final solution to the problem after the last time interval has elapsed [4]. From this point of view, the real-time paradigm described so far falls in the second category, being similar with the setup for computing the minimum spanning tree in real-time described in [4].

The evaluation of each subtree provided at the beginning of a time interval must be completed before a new subtree arrives, and the exact result (a four integer vector or an approximation of it) must be produced as output before the time interval ends. We note that the computation of such a partial solution corresponds to a reduction of the binary subtree to its root. It is possible for subsequent subtrees provided to the algorithm at the

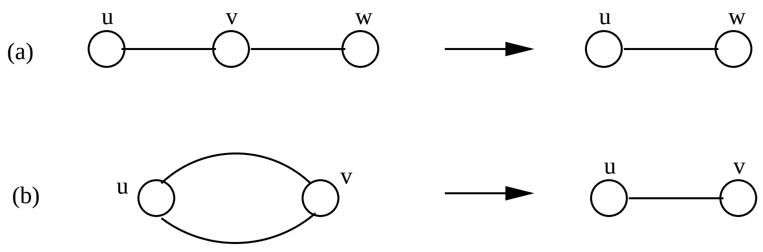

Figure 8: (a) Series reduction (requires that v have degree two). (b) Parallel reduction.

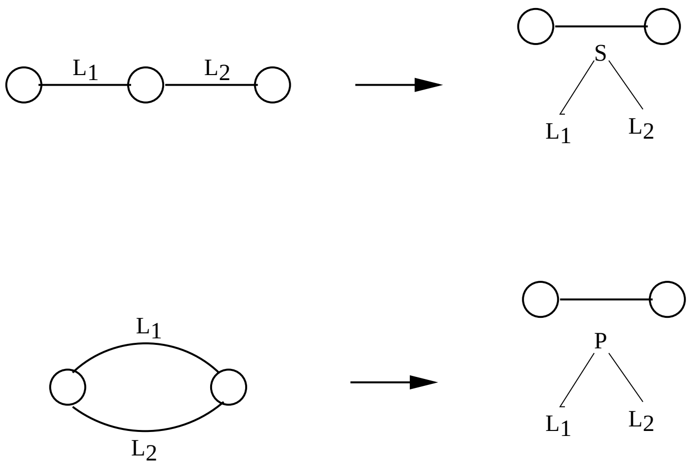

Figure 9: Computing the label of an edge introduced by a series or parallel reduction.

beginning of a future time interval, to include such a *reduced* root as a leaf, together with its previously computed vector. In this way, partial results are subsequently used in the computation of the final value.

In the real-time paradigm described above, the parse tree is continuously evolving, being reduced at each time interval. At the beginning of the last time interval, the whole current parse tree is given as the last input to the algorithm, which is supposed to produce the cardinality of the minimum vertex cover (or an approximation of it) after no more than cn^ϵ time units. At this point, an interesting question suggests itself. Is this real-time paradigm of any practical interest, or is it just artificially constructed? A closer look at a decomposition algorithm for TTSP graphs will help us answer this question.

As shown in [23], a decomposition tree for a TTSP graph can be obtained by repeatedly applying series and parallel reductions (Figure 8) to the original graph until the result is a graph with a single edge (the base graph of the family). Through this reduction process, we can obtain as a byproduct a decomposition tree of the original TTSP graph. We associate a label consisting of a binary tree with each edge of the TTSP graph being reduced. Initially, the label of each edge is a single-node binary tree. As the reduction process proceeds, we use the rules of Figure 9 to update the edge labels. The label of the last edge remaining after all reductions is the binary decomposition tree of the original TTSP graph.

