Discrete Applied Mathematics 10 (1985) 27-45
North-Holland

SOLVING NP-HARD PROBLEMS IN 'ALMOST TREES': VERTEX COVER

Don COPPERSMITH
Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA

Uzi VISHKIN*
Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

Received 4 May 1983
Revised 22 February 1984

We present an algorithm which finds a minimum vertex cover in a graph $G(V,E)$ in time $O(|V|+(a/k)2^{k/3})$, where for connected graphs G the parameter a is defined as the minimum number of edges that must be added to a tree to produce G, and k is the maximum a over all biconnected components of the graph. The algorithm combines two main approaches for coping with NP-completeness, and thereby achieves better running time than algorithms using only one of these approaches.

1. Introduction

Garey and Johnson [1], in the introduction to the chapter 'Coping with NP-Complete Problems', note that 'it is sometimes possible to reduce substantially the worst case time complexity of exhaustive search merely by making a more clever choice of the objects over which the exhaustive search is performed.' As examples for this approach, they give [5] and other works. On the other hand, Gurevich, Stockmeyer, and Vishkin [2] suggest algorithms for NP-hard graph problems for sparse graphs, in which the complexity is exponential in the maximum number of additional edges to a tree in a biconnected component of the graph, rather than in the size of the problem instance. This makes sense especially for problems that are polynomial for trees but NP-hard for fairly sparse graphs. As was noted in that paper, the idea of analyzing worst case running time of algorithms as a function of parameters other than the size of the input has a precedent in Chapter 4 of [1] in their "pseudo-polynomial time algorithms". The present paper exemplifies the intersection of these two approaches.

* This research was performed while the second author was a visiting scientist at IBM Thomas J. Watson Research Center.

0166-218X/85/$3.30 © 1985, Elsevier Science Publishers B.V. (North-Holland)

The 'Independent Set Problem' is to find a maximum set of vertices such that no two are connected by an edge. The 'Vertex Cover Problem' is to find a minimum set of vertices such that at least one endpoint of every edge belongs to the set. Obviously, the complement of a solution for the first problem is a solution for the second, and vice versa. The 'Clique Problem' is to find a maximum set of vertices, each pair of which is connected by an edge. A solution of the independent set problem in a graph G is a solution of the clique problem in the complement of G. The three problems are known to be NP-complete.

Let $G(V,E)$ be an undirected graph, where V is the set of vertices, and E is the set of edges. Let $n=|V|$ and $m=|E|$. For G connected, define $a(G)=m-n+1$. More generally, if C is the number of connected components in G, set $a(G)=m-n+C$. We can view $a(G)$ as the number of additional edges to a spanning forest of G. Define $k(G)$ to be the maximum of $a(G_j)$, where G_j range over the biconnected components of G.

We present an algorithm for computing a minimum vertex cover for G, which runs in time $O(n+(a/k)\theta^k)$, where $\theta=1.2365<2^{\frac{1}{3}}$.

Tarjan and Trojanovsky [5] give an $O(2^{n/3})$ algorithm for the independent set problem, and therefore the vertex cover problem. It is interesting to note that, for a connected regular graph of degree 3, a class for which these problems remain NP-complete, we have $k\leq n/2+1$, whence our result gives time $O(n+2^{n/6})$ for vertex cover and independent set, therefore improving upon [5] for this class of graphs.

Our result is also an obvious improvement upon the $O(m2^{k/2})$ algorithm given in [2]. We feel that our present analysis supports the use of the measure of "maximum additional edges to a tree in a biconnected component" in the design and analysis of algorithms; this paper therefore strengthens [2] in supporting it as a natural and relevant measure.

Due to a simple technique which is presented in [2], we can derive a parallel implementation for our algorithm. Its running time is $O(n+a\theta^k/kp)$ for $p\leq\theta^k/k^2$ processors. This implies an optimal speed-up over the sequential version for a fairly wide range of the parameter p.

Sections 3 and 4 give the algorithm and the main procedure it employs (Procedure Graph) as well as a verbal overview. The second procedure is given and described in detail in Section 5. In Section 6 we elaborate on the key instruction of Procedure Graph, preparing the ground for the complexity analysis of Section 7. Section 8 deals with a parallel implementation of the algorithm.

2. Preliminaries

Two edges satisfy the *biconnectivity relation* if they are identical, or if they are contained in a simple cycle (i.e. a cycle with no repeated vertices). It can be shown that the biconnectivity relation is an equivalence relation.

An equivalence class of edges in G, under the biconnectivity relation, together

with their vertices, is called a *biconnected component*. A graph G is *biconnected* if it has a single biconnected component and no isolated vertices. The intersection of two biconnected components, if nonempty, is a single vertex, called an *articulation point*.

A few remarks about k and a:

If G is connected, $a(G)$ is nonnegative, and $a(G)=0$ iff G is a tree.

If a connected graph G has several biconnected components G_j, then $a(G)=\sum_j k(G_j)$.

Our algorithm will be exponential in $k(G)$ (rather than in $a(G)$), and for this reason we choose k as the measure of complexity.

In presenting our algorithm, we use a programming construct called 'Select', in addition to the more commonly used constructs. Let P_i be predicates and I_i be instructions. We will write

```
Select
    When P_1
    Then I_1
    When P_2
    Then I_2
    ...
    When P_t
    Then I_t
    Otherwise I_{t+1}
```

By this we will mean:

```
If P_1
Then I_1
Else If P_2
    Then I_2
    Else If P_3
    ...
    Else If P_t
        Then I_t
        Else I_{t+1}
```

3. The algorithm

A high-level description of the algorithm is followed by the algorithm itself.

We employ a recursive procedure, Graph(G), which takes as input a graph G, and adds to the variable $D(G)$ the cardinality of a minimum vertex cover for G.

