Linear time algorithms for weighted offensive and powerful alliances in trees

Ararat Harutyunyan
Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 7, France.
email: ararat.harutyunyan@ens-lyon.fr

Sylvain Legay
Laboratoire de Recherche en Informatique
Université Paris-Sud
91405 Orsay, France
email: legay@lri.fr

February 26, 2015

Abstract

Given a graph $G = (V, E)$ with a positive weight function w on the vertices of G, a global powerful alliance of G is a subset S of V such that for every vertex v at least half of the total weight in the closed neighborhood of v is contributed by the vertices of S. A global offensive alliance is a subset S of V such that the above condition holds for every vertex v not in S. Finding the smallest such set in general graphs is NP-complete for both of these problems, even when the weights are all the same. In this paper, we give linear time algorithms that find the smallest global offensive and global powerful alliance of any weighted tree $T = (V, E)$.

Keywords: Global weighted alliances, offensive alliance, powerful alliance, trees, algorithm.

1 Introduction

The study of alliances in graphs was first introduced by Hedetniemi, Hedetniemi and Kristiansen [11]. They introduced the concepts of defensive and

offensive alliances, global offensive and global defensive alliances and studied alliance numbers of a class of graphs such as cycles, wheels, grids and complete graphs. The concept of alliances is similar to that of *unfriendly partitions*, where the problem is to partition $V(G)$ into classes such that each vertex has at least as many neighbors outside its class than its own (see for example [1] and [12]). Haynes et al. [9] studied the global defensive alliance numbers of different classes of graphs. They gave lower bounds for general graphs, bipartite graphs and trees, and upper bounds for general graphs and trees. Rodriquez-Velazquez and Sigarreta [17] studied the defensive alliance number and the global defensive alliance number of line graphs. A characterization of trees with equal domination and global strong defensive alliance numbers was given by Haynes, Hedetniemi and Henning [10]. Some bounds for the alliance numbers in trees are given in [6]. Rodriguez-Velazquez and Sigarreta [14] gave bounds for the defensive, offensive, global defensive, global offensive alliance numbers in terms of the algebraic connectivity, the spectral radius, and the Laplacian spectral radius of a graph. They also gave bounds on the global offensive alliance number of cubic graphs in [15] and the global offensive alliance number for general graphs in [16] and [13]. The concept of powerful alliances was introduced recently in [3].

Given a simple graph $G = (V, E)$ and a vertex $v \in V$, the *open neighborhood* of v, $N(v)$, is defined as $N(v) = \{u : (u, v) \in E\}$. The *closed neighborhood* of v, denoted by $N[v]$, is $N[v] = N(v) \cup \{v\}$. Given a set $X \subset V$, the *boundary* of X, denoted by $\delta(X)$, is the set of vertices in $V - X$ that are adjacent to at least one member of X.

Definition 1.1. *A set $S \subset V$ is a* defensive alliance *if for every $v \in S$, $|N[v] \cap S| \geq |N[v] \cap (V - S)|$. For a weighted graph G, where each vertex v has a non-negative weight $w(v)$, a set $S \subset V$ is called a* weighted defensive alliance *if for every $v \in S$, $\sum_{u \in N[v] \cap S} w(u) \geq \sum_{u \in N[v] \cap (V-S)} w(u)$. A (weighted) defensive alliance S is called a* global (weighted) defensive alliance *if S is also a dominating set.*

Definition 1.2. *A set $S \subset V$ is an* offensive alliance *if for every $v \in \delta(S)$, $|N[v] \cap S| \geq |N[v] \cap (V - S)|$. For a weighted graph G, where each vertex v has a non-negative weight $w(v)$, a set $S \subset V$ is called a* weighted offensive alliance *if for every $v \in \delta(S)$, $\sum_{u \in N[v] \cap S} w(u) \geq \sum_{u \in N[v] \cap (V-S)} w(u)$. A (weighted) offensive alliance S is called a* global (weighted) offensive alliance *if S is also a dominating set.*

Definition 1.3. *A* global (weighted) powerful alliance *is a set $S \subset V$ such*

that S is both a global (weighted) offensive alliance and a global (weighted) defensive alliance.

Definition 1.4. *The* global powerful alliance number *of G is the cardinality of a minimum size global (weighted) powerful alliance in G, and is denoted by $\gamma_p(G)$. A minimum size global powerful alliance is called a $\gamma_p(G)$-set.*

Definition 1.5. *The* global offensive alliance number *of G is the cardinality of a minimum size global (weighted) offensive alliance in G, and is denoted by $\gamma_o(G)$. A minimum size global offensive alliance is called a $\gamma_o(G)$-set.*

There are many applications of alliances. One is military defence. In a network, alliances can be used to protect important nodes. An alliance is also a model of suppliers and clients, where each supplier needs to have as many reserves as clients to be able to support them. More examples can be found in [11].

Balakrishnan et al. [2] studied the complexity of global alliances. They showed that the decision problems for global defensive and global offensive alliances are both NP-complete for general graphs. It is clear that the decision problems to find global defensive and global offensive alliances in weighted graphs are also NP-complete for general graphs.

The problem of finding global defensive, global offensive and global powerful alliances is only solved for trees. In [5], an $O(|V|^3)$ dynamic programming algorithm is given that finds the global defensive, global offensive and global powerful alliances of any weighted tree $T = (V, E)$.

In this paper, we give linear time algorithms that find the smallest global offensive and global powerful alliances of any weighted tree $T = (V, E)$. The algorithm for the global powerful alliances (with a proof sketch) first appeared in the conference article [8]. In this paper, we give a complete proof as well as provide an algorithm for global offensive alliances.

The paper is organized as follows. In Section 2, we present a linear algorithm for the global offensive alliance problem in an arbitrary weighted tree. In Section 3, we present a linear algorithm for minimum cardinality powerful alliance in weighted tree.

2 Weighted Offensive Alliances in Trees

Let $G = (V, E)$ be a graph and $w : V \to R^+$ a weight function. The *weighted global offensive alliance number* of G is the cardinality of the minimum

dominating set S with the property that for every $v \notin S$,

$$\sum_{u \in N[v]-S} w(u) \leq \sum_{u \in N[v] \bigcap S} w(u). \tag{1}$$

The above condition is called *the alliance condition of* v. In this section we present a linear time algorithm which finds the weighted global offensive alliance number of a tree.

First, we need some notation. We denote the set of all leaves of T by $L(T)$ and the set of all children of a vertex x which are leaves by $L(x)$. The parent of a leaf is called a *support vertex