A 2-approximation NC algorithm for connected vertex cover and tree cover

Toshihiro Fujito a,*,1, Takashi Doi b

a Graduate School of Information Science, Nagoya University, Furo, Chikusa, Nagoya 464-8603, Japan
b Department of Information Electronics, Nagoya University, Furo, Chikusa, Nagoya 464-8603, Japan

Received 14 July 2003; received in revised form 11 December 2003
Communicated by F.Y.L. Chin

Abstract

The connected vertex cover problem is a variant of the vertex cover problem, in which a vertex cover is additional required to induce a connected subgraph in a given connected graph. The problem is known to be NP-hard and to be at least as hard to approximate as the vertex cover problem is. While several 2-approximation NC algorithms are known for vertex cover, whether unweighted or weighted, no parallel algorithm with guaranteed approximation is known for connected vertex cover. Moreover, converting the existing sequential 2-approximation algorithms for connected vertex cover to parallel ones results in RNC algorithms of rather high complexity at best.

In this paper we present a 2-approximation NC (and RNC) algorithm for connected vertex cover (and tree cover). The NC algorithm runs in $O(\log^2 n)$ time using $O(\Delta^2(m + n)/\log n)$ processors on an EREW-PRAM, while the RNC algorithm runs in $O(\log n)$ expected time using $O(m + n)$ processors on a CRCW-PRAM, when a given graph has $n$ vertices and $m$ edges with maximum vertex degree of $\Delta$.

Keywords: Approximation algorithms; Parallel algorithms; Connected vertex cover; Tree cover

1. Introduction

Let $G = (V, E)$ be an undirected graph with $n$ vertices and $m$ edges (i.e., $|V| = n, |E| = m$). A vertex set $C \subseteq V$ is called a vertex cover (or abbreviated to “vc”) if every edge of $G$ is incident to some vertex in $C$, and the vertex cover (VC) problem is to compute a minimum vc in given $G$. When a vertex cover is additionally required to induce a connected subgraph in a given connected graph, such a vc is called a connected vertex cover (or abbreviated to “cvc”) and the problem is termed CVC. In the tree cover (TC) problem it is required to compute a minimum edge set $T$ in a connected graph s.t. $T$ is connected, and every edge outside of $T$ is adjacent to some in $T$. Easily, any minimal solution for TC is necessarily a tree, and the problem is also called connected edge dominating set.

The VC problem is a classic NP-hard graph problem [15], and has been studied extensively in the litera-
ture (see [13] for an overview). The CVC problem typically arises in applications of VC where certain connectivity constraints need to be enforced on possible solutions. This problem is also known to be NP-hard, and in fact, it can be shown, by a reduction from VC, to be as hard (to approximate) as VC is [10]. The TC problem on the other hand is computationally equivalent (exact or approximate) to the CVC problem because $C = V(T)$ is a cvc for any tree cover $T$, and conversely, any tree $T$ spanning all the vertices in a cvc $C$ is a tree cover. Moreover, $|T| = |C| - 1$ always holds for any $T$ and the corresponding $C$ (and vice versa). Therefore, any approximate solution for one with a ratio $\alpha$ of the optimal size to its size can be directly translated into a solution for another having the same approximation ratio of $\alpha$, implying that these two problems possess the same approximation complexity.