During the algorithm we follow a convention which may seem confusing at first,

but will turn out to have some advantges. We refer to a graph by a name, say T. Later the graph is changed: some vertices are deleted, edges change place, etc. In spite of that, we still call the reduced graph by the name T. Meanwhile, in $D(T)$ we accumulate the number of vertices in a minimum vertex cover (MVC) for the graph which was first associated with the name T. Therefore, at the end of the algorithm, $D(G)$ will equal the cardinality of an MVC of G.

The vertex cover itself is returned by Graph(G) in the vector $L(G, v)$ for all $v \in G$. Its possible values are:

$$L(G, v)=\begin{cases} 0, & v \text{ does not belong to the vertex cover,} \\ 1, & v \text{ belongs to the vertex cover,} \\ \text{``}w\text{''}, & v \text{ belongs to the vertex cover iff } w \text{ does,} \\ \text{``}\neg w\text{''}, & v \text{ belongs to the vertex cover iff } w \text{ does not.} \end{cases}$$

Let us define an auxiliary directed graph F with the same vertices as G, and (v, w) is an edge of F iff $L(G, v) =$ "w" or "$\neg w$". The outdegree of each vertex is either 0 or 1. If outdeg(v) = 1, there is a unique directed path in F from v to a vertex u such that $L(G, u) =$ either 0 or 1. With the help of a stack, we assign 'final' values to $L(G, v)$ for all $v \in G$.

Main Algorithm

```
For all vertices x in G, L(G,x) ← "UNDEFINED";
D(G) ← 0;
Graph(G, 0);
(* Post-process L *)
Stack is empty initially
For every vertex x such that L(G,x) = "w" or L(G,x) = "¬w"
Do
   u ← w
   Stack ← x
   While L(G,u) = "z" or "¬z"
   Do
       Stack ← u
       u ← z
   Od
   While Stack is not empty
   Do
       u ← Stack
       If L(G,u) = "w"
       Then L(G,u) ← L(G,w)
       Else L(G,u) ← 1 − L(G,w)
   Od
Od
End;
```

4. Procedure Graph

An overview of procedure graph is followed by a statement of this procedure.

After computing the biconnected components of G, the algorithm chooses a biconnected component with maximum value k to be the 'distinguished vertex' in each tree of an auxiliary forest H (which describes the build-up of G from its biconnected components; see the definition of H in the program itself). The reason for this choice will become clear in the complexity analysis. The algorithm computes the vertex cover of one biconnected component at a time, taking into account the fact that a few vertices of the biconnected component may have been chosen earlier to belong to the vertex cover. At each point, it selects a leaf in H as the component to treat next, leaving the 'distinguished' component for the end. The procedure is designed to minimize the size of the vertex cover as a primary objective, and secondarily to include the articulation vertex in this cover.

A non-distinguished component is first 'disconnected' from the rest of the graph at its articulation vertex, x. This is done by forming two copies of the component. In the first copy, x is assumed to belong to the MVC, and in the second, it is assumed not to belong to it. (These assumptions on x are implemented via the labelling vector L; if x is already known to be a member of MVC we avoid the branching.) Both copies are 'cleaned' from previously labelled vertices or vertices of degre 0, 1, or 2, by the procedure Clean.

During Clean, besides deleting vertices we update $D(G)$. When we delete a vertex which is labelled 1 we add one to $D(G)$, for obvious reasons. When we delete a vertex labelled "w" we also add one to $D(G)$. This is because in every instruction in which a vertex receives the label "w", exactly one corresponding vertex receives the label "$\neg w$", so that between that pair of vertices, exactly one will be included in the MVC.

Every vertex of a 'cleaned' component is of degree at least 3. Later, when we elaborate further on the procedure Clean, we will show that a 'cleaned' biconnected component has no more than $2k$ vertices or $3k$ edges, where $k = k$(original biconnected component).

Now Graph is called recursively from both copies. We choose between the MVC's that are suggested by the two copies according to the criterion stated above: primarily minimizing cardinality, secondarily including x.

Finally we reach a distinguished biconnected component. After cleaning it, we invoke a procedure described in Section 6; below we give only an overview of this procedure. If any vertex of degree >3 exists, we branch over this vertex, clean each copy, and call Graph recursively for each copy. If not, we search for one of a few patterns, eliminate that pattern after doing some required operations, and then call Clean and Graph recursively on one or more copies (according to the pattern). Otherwise, there will be a vertex u of degree 3 which does not belong to a triangle (as will be shown). Again we branch over this vertex, clean each copy, and call Graph recursively for each copy.

A common practice upon recursive calls to Graph(G_i) for a distinguished or non-distinguished component, is the following. Before a call, record in V_i the vertices whose labels may change (i.e. their membership in MVC may be decided) during the operation of Graph(G_i). After the call, update the labelling of the vertices of the original graph G, V_i, and update $D(G)$. For the latter operation we have to be careful about counting articulation vertices. The articulation vertex of G_i is counted in $D(G_i)$ if it is in the MVC, but it is not counted in $D(G)$, since it will be counted again later (in the biconnected component corresponding to the 'father' of G_i in the forest H).

As an aid in the complexity evaluation, we define here three quantities. Let $H(n, s)$ be the worst-case running time of Graph(G), where G is connected, $a(G)=s$, and n is the number of vertices. Set $H(s)=H(2s, s)=\max H(n, s)$ for $n\leq 2s$. Let $H^*(s)=\max(\sum H(s_i))$, where the maximum is over positive integers s_i such that $\sum_i s_i=s$. Set $F(s)=$ worst-case running time for Graph(B) where B is biconnected, $k(B)=s$, and $\deg(v)\geq 3$ for all vertices v in B.