Using this methodology, the parse tree is constructed in a bottom-up manner, starting with the leaves, continuing with the neighbours of the leaves, and so on, up to the root. This

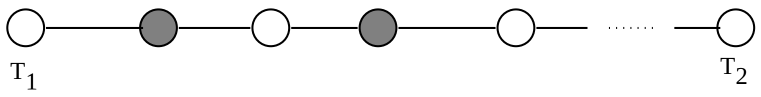

Figure 10: Best case for the approximating algorithm: a TTSP graph constructed only through series connections.

is very important for the real-time algorithm which has to compute the cardinality of the minimum vertex cover. Indeed, the algorithm can start processing the parse tree before it is completely constructed, as soon as a subtree of the desired size becomes available. The result is then incorporated in the subsequent computation. To see why this is possible, consider an intermediate step of the algorithm that generates the decomposition tree (the algorithm of [23]). The label associated with an edge of the current graph being reduced represents the subtree given as input to the real-time algorithm, at the beginning of a time interval.

4.1 Sequential algorithm

A sequential algorithm will be faced with the evaluation of a binary tree with $O(n)$ nodes at the beginning of each time interval. Obviously, no sequential algorithm will be able to process the binary tree and return the four integers composing the result vector in cn^ϵ time units, that is, before a new chunk of data arrives and has to be immediately processed. The only possibility that remains for the sequential algorithm is to somehow provide an approximation for the evaluation of the binary tree, based on the computation carried out so far. This is not an easy task, due to the four components of the vector that should be evaluated.

In these conditions, the final result computed by the sequential algorithm at the end of the last time interval will not be guaranteed to be the cardinality of the minimum vertex cover of the original TTSP graph. Furthermore, the approximation could be quite different from the actual minimum, since the output generated in one time interval can become the input in a subsequent time interval, thus propagating and increasing the error.

An alternative for the sequential algorithm would be to abandon altogether the evaluation of the parse tree and try some other approach for approximating the cardinality of the minimum vertex cover. In this case, the input to the sequential algorithm at the beginning of a time interval is no longer a subtree of the decomposition tree, but the original TTSP graph corresponding to that subtree. One possible approach could be to start with a vertex cover consisting of all the graph vertices and then try to eliminate as many of these as possible in the available time. The best case for such an approximation algorithm is illustrated in Figure 10. Assuming that in cn^ϵ time units $O(n^\epsilon)$ vertices can be checked and possibly eliminated from the initial vertex cover, the cardinality of the remaining set will still be on the order of n, which is asymptotically equal to the optimal solution. But in the worst case (Figure 11), although the approximation computed by the sequential algorithm is again $O(n)$, the actual minimum vertex cover of such a TTSP graph contains only the two terminals (in other words, its cardinality is a constant), regardless of how many vertices the graph might have.

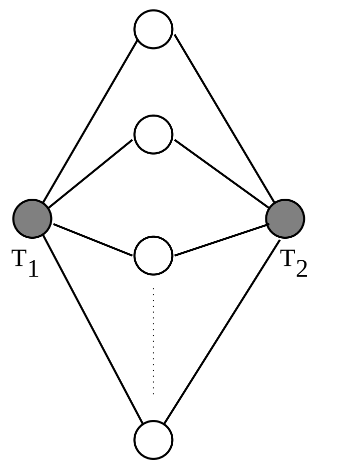

Figure 11: Worst case for the approximating algorithm.

4.2 Parallel algorithm

On the other hand, efficient parallel algorithms for tree contraction are known to exist [1, 10, 15, 19] and if the operations defined in our algebra for solving the minimum vertex cover problem meet some requirements [1] these algorithms can be successfully applied to our particular problem. Although proving the satisfiability of these conditions is not very difficult, a more elegant justification is provided by a recent result of Miller and Teng [20]. According to their research, we are entitled to use a parallel tree contraction algorithm in our particular case due to the fact that the only operators used in the formulas that describe the series and parallel connections are $+$, $-$ and min. As a consequence, we are able to process a binary tree with $O(n)$ nodes in a bottom-up fashion, in order to compute the vector associated with the root of the binary tree, in $O(\log n)$ time. The model of parallel computation employed in all cases is an EREW PRAM [2]. Some of the algorithms use $O(n)$ such processors in order to complete the task, while others use only $O(n/\log n)$ processors, thus achieving cost optimality.

However, the important feature of these algorithms is their running time. Such a parallel algorithm will be able to reduce the binary tree of $O(n)$ nodes, received as input at the beginning of a time interval, to its root (computing the exact values of the associated vector), in $O(\log n)$ time units. Therefore, within each time interval of length cn^{ϵ} time units, the corresponding partial solution can be exactly computed, eventually leading to obtaining the actual value for the cardinality of the minimum vertex cover.