Given the apparent intractability in exact computation of these problems, the development of efficient (sequential) approximation algorithms for them has been very active and popular. A factor $2$-approximation of VC was found early [11]; it suffices to compute and output the set $V(M)$ of vertices matched by any maximal matching, from which an “approximate” min-max relation between maximal matchings and vertex covers can be observed. Currently the best approximation algorithm for VC achieves a ratio of $2 - \log_{\log n} n$ [2,20]. The approximation of CVC has been mainly approached through that of TC, and they are also known to be approximable within a factor of $2$ [22,2]. In more general weighted variants of these problems, elements (i.e., either vertices or edges) are a priori associated with nonnegative weights, and the goal becomes to minimize the total weight of elements in a solution. The best approximation bound for unweighted VC holds for weighted VC as well [3], whereas we have a different story for CVC and TC. Notice first that, when weighted, CVC and TC are no longer computationally equivalent as different types of elements are weighted. The weighted TC was first shown by Arkin, Halldörsson and Hassin [2] to be approximable within a factor of $3.55$, and it was recently improved to $3 + \varepsilon$ in [18,9]. Contrary to this, it was shown in [9] that, while weighted CVC can be approximated within a factor of $\ln n + 3$, it cannot be within $(1 - \varepsilon) \ln n$ for any $\varepsilon > 0$ (unless $\text{NP} \subseteq \text{DTIME}(n^{o(\log \log n)})$) using the result of Feige [8].

As the field of approximation algorithms grows more mature and provides more approximation techniques for dealing with hard optimization problems, while even better approximation guarantees are eagerly sought, it is naturally desirable at the same time to have more efficient implementations without losing approximation accuracy known possible, and the fast parallel approximation has become a subject of current interest (see e.g., the book of Díaz et al. [7]). A maximal matching is known to be in $\text{NC}$ [19,16,4], and it follows immediately that a $2$-approximation of VC is in $\text{NC}$. Moreover, a $2$-approximation NC algorithm was designed, based on the primal-dual technique, even for weighted VC by Khuller, Vishkin and Young [17]. On the other hand, no parallel algorithm with proven approximation guarantee is known up to today for CVC or TC. Even worse, either of the existing sequential approximation algorithms for CVC contains computational tasks for which no NC algorithm is known; namely, depth-first search in the Savage’s [22] and maximum bipartite matching in the Arkin et al.’s [2]. Using the RNC algorithms for dfs (of Aggarwal and Anderson [1]) and minimum weight perfect matching (of Mulmuley et al. [21]), and the fast matrix multiplication algorithm [6], the Savage’s algorithm for CVC can be converted into an RNC algorithm running in time $O(\log^5 n)$ using $O(n^{4.376})$ processors on CRCW-PRAM. To parallelize the algorithm of Arkin et al. [2] for TC, on the other hand, we may use the NC algorithm of Gazit and Miller [12] for breadth-first search ($O(\log^2 n)$ time, $O(n^{2.378})$ processors on EREW-PRAM), and the Mulmuley et al.’s RNC algorithm [21] for maximum bipartite matching ($O(\log^2 n)$ expected time, $O(mn^{3.376})$ processors on CREW-PRAM). We then obtain a $2$-approximation RNC algorithm for CVC, running in time $O(\log^2 n)$ with $O(mn^{3.376})$ processors on CREW-PRAM.

In this paper we present a new $2$-approximation algorithm for CVC (and TC). The parallel implementation of this algorithm will be shown to result in a $2$-approximation NC algorithm running in $O(\log^2 n)$ time using $O(\Delta^2(m+n)/\log n)$ processors on EREW-PRAM, and a $2$-approximation RNC algorithm running in $O(\log n)$ expected time using $O(m+n)$ processors on CRCW-PRAM, where $\Delta$ is the maximum vertex degree of a given graph. On our way to developing this algorithm, we introduce so called compactly maximal matchings as a counterpart of maximal match-
Proposition 1. For any cmm $M$ in $G$, its vertex set $V(M)$ is a cvc of size at most twice the minimum cvc size for $G$.

Proof. Clearly $V(M)$ is a vc for $G$ because $M$ is a maximal matching, and by the definition of a cmm, $V(M)$ is a cvc of size $2|M|$. On the other hand, no cvc is of size smaller than $|M|$; any vc is of size at least $|M|$ since $M$ is a matching, and any cvc is a vc as well. Thus, $V(M)$ is a cvc of size no larger than twice the minimum cvc size.