Procedure Graph(G)

1. If G is empty
   Then Return;

2.1. Find the biconnected components of G [4]. Create an *auxiliary graph* (forest) H. Vertices of H correspond to biconnected components of G and articulation points of G. An edge in H joins a 'biconnected component' and an 'articulation point' if the corresponding articulation point is contained in the corresponding biconnected component in G.

2.2. From each tree H_i in the forest H, choose as a *distinguished vertex* a biconnected component $G_{\text{root},i}$ with largest possible value of k.

3. While [there are non-distinguished biconnected components which have not been handled]
   Do
   3.1. Choose such a biconnected component G_i which is a leaf in H, and let x be its unique articulation point, corresponding to G_i's unique neighbor in H.
   3.2. $V_i \leftarrow$ {the vertices in G_i}
   3.3. If $L(G, x)=1$
        Then
          3.3.1.
             Create a copy of G_i, to be referred to as G_i;
             $D(G_i)\leftarrow 0$; $L(G_i, v)\leftarrow L(G, v)$ for all $v\in V_i$
             Clean (G_i); Graph(G_i)
             (* Update G *)
             $D(G)\leftarrow D(G)+D(G_i)-1$
             $L(G, v)\leftarrow L(G_i, v)$ for all $v\in V_i$

Else
3.3.2. Create two copies of G_i, to be referred to as F_1, F_2
For $j=1,2$
$D(F_j) \leftarrow 0; L(F_j, v) \leftarrow L(G_i, v)$ for all $v \in V_i$
$L(F_1, x) \leftarrow 1$
For all vertices $y \in F_2$ such that (x, y) is an edge in F_2
$L(F_2, y) \leftarrow 1$
For $j=1,2$
Clean(F_j); Graph(F_j)
(* Choosing *)
If $D(F_1) \leq D(F_2)$
Then $D(G) \leftarrow D(G) + D(F_1) - 1$
$L(G, v) \leftarrow L(F_1, v)$ for all $v \in V_i$
Else $D(G) \leftarrow D(G) + D(F_2)$
$L(G, v) \leftarrow L(F_2, v)$ for all $v \in V_i$
3.4. Delete G_i from H.
3.5. If the vertex x is now a leaf of H
Then delete x from H.

4. For each distinguished biconnected component G_i
Do
4.1.1. Create a copy of G_i, to be referred to as G_i
4.1.2. $V_i \leftarrow$ {the vertices of G_i}
4.1.3. $D(G_i) \leftarrow 0$; $L(G_i, v) \leftarrow L(G, v)$ for all $v \in V_i$
4.1.4. Clean (G_i)
4.2. If G_i was changed in Instruction 4.1.4,
Then
4.2.1. Create a copy of the current G_i, to be referred to as F
$U_i \leftarrow$ {vertices of G_i}
$D(F) \leftarrow 0$
$L(F, v) \leftarrow L(G_j, v)$ for all $v \in U_i$
Graph(F)
$D(G_i) \leftarrow D(G_i) + D(F)$
$L(G_i, v) \leftarrow L(F, v)$ for all $v \in U_i$
Else
4.2.2. Select
4.2.2.1. When there is a vertex x in G_i with $\deg(x) > 3$,
Then create F_1, F_2, two copies of G_i
$U_i \leftarrow$ {vertices of G_i}
For $j=1,2$
$D(F_j) \leftarrow 0$
$L(F_j, v) \leftarrow L(G_i, v)$ for all $v \in U_i$
$L(F_1, x) \leftarrow 1$

For all vertices $y \in F_2$ such that (x, y) is an edge in F_2,
  $L(F_2, y) \leftarrow 1$
For $j = 1,2$
  Clean(F_j); Graph(F_j)
If $D(F_1) \leq D(F_2)$
Then $D(G_i) \leftarrow D(G_i) + D(F_1)$
  $L(G_i, v) \leftarrow L(F_1, v)$ for all $v \in U_i$
Else $D(G_i) \leftarrow D(G_i) + D(F_2)$
  $L(G_i, v) \leftarrow L(F_2, v)$ for all $v \in U_i$

4.2.2.2. When there is a 'diamond' (Fig. 1)
  Then $L(G_i, x) \leftarrow 1$;
  Clean (G_i); Graph(G_i)

4.2.2.3. When there is a 'house' (Fig. 2)
  Then $L(G_i, u) \leftarrow 1$
  Clean(G_i); Graph(G_i)

4.2.2.4. When there is a $K_{2,3}$ (Fig. 3)
  Then $L(G_i, u) \leftarrow$ "$\neg x$"; $L(G_i, w) \leftarrow$ "$\neg x$";
  $L(G_i, y) \leftarrow$ "x"; $L(G_i, z) \leftarrow$ "x";
  Combine (G_i, x, y); Combine(G_i, x, z)
  Clean(G_i); Graph(G_i)

4.2.2.5. When there is a 'double triangle' (Fig. 4)

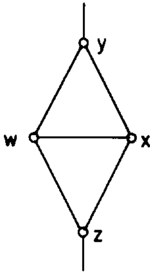


Fig. 1. A 'diamond'.

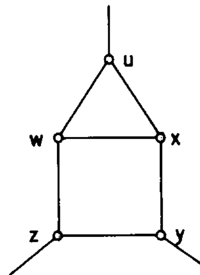


Fig. 2. A 'house'.

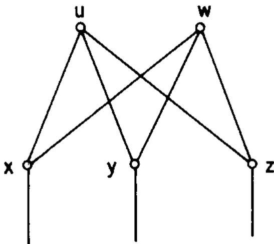


Fig. 3. $K_{2,3}$.

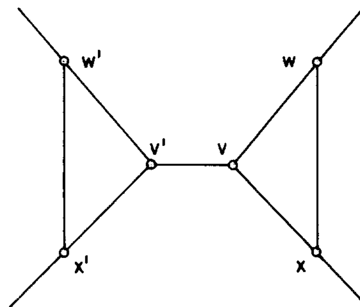


Fig. 4. A 'double triangle'.