5 Analysis

Let us compare the performances of the two real-time algorithms exhibited, the sequential one and the parallel one, in terms of their output. If the sequential algorithm cannot provide any approximation for the vector to be computed, then in terms of their results, the difference between the two algorithms is that separating success and failure.

Otherwise, the performance of the algorithms could be judged by the accuracy of the solution provided to the MCS problem. The accuracy of the solution, in our case, is defined as:

$$accuracy\ of\ the\ solution = \frac{1}{size\ of\ the\ solution}$$

Based on this definition, the *accuracy ratio* can be used as a measure of the relative performance of the two algorithms:

$$accuracy\ ratio = \frac{accuracy\ of\ the\ parallel\ solution}{accuracy\ of\ the\ sequential\ solution}$$

The parallel algorithm will always compute the exact cardinality of the minimum covering set. For the sequential algorithm, the most trivial and rough approximation would be to count a vertex for each edge of the original TTSP graph. Note that, asymptotically, this is equivalent to including all the vertices in the covering set, due to the fact that in a TTSP graph the number of edges is on the order of the number of vertices. As a consequence, depending on the approximation method employed, the final value computed by the real-time sequential algorithm could be anywhere in the interval delimited by the actual cardinality of the MCS and the number of edges in the TTSP graph.

Compared with the particular approximation algorithm described in section 4.1, the parallel real-time algorithm yields a solution which is $O(n)$ times more accurate than the sequential solution and this applies to the partial solution computed at the end of a time interval. Note that if a cost optimal parallel tree contraction algorithm is employed, which uses only $O(n/\log n)$ processors, the accuracy ratio is superlinear in the number of processors. The cumulative effect of approximations computed in each time interval will eventually determine an even greater *accuracy ratio* to the advantage of the parallel algorithm (naturally, at the cost of increasing the overall running time).

Usually, the performance of a parallel algorithm is judged in terms of the speedup achieved, relative to the best known sequential algorithm for the same problem. Speedup is a natural consequence of the increased resources (more than one processing unit) of any parallel model of computation, which makes it faster than a sequential machine. But speedup is not the only measure of the superiority of a parallel algorithm, when compared with a sequential one. The real-time paradigm exhibited in this paper clearly suggests that success versus failure, or the accuracy ratio of the solutions computed, are alternative measures for the performance of a parallel algorithm. In some cases, the fact that a parallel machine is faster than a sequential one might be translated, in a real-time environment, not into a remarkable speedup, but into a dramatically improved quality of the solution computed.

6 Conclusions

We conclude this paper with some observations about the real-time paradigm defined herein and suggest some possible continuations for this research. As one can see from the above analysis, the parallel real-time algorithm is always superior in performance to the sequential algorithm. This result is true not only for the minimum vertex cover problem for

TTSP graphs, but it could be extended in a similar way to other problems and/or families of graphs listed in [24]. The key factor in ensuring the superiority of parallel algorithms over sequential ones in this kind of real-time computation is imposing a deadline on when the output should be produced.

We note that this real-time paradigm is much different from the one exhibited in [14], for example, which is merely an on-line paradigm, where no rate at which data are to be received or results are to be produced is specified. Furthermore, the algorithms in [14] are forced to make decisions without any knowledge of the future, and this can dramatically affect their performance. In our case, although the real-time algorithm still has no knowledge of what is about to arrive as input, this fact is of no consequence on the performance of the algorithm, each chunk of data being processed in exactly the same way, without any crucial decision to be made.

The model of computation used in this paper is an EREW PRAM. Another idea which appears to be worthwhile to investigate is to use other models of parallel computation, less sophisticated than a PRAM (a linear array of processors, for example) in the real-time environment and still maintain the same superiority of the parallel algorithm, when compared to the best possible sequential one. Finally, other paradigms of real-time computation could be investigated. In one such paradigm, corrections to the existing data arrive on-line and must be incorporated in the solution to the problem at hand [8].