Corollary 2. In any (connected) graph,

$$|\text{any cmm } M| \leq \min_{\text{cvc } C} |C| \leq 2 \cdot |\text{any cmm } M|.$$  

(Note: merely maximal matchings cannot be related to connected vertex covers within a factor of 2.)

It is now a simple matter to design a 2-approximation algorithm for CVC; find a cmm $M$ in $G$, and output $V(M)$, the set of matched vertices. In general a cmm can be computed fast “sequentially”. To facilitate efficient parallel computation, however, a cmm will be constructed in two step; first within a spanning tree, and then within the remainder of $G$.

For a tree $T$, let $L(T)$ denote the set of leaves in $T$, and $NL(T)$ the set of non-leaves.

Lemma 3. For any tree $T$ with $|V(T)| \geq 3$ and any cvc $C$ in $T$, $C \supseteq NL(T)$. In particular, $NL(T)$ is the unique minimum cvc for $T$.

Proof. If $C$ excludes any $v \in NL(T)$, $C$ cannot induce a connected subgraph in $T$. So, it must be the case that $C \supseteq NL(T)$. On the other hand, $C \setminus \{v\}$ is still a cvc for any leaf $v$ contained in $C$. Thus, a minimum cvc cannot have any leaf vertex in it.

Consider the following algorithm, where $G[U]$ denotes the subgraph of $G$ induced by $U \subseteq V$:

Algorithm 1.

1. Compute a spanning tree $T$ for $G$.
2. Compute a cmm $M_1$ in $T$.
3. Compute a maximal matching $M_2$ in $G - V(M_1) = G[V - V(M_1)]$.
4. Output $C = V(M_1 \cup M_2)$.

Theorem 4. Algorithm 1 computes a cvc $C$ of size no larger than twice the size of a small cvc in $G$.

Proof. It suffices by Proposition 1 to show that $M_1 \cup M_2$ is a cmm in $G$. Since a matching $M_2$ is maximally constructed in $G[V - V(M_1)]$ for a matching $M_1$, $M_1 \cup M_2$ is necessarily a maximal matching in $G$. Since $M_1$ is a cmm in the spanning tree $T$ of $G$, its vertex set is a cvc in $T$, and hence, $V(M_1) \supseteq NL(T)$ by Lemma 3. So, any vertex $u$ matched by $M_2$ is a leaf of $T$, and $u$ is adjacent to a non-leaf which is necessarily matched by $M_1$. Hence, $M_1 \cup M_2$ is also compact.

2.1. Compactly maximal matchings in a tree

Here we discuss how to compute a cmm in a tree $T$ (step 2 of Algorithm 1) in such a manner suitable for parallel execution. For any rooted tree $\bar{T}$, the directed tree $\bar{T}$ is called an out-tree of $T$ if $\bar{T}$ is obtained from $T$ by replacing each (undirected) edge (of $T$) by a directed one s.t. every edge of $\bar{T}$ is directed away from the root toward a leaf. When edge directions are ignored in $\bar{T}$, we denote it by $T$.

For each non-leaf $u \in NL(T)$, choose one child $v$ of $u$, and mark the directed edge $(u, v)$ of $\bar{T}$ “chosen”. Let $P$ be a set of the edges chosen this way. Then, $P$ can be divided into subsets $P_i$’s s.t. each $P_i$ constitutes a directed path starting in $NL(T)$ and ending at some leaf in $T$. When edges in a dipath $P_i$ are numbered successively from the starting edge to the ending one, the set $M_i$ of odd numbered edges in $P_i$ is clearly a
cmm in $P_i$. The next algorithm computes all these $P_i$'s and $M_i$'s in parallel:

**Algorithm 2.**

1. Convert a given undirected unrooted $T$ into a rooted out-tree $\overline{T}$.
2. For each non-leaf $u$ of $\overline{T}$, choose one child $v$ and mark the directed edge $(u, v)$ “chosen”.
3. Compute a set of disjoint dipaths $P_i$ formed by those edges marked “chosen” in step 2.
4. For each dipath $P_i$ computed in step 3, label all the edges in $P_i$ with “1” and “0” alternatively, starting with “1” at the first edge, continuing with “0” at the second, and so on.
5. Collect all the edges labeled “1” into the set $M$.