Then create F_1, F_2, two copies of G_i
  $U_i \leftarrow \{\text{vertices of } G_i\}$
  For $j = 1,2$
    $D(F_j) \leftarrow 0$
    $L(F_j, v) \leftarrow L(G_i, v)$ for all $v \in U_i$
  $L(F_1, v') \leftarrow 1$; $L(F_1, v) \leftarrow 0$
  $L(F_1, w) \leftarrow 1$; $L(F_1, x) \leftarrow 1$;
  $L(F_2, v) \leftarrow 1$; $L(F_2, v') \leftarrow 0$
  $L(F_2, w') \leftarrow 1$; $L(F_2, x') \leftarrow 1$;
  For $j = 1,2$
    Clean(F_j); Graph(F_j)
  If $D(F_1) \leq D(F_2)$
  Then $D(G_i) \leftarrow D(G_i) + D(F_1)$
    $L(G_i, v) \leftarrow L(F_1, v)$for all $v \in U_1$
  Else $D(G_i) \leftarrow D(G_i) + D(F_2)$
    $L(G_i, v) \leftarrow L(F_2, v)$ for all $v \in U_i$

4.2.2.6. Otherwise choose a vertex x not involved in a triangle
  (* Such an x exists; see Lemma 6.1 *)
  Create F_1, F_2, two copies of G_i
  $U_i \leftarrow \{\text{vertices of } G_i\}$
  For $j = 1,2$
    $D(F_j) \leftarrow 0$
    $L(F_j, v) \leftarrow L(G_i, v)$ for all $v \in U_i$
  $L(F_1, x) \leftarrow 1$
  For all vertices $y \in F_2$ such that (x, y) is an edge in F_2,
    $L(F_2, y) \leftarrow 1$
  For $j = 1,2$
    Clean(F_j); Graph(F_j)
  If $D(F_1) \geq D(F_2)$
  Then $D(G_i) \leftarrow D(G_i) + D(F_1)$
    $L(G_i, v) \leftarrow L(F_1, v)$ for all $v \in U_i$
  Else $D(G_i) \leftarrow D(G_i) + D(F_2)$
    $L(G_i, v) \leftarrow L(F_2, v)$ for all $v \in U_i$

4.3. $D(G) \leftarrow D(G) + D(G_i)$; For all $v \in V_i$, $L(G, v) \leftarrow L(G_i, v)$
  Od
5. Return
End Graph

5. Procedure Clean

A detailed description of Procedure Clean is followed by a statement of this procedure.

As was mentioned above, Procedure Clean(G) deletes all vertices in G that are known to belong, or known not to belong, to the vertex cover, as well as those labelled "w" or "$\neg w$". It then shrinks the rest of the graph into a smaller graph, 'equivalent' in terms of finding a vertex cover (as will be explained later), in which all vertices are of degree at least 3.

In this involved algorithm, we do not wish to discuss in too much detail the data-structures employed and the exact implementation.

During the procedure we *delete* edges or vertices, or *combine* vertices. We use three primitives for this purpose:

(1) *Delet*(G, e), or simply *Delete*(e), deletes the edge e from the Graph G.

(2) *Delete*(G, v), or *Delete*(v), deletes the vertex v and all its incident edges from G.

(3) *Combine*(G, v, w), or *Combine*(v, w), is used to combine two vertices, v and w, into one, v: for each neighbor x of w create a new edge (v, x). We refer to the original edge (x, w) as a *merged edge.*

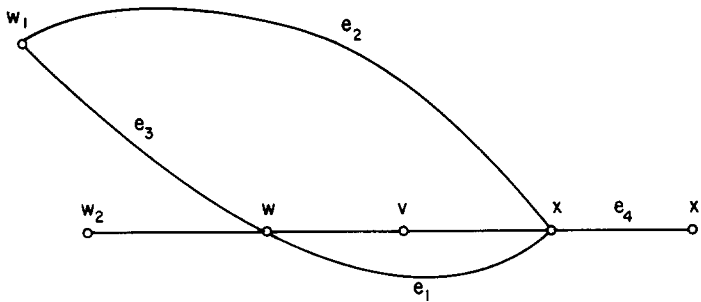
Whenever we apply Delete(e) or Delete(v), we move from our original graph G to a smaller graph G', so we have to make sure that the MVC problem on the original graph is 'equivalent' (this term is clarified later) to the same problem on the reduced graph in conjunction with the labelling L on vertices that have been deleted. Note that, although the algorithm calls, by the name G, both the original graph and the reduced one, in the present description we prefer to call the reduced graph G', for reasons of clarity.

The first two steps in the Procedure Clean were explained in Section 4.

The third step searches, in order, for self loops, parallel edges, vertices of degree 0, vertices of degree 1, adjacent pairs of vertices of degree 2, and other vertices of degree 2, and either eliminates them or prepares them for elimination by Steps 1 and 2.