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick and T. Przytycka, "A Simple Tree Contraction Algorithm", Journal of Algorithms 10 (1989), pp. 287-302.

[2] S.G. Akl, "Parallel Computation: Models and Methods", Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[3] S.G. Akl, "Nonlinearity, maximization, and parallel real-time computation", Proceedings of the 12th Conference on Parallel and Distributed Computing and Systems, Las Vegas, Nevada, November 2000.

[4] S.G. Akl and S.D. Bruda, "Parallel real-time optimization: beyond speedup", Parallel Processing Letters 9 (1999), pp. 499-509.

[5] S.G. Akl and S.D. Bruda, "Parallel real-time cryptography: Beyond speedup II", Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June 2000, pp. 1283-1289.

[6] S.G. Akl and S.D. Bruda, "Parallel real-time numerical computation: Beyond speedup III", International Journal of Computers and their Applications, Special Issue on High Performance Computing Systems 7 (2000), pp. 31-38.

[7] A. Bestavros and V. Fay-Wolfe, Eds., "Real-Time Database and Information Systems", Kluwer Academic Publishers, Boston, 1997.

[8] S.D. Bruda and S.G. Akl, "A case study in real-time parallel computation: Correcting algorithms", to appear in Journal of Parallel and Distributed Computing.

[9] M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, San Francisco, 1979.

[10] H. Gazit, G.L.Miller and S.-H. Teng, "Optimal tree contraction in an EREW model", Proceedings of the 1987 Princeton Workshop on Algorithm,Architecture and Technology: Issues for Models of Concurrent Computation, pp. 139-156, 1987.

[11] X. He, "Efficient parallel algorithms for solving some tree problems", in 24th Allerton Conference on Communication, Control and Computing, 1986, pp. 777-786.

[12] X. He, "Efficient Parallel Algorithms for Series Parallel Graphs", Journal of Algorithms 12 (1991), pp. 409-430.

[13] X. He and Y. Yesha, "Parallel recognition and decomposition of two terminal series parallel graphs", Inform. Comput. 75, No. 1 (1987), pp. 15-38.

[14] S. Irani and A.R. Karlin, Online computation, in: D.S. Hochbaum, Ed., "Approximation Algorithms for NP-Hard Problems", International Thomson Publishing, Boston, Massachusetts, 1997, pp. 521-564.

[15] S.R. Kosaraju and A.L. Delcher, "Optimal parallel evaluation of tree-structured computation by raking (extended abstract)". In "VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88", ed. J.H. Reif, pp. 101-110. LNCS, Vol. 319, Springer-Verlag, Berlin, 1988.

[16] H.W. Lawson, "Parallel Processing in Industrial Real-Time Applications", Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[17] R. Lin and S. Olariu, "An optimal parallel matching algorithm for cographs", Journal of Parallel and Distributed Computing 22 (1994), pp. 26-36.

[18] I. Parfenoff, "An Efficient Parallel Algorithm for Maximum Matching", Journal of Parallel and Distributed Computing, Vol. 52, No. 1, July 1998, pp. 96-108.

[19] G.L. Miller and J.H. Reif, "Parallel Tree Contraction and its Applications", Proc. 26th FOCS, 1985, pp. 478-489.

[20] G.L. Miller and S.-H. Teng, "Tree-Based Parallel Algorithm Design", Algorithmica 19 (1997), pp. 369-389.

[21] K. Takamizawa, T. Nishizeki and N. Saito, "Linear-time computability of combinatorial problems on series-parallel graphs", J. Assoc. Comput. Mach., 29 (1982), pp. 623-641.

[22] M. Thorin, "Real-Time Transaction Processing", Macmillan, London, 1992.

[23] J. Valdes, R.E. Tarjan and E.L. Lawler, "The recognition of series parallel digraphs", SIAM J. Comput. Vol.11, No. 2, May 1982, pp. 298-313

[24] T.V. Wimer, S.T. Hedetniemi and R. Laskar, "A methodology for constructing linear graph algorithms", Congressus Numerantium 50 (1985), pp. 43-60.