**Lemma 5.** The edge set $M$ computed by Algorithm 2 is a cmm in $T$.

**Proof.** From the way it is constructed, each $M_i$ is clearly a cmm in $P_i$. Since $P_i$'s are vertex disjoint from each other, $M = \bigcup_i M_i$ is a matching in $T$. Observe that every non-leaf $v$ of $T$ lies on some path $P_i$, and $M_i$ matches all the vertices in $P_i$ except possibly for the last vertex of $P_i$, which is necessarily a leaf of $T$. Therefore, every non-leaf of $T$ is matched by some edge in $M$ (i.e., $NL(T) \subseteq V(M)$), and it follows from this that $M$ is a cmm in $T$. □

3. Parallel complexity

Describing the parallel implementations of Algorithms 1 and 2, we estimate their parallel complexity.

**Lemma 6.** Algorithm 2 runs on EREW-PRAM in time $O(\log n)$ using $O(n)$ processors.

**Proof.** We use the so-called “Euler tour technique”, described in [23], for the execution of step 1. Given a tree $T$ in an adjacency list representation, each edge is replaced by two anti-parallel edges; the resulting graph is then Eulerian. A traversal list $L_i$ of $T$ is constructed from the circular list $L_e$ of edges representing an Euler cycle, by simply breaking $L_e$ at an arbitrary edge, and fixing the first edge of $L_i$ to some edge $(u_1, u_2)$. By ranking the elements of $L_i$ by the distance from $u_1$ in $L_i$, and choosing the lower ranked edge, out of $(v_1, v_2)$ and $(v_2, v_1)$, for each edge $(v_1, v_2)$ in $T$, we obtain an out-tree $\overline{T}$ rooted at $u_1$. Using the standard doubling technique, this step can be executed in $O(\log n)$ time with $O(n)$ processors.

After choosing one edge $(u, v)$ at each vertex $u$ in step 2, if $v$ is a non-leaf and an edge $(v, w)$ is chosen by $v$, the “next” pointer of $(u, v)$ is set pointing to $(v, w)$ in step 3. Each of the resulting lists of edges represents a dipath $P_i$, and either of steps 2 and 3 can be executed in $O(1)$ time using $O(n)$ processors.

Step 4 is executable in $O(\log n)$ time with $O(n)$ processors, using the doubling technique, and step 5 is an $O(1)$ time operation with $O(n)$ processors. □

The next theorem proves that 2-approximation of CVC is in NC$^2$ and in RNC$^1$.

**Theorem 7.** Algorithm 1 runs in $O(\log^2 n)$ time using $O(\Delta^2(m + n)/\log n)$ processors on EREW-PRAM, and in $O(\log n)$ expected time using $O(m + n)$ processors on randomized CRCW-PRAM, where $\Delta$ is the maximum vertex degree of a given graph.

**Proof.** In step 1 a spanning tree $T$ for $G$ can be computed in $O(\log n)$ time using $O(m + n)$ processors on EREW-PRAM, due to the work of Chong, Han and Lam [5]. Step 2 requires $O(\log n)$ time and $O(n)$ processors on EREW-PRAM according to Lemma 6. A maximal matching can be computed in step 3 using $O(\log^2 n)$ time and $O(\Delta^2(m + n)/\log n)$ processors on EREW-PRAM by the algorithm of Chen [4]. Alternatively, step 3 can be implemented by the RNC algorithm of Israeli and Itai [14], spending this time on CRCW-PRAM, $O(\log n)$ expected time and $O(m + n)$ processors. □

References