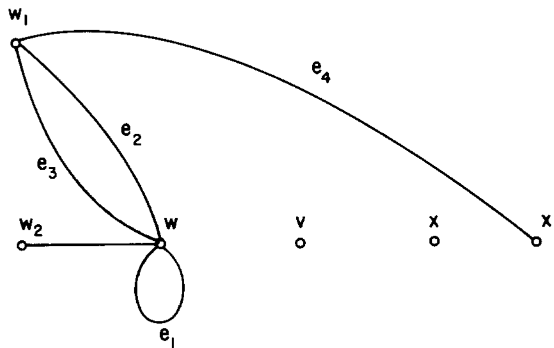
Let us explain Steps 3.5 and 3.6. (See also Fig. 5. Steps 3.1 through 3.4 are simpler and their explanations follow the same lines, and are therefore omitted.) Here $\deg(v) = 2$, and v is adjacent to x and w. If x is included in an MVC, we can demand that w be included as well, and that v be excluded. Reason: Take an MVC that includes x. Because of the edge (v, w), it includes either v or w, or both. We can obtain another MVC by including w and excluding v. Similarly, if w is included, we may demand that x is included and v excluded. If neither w nor x is included, obviously v is included. Thus x and w are both included or excluded together, while v has the opposite behavior. We reflect this situation by labelling x with "w", and v with "$\neg w$", and 'Combining' w and x. The 'Combine' operation attaches to vertex w in the reduced graph G', all edges incident on either x or w in the original graph G. Inclusion of w in an MVC of G' covers all edges incident on w in G', giving the effect of covering all edges incident on either w or x in G, thus corresponding to inclusion of both w and x in the MVC of G. Similarly, exclusion of w from the MVC of G' corresponds to exclusion of w and x (and inclusion of v) in the MVC of G.

In order to avoid searching the graph repeatedly, we may keep the vertices and



L(w) = L(v) = L(x) = 'Undefined'

Before the instruction



L(v) = "¬w", L(x) = "w"

Following the instruction

Fig. 5. The effect of instruction 3.6 (Procedure Clean).

edges to be handled by Steps 3.1 through 3.6 in queues, one queue for each step.

When Procedure Clean terminates, we are left with a graph G_r, all of whose vertices are of degree at least 3, with no self-loops or parallel edges. Further, $a(G_r) \leq k(G)$. (Note that when we call Clean(G), G is biconnected, implying $a(G) = k(G)$.) This is seen inductively. When processing a vertex of degree 0, we delete one vertex and remove one connected component, so that a does not change. When processing a vertex of degree 1, we delete two vertices (v and w), delete deg(w) edges, and increase the number of connected components by at most deg(w) − 2, so that a either decreases in value or stays constant. When processing a vertex of degree 2, we end up deleting two vertices and two edges (as well as moving several other edges), and not affecting the connectivity, so that a does not change in value. Similarly, we check that elimination of multiple edges or loops cannot increase a.

Clean(G) cannot add vertices or edges to other biconnected components, and therefore cannot change their values of k.

Since all vertices of G_r are of degree at least 3, we have $|E_r| \geq \frac{3}{2}|V_r|$, where E_r, V_r are the edges and vertices of G_r, respectively. Let C_r be the number of connected components of G_r. Then $k(G) = a(G) \geq a(G_r) = |E_r| - |V_r| + C_r \geq \frac{1}{2}|V_r|$. Thus $|V_r| \leq 2k(G)$.

Procedure Clean(G)

1. For every v such that $L(G, v) = 0$ or $L(G, v) =$ "$\neg w$"
   Delete(v)
2. For every v such that $L(G, v) = 1$ or $L(G, v) =$ "w"
   Delete(v); $D(G) \leftarrow D(G) + 1$
3. Select
   3.1. When there exists an edge e such that $e = (v, v)$ (* self-loop *)
        Then $L(G, v) \leftarrow 1$
   3.2. When there exist parallel edges e_1, e_2 between u and v
        Then Delete(e_1)
   3.3. When there exists a vertex v such that deg(v) = 0
        Then $L(G, v) \leftarrow 0$
   3.4. When there exists a vertex v such that deg(v) = 1, ((v, w) an edge)
        Then $L(G, v) \leftarrow 0$; $L(G, w) \leftarrow 1$
   3.5. When there exist vertices v, x such that deg(v) = deg(x) = 2, (v, x) is an edge, and (v, w) is another edge
        Then $L(G, v) \leftarrow$ "$\neg w$"; $L(G, x) \leftarrow$ "w"; Combine(w, x)
   3.6. When there exists a vertex v such that deg(v) = 2 ((v, x), (v, w) are edges, deg(x) $\leq$ deg(w))
        Then $L(G, v) \leftarrow$ "$\neg w$"; $L(G, x) \leftarrow$ "w"; Combine(w, x)
   3.7. Otherwise
        Return
4. Go to 1.

6. Detailed description of Step 4.2.2 of Procedure Graph

We describe here the way Procedure Graph(G) deals with the distinguished biconnected components. Throughout the discussion, we will derive conclusions regarding the running time of Graph. Recall that, at this point, all vertices are of degree at least 3, self-loops and parallel edges have been eliminated, and no vertices in G have been labelled.

Step 4.2.2.1

Step 4.2.2.1 treates a vertex x of degree at least 4. We branch on the vertex x.

Consider two possibilities: either x will be labelled 1 or 0. If 1, then we need to find a minimum cover V_1 of F_1, the subgraph of G induced by deleting x and all its incident edges. If 0, then all of x's neighbors in G will be labelled 1, and we need to find a minimum cover V_2 of F_2, the subgraph of G induced by deleting x, all its neighbors, and all incident edges.

In the first case (x labelled 1), the graph F_1 is still connected, since G was biconnected. (If w and y are two neighbors of x in G, edges (x, w) and (x, y) are contained in a simple cycle in G; this cycle contains a path from w to y not involving x, and this path is a path in F_1. Also, each vertex in F_1 is connected to a neighbor of x by a path in F_1.) Now $a(F_1)$ is gotten from $a(G)$ by noting that we have deleted one vertex and at least four edges, and have not changed the connectivity; thus $a(F_1) \leq a(G) - 3$.

For the second case (x labelled 0), we claim that $a(F_2) \leq a(G) - 4$. This requires more careful analysis. For each pair of neighbors w_i and w_j of x, there is a path from w_i to w_j not involving x. Create an auxiliary graph J, whose vertices are the neighbors w_i of x, and with an edge joining w_i and w_j if there is a simple path from w_i to w_j in G involving neither x nor any other w_k. By biconnectivity of G, J is connected. Find a spanning tree T of J. Let w_1 be a leaf of this spanning tree. The unique edge in T containing w_1 corresponds to a simple path from w_1 to, say, w_2, involving neither x nor any other w_k. The first edge of this path is, say, (w_1, y), where $y \neq x$. Since $\deg(w_1) \geq 3$, there is an edge (w_1, z) involving w_1, where $z \neq y$ and $z \neq x$.

Consider the graph G_t obtained from G by deleting x, all its incident edges, and the edge (w_1, z). We claim that G_t is connected. To show this, it suffices to show that each pair (w_i, w_j) is connected by a path in G_t, as is the pair (w_i, z) for some i. Clearly the paths represented by the edges in T have not been disturbed by the deletion of x and the several edges, so that all the w_i are still connected. Biconnectivity of G implies that (w_1, z) and (w_1, x) are contained in a simple cycle. Part of that cycle is a path (z, w_1, x, w_i) for $i \neq 1$. The rest of the cycle is a path from w_i to z; it contains none of the deleted edges (x, w_j) or (w_1, z), so that it is a path in G_t.

Since G_t is still connected, $a(G_t)$ is gotten from $a(G)$ by noting that one vertex (x) and $(\deg(x) + 1)$ edges have been deleted. Thus $a(G_t) = a(G) - \deg(x) \leq a(G) - 4$.

Label each w_i of G_t by 1. In turn we delete each w_i and each incident edge. This cannot increase the a value, as we discussed in the description of the Procedure Clean. So $a(F_2) \leq a(G_t) \leq a(G) - 4$.

So, faced with a biconnected graph G and the value $k(G)$, where G had a vertex of degree at least 4, we have transformed the problem into consideration for two smaller graphs F_1 and F_2, with $a(F_1)$ and $a(F_2)$ bounded by $k(G) - 3$ and $k(G) - 4$, respectively. Let $T_1(s)$ be the worst-case total running time of Graph(G) on a biconnected graph G, where $k(G) = s$, $\deg(x) \geq 3$ for all $x \in G$, and G includes a vertex of degree at least 4. We have:

$$T_1(s) \leq H(s-3) + H^*(s-4) + cs,$$

where the cs (c a constant) reflects, among other things, the time spent searching for a vertex of degree ≥ 4, and making two copies of the graph. (Recall the definitions of H and H^* from the end of Section 4.)

Step 4.2.2.2

There is a diamond, with vertices w, x, y, z; edges (w, x), (w, y), (w, z), (x, y), and (x, z). (See Fig. 1.) Recall all vertices are of degree 3. Then the only edges involving w or x are those listed. We wish to reduce G to a graph without this diamond. The edge (w, x) forces either x or w to belong to the MVC. We may demand that x belong to the MVC, because an MVC containing w but not x may be easily transformed to an MVC containing x and not w. So, set $L(G, x) = 1$.

Later, when we call Clean, we will delete x and its incident edges, without affecting connectivity. So $k(G)$ will decrease by 2. In this case the amount of work required is:

$$T_2(s) \leq H(s-2) + cs,$$

the term cs representing the time spent searching for the diamond, plus the time spent previously searching (unsuccessfully) for a vertex of degree ≥ 4.

Step 4.2.2.3

There is a house (vertices u, w, x, y, z; edges (u, w), (u, x), (w, x), (x, y), (y, z), (w, z); see Fig. 2). Again all nearby vertices are of degree 3, so that w and x have no other edges, while u, y, and z each has one edge not listed.

We can demand that u belongs to our MVC. The reason: at least two vertices of the triangle (u, w, x) must belong to any MVC. Assume that w and x belong to a certain MVC, while u does not. We have to show that we can transform this MVC into one which contains u. Because of edge (y, z), at least one vertex among (y, z) is contained in the MVC. If y is included, then exclude x and include u. Otherwise, z is included, so we exclude w and include u. One can check that in either case we still have an MVC.

So, set $L(G, u) = 1$. Later, when we call Clean, we will delete u and its incident edges, without affecting connectivity. (G remains connected because it was originally biconnected.) So $k(G)$ will decrease by 2. In this case the amount of work required is:

$$T_3(s) \geq cs + H(s-2).$$

Step 4.2.2.4

There is a $K_{2,3}$ (vertices u, w, x, y, z; edges (u, x), (u, y), (u, z), (w, x), (w, y), (w, z); see Fig. 3). All vertices have degree 3, so that u and w have no edges other than those listed, while x, y, and z each has another edge.

We can demand that either our MVC contains x, y, and z and excludes u and w, or it contains u and w and excludes x, y, and z.

Reason: if one of x, y, or z (say x) is included in an MVC, then at least two of the other four vertices have to be included. Exchange these vertices for y and z. We can easily check that we still have an MVC in this case. Otherwise (x, y, and z were all excluded), u and w were included, and our original MVC satisfied our demand.

In the algorithm, we 'merge' (Combine) the three vertices x, y, and z, into a single vertex x, and eliminate u and w. The labelling $L(G,u)=L(G,w)=$ "$\neg x$", $L(G,y)=L(G,z)=$ "x", reflects our demand, above.

Here we eliminate 4 vertices and 6 edges, without affecting connectivity, so that the amount of work is:

$$T_4(s) \leq H(s-2)+cs.$$

Step 4.2.2.5

There is a 'double triangle' (vertices v, w, x, v', w', x'; edges (v,w), (v,x), (w,x), (v',w'), (v',x'), (w',x'), (v,v'); see Fig. 4). All vertices have degree 3.

Here we break into two cases. (A) If v belongs to our MVC, we can demand that w' and x' also belong to the MVC, and v' be excluded. Reason: Take an MVC that includes v. Since at least two of the vertices x', v', w' must be included, obtain from this MVC an MVC in which w' and x' are included while v' is excluded.

(B) If v does not belong to the MVC, then v', w, and x must belong.

In each subgraph, after Clean has deleted labelled vertices and their adjacent edges, we find that 4 vertices and 8 edges have been deleted. Biconnectivity of G implies that the remaining graph is still connected (with a little work but no new ideas). Thus in this case the running time is

$$T_5(s) \leq 2H(s-4)+cs.$$

Step 4.2.2.6

Lemma 6.1. *Upon arriving at this step, there is a vertex x not involved in a triangle (i.e., x has neighbors y_1, y_2, y_3, but none of the edges (y_i, y_j) exists).*

Proof. Suppose (x, y_1, y_2) is a triangle. Then either y_3 is connected to y_1 or y_2 (and we have a 'diamond'), or y_3 is involved in another triangle (giving us a 'double triangle'), or y_3 is not involved in any triangle, in which case we select it instead of x. □

All vertices are of degree 3. Thus each y_i has neighbors z_{i1}, z_{i2}, and x. Some of the z_{ij} may coincide, but they are unequal to y_k.

Call G_D the graph obtained from G by deleting x, and y_i, and their incident edges.

Theorem. *Either G_D is a tree, or none of its connected components is a tree.*

Proof. Let a component of G_D be a tree S with t vertices. The sum of the degrees of these vertices in G_D is $2(t-1)$, while the sum of their degrees in G is $3t$. Thus among these t vertices are various z_{ij} incident on exactly $3t-2(t-1)=t+2$ of the edges (y_k, z_{ij}). Clearly $t \leq 4$, since only six such edges exist.

If $t=4$, then G_D is itself a tree.

If $t<4$, then its vertices are all z_{ij}, since the other vertices are of degree 3 in G_D.

If $t=1$, some z_{ij} is an isolated vertex in G_D. Then it must be adjacent to y_1, y_2, and y_3 in G, forming (with x) a $K_{2,3}$, a contradiction.

If $t=2$, then S is a single edge joining two z_{ij}. Each z_{ij} is adjacent to two y_k. Thus, there is a y_k (say y_1) adjacent to both of them, and another (say y_2) adjacent to at least one of them (say z_{12}). Then $(z_{11}, y_1, z_{12}, x, y_2)$ form a house, a contradiction.

If $t=3$, then the z_{ij} attract five of the six edges (y_k, z_{ij}). The remaining z_{ij} is then an articulation point of G, a contradiction. □

By biconnectivity of G, and nonexistence of $K_{2,3}$ (see the proof of the previous Theorem), we can show that each connected component of G_D has at least two different z_{ij}'s. So G_D has at most three connected components. Let us assume first that G_D has three connected components, and denote their a values by a_1, a_2, and a_3. By deleting x and its edges from G we get a connected graph with a value of a: $a(G)-2$. By deleting x, the y_i, and their incident edges (4 vertices and 9 edges), we decrease k by 5, but as we disconnect the graph into three pieces, the resulting equation on the a_i is

$$a_1+a_2+a_3=a(G)-5+2.$$

By the Theorem, each $a_i \geq 1$, so $a_i \leq a(G)-5$. The same holds if G_D has two or one connected component(s).

The timing in this case is then:

$$T_6(s) \leq H(s-2)+H'(s)+cs,$$

where

$$H'(s)=$$
$$\mathrm{Max}\{H(s-5), \mathrm{Max}_1(H(k_1)+H(k_2)),\ \mathrm{Max}_2(H(k_1)+H(k_2)+H(k_3))\}$$

where Max_1 is taken over the set $\{k_1, k_2 \mid k_1+k_2=(s-5)+1, k_i \geq 1\}$, and Max_2 is taken over the set $\{k_1, k_2, k_3 \mid k_1+k_2+k_3=(s-5)+2, k_i \geq 1\}$.

7. Complexity analysis

Recall the definitions of $H(n,s)$, $H(s)$, and $F(s)$, from the end of Section 4, and $T_i(s)$ as defined above.

In view of Procedure Graph, we find that

$$H(n,s) \le \max\{c_1(n_1+s_1^2)+H(s_1)+2\sum_{i>1}(c_1(n_i+s_i^2)+H(s_i))\}$$
$$\simeq c(n+s)+\max\{c_1 s_1^2+H(s_1)+2\sum_{i>1}(c_1 s_i^2+H(s_i))\}$$

where the maximum is taken over tuples of positive integers s_i, such that $\sum_i s_i = s$, and $s_1 = \max(s_i)$. For a biconnected component G_i, $s_i = k(G_i)$, and n_i is the number of vertices in G_i. We have $n+s \ge n \ge \frac{1}{2}\sum n_i$.

Here c_1 and c are constants, reflecting the linear time for finding biconnected components, processing self-loops, multiple edges, vertices of degree 0 or 1, and pairs of adjacent vertices of degree 2 (Steps 3.1 through 3.5, Procedure Clean). The s_i^2 term comes from the Combine operation in Step 3.6 of Procedure Clean; this step is exercised at most $2s$ times and requires at most s steps at each invocation. (When we reach Step 3.6, the graph has at most $2s$ vertices, and each call to Procedure Combine eiminates a vertex.) The remaining terms $(H(s_i))$ reflect the difficulty of processing the biconnected components G_i.

We bound $H(s)$ as follows:

$$H(s) \le \max\{c_1 s+F(s), \max\{c_1 s_1^2+H(s_1)+2\sum_{i>1}(c_1 s_i^2+H(s_i))\}\}.$$

The $cs+F(s)$ term comes from the possibility that G is biconnected; if G is not biconnected, the preceding analysis goes through, with s_1 the k-value of the largest biconnected component, $\sum s_i = s$.

For the case of G biconnected, we find

$$F(s) \le \max(T_1(s), \ldots, T_6(s)).$$

We find that $T_6(s)$ implies the recurrence relation with the worst behavior, namely

$$F(s) \le H(s-2)+H(s-5)+cs.$$

To solve this recurrence relation, we let θ be the solution of $1=\theta^{-2}+\theta^{-5}$, i.e., $\theta = 1.2365$. Then we find that the following is a solution to our recurrence relation:

$$F(s) = c_1\theta^s + c_2 s^3,$$

$$H(s) = \frac{s}{s_1}(c_1\theta^{s_1}+c_2 s_1^3),$$

$$H(n,s) = O\left(n+\frac{s}{s_1}(\theta^{s_1}+s_1^3)\right).$$

Since for large $s_1, s_1^3 \le \theta^{s_1}$, we omit s_1^3. In terms of our original parameters, the complexity of our algorithm is

$$H(n,a(G)) = O\left(n+\frac{a(G)}{k(G)}\theta^{k(G)}\right).$$

As $\theta < 2^{1/3}$, we get that the complexity of our algorithm is $O(n + (a/k)2^{k/3})$. This result is valid for unconnected graphs, as well.

8. A parallel implementation

The framework of the parallel implementation is the same as in [2] and follows the ideas of the following theorem and its proof.

Theorem (Brent). *Any synchronized parallel algorithm that runs in parallel time d and consists of x elementary operations can be implemented by p processors within time* $\lceil x \rceil + d$.

Proof. Let x_i denote the number of operations performed by the algorithm at step i, $(\sum_{i=1}^{d} x_i = x)$. We now use the p processors to simulate the algorithm. Since all the operations at step i can be executed simultaneously, they can be computed by p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_{i=1}^{d} \lceil x_i/p \rceil \leq \sum_{i=1}^{d} (x_i/p + 1) \leq \lceil x/p \rceil + d. \quad \square$$

Remark. The proof of Brent's theorem poses an implementation problem: how to assign the processors to their jobs.

Let us go back to our algorithm. Assume that we had all the processors we might have needed in every time unit of the algorithm, and that we could assign these processors to their jobs in constant time.

We start by calling the biconnectivity algorithm as in the sequential case. The operation of Procedure Clean at the first time we arrive at a biconnected component is also the same. The parallel implementation begins to differ from the sequential implementation only after the biconnected component is reduced to a graph with $\leq 2s$ vertices, each of them of degree ≥ 3, where s is the k of the biconnected component. We branch over vertices, and continue processing each copy in parallel.

The k of each copy is smaller by at least a constant from the k of the biconnected component before the branching. Therefore there are $\leq s$ branching in a row. Forgetting, for a while, the Combine operation of Procedure Clean, the number of operations between two branchings is $\leq cs$, and for the parallel implementation from the moment it 'really' started $\leq cs^2$.

Let us go back to the Combine operations. From the moment we started to branch over the biconnected component, the number of operations in a row is $\leq 2s$, the number of vertices; each takes $\leq cs$ time units. So, we can bound by $O(s^2)$ the time in which the algorithm deals with a biconnected component with $\leq 2s$ vertices each

of degree ≤ 3 where s is its k. This implies that the d of the theorem is $O(n+a+(a/k)(k^2))$.

The problem of assigning processors to their jobs is solved by the following instruction added where branchings of Graph over vertices is done. If p (>1) processors are to be assigned between two choices, one taking sequential time $\leq T_1$ and the other $\leq T_2$, then $T_1 p/(T_1+T_2)$ processors are assigned to the first choice and the rest to the other. (Precomputation of $H(k)$ for $k \leq s$ is sufficient to know T_1 and T_2.) If there is one processor, it takes care of both choices, as in the sequential case. The parallel time we get is, therefore

$$O(n+(a/k)\theta^k/p) \quad \text{for } p \leq \frac{n+(a/k)\theta^k}{n+a+(a/k)(k^2)} \text{ processors.}$$

It is not too difficult to see from the outline of Brent's theorem that it is the same as to say that the time is

$$O\left(n+(a/k)\left(\frac{\theta^k}{p}\right)\right) \quad \text{for } p \leq \frac{\theta^k}{k^2} \text{ processors.}$$

Remark. A further parallelization of our algorithm is possible. Many algorithms on rooted trees that work from the leaves to the root can be parallelized using the 'centroid decomposition' technique (see, for instance, [3]) in order to get $O(\log^2 n)$ parallel time instead of $O(n)$ sequential time, where n is the number of vertices. This, in conjunction with the recent parallel biconnectivity algorithm of [6] can be used in order to get parallel time $O(k \log^2 n)$ using sufficiently many processors. Since this involves utilization of known methods and seems tedious, we do not elaborate on this but rather leave it as an exercise for the interested reader.

References

[1] M.R. Garey and D.S. Johnson, Computers and Intractability – A Guide to the Theory of NP-Completeness (W.H. Freeman, San Francisco, 1979).

[2] Y. Gurevich, L. Stockmeyer and U. Vishkin, Solving NP-hard problems on graphs that are almost trees and an application to facility location problems, J. ACM 31 (1984) 459–473.

[3] N. Megiddo, Applying parallel computation in the design of serial algorithms, J. ACM 30 (1983) 852–865.

[4] R.E. Tarjan, Depth first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.

[5] R.E. Tarjan and A.E. Trojanowski, Finding a maximum independent set, SIAM J. Comput. 6 (1977) 537–546.

[6] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, TR-69, Dept. of Comp. Sci., Courant Inst., New York Univ., 251 Mercer St., NY 10012, to appear in SIAM J. Comput.

[7] U. Vishkin, Synchronous parallel computation – a survey, TR-71, Dept. of Comp. Sci., Courant Inst., New York Univ., 251 Mercer St., NY 10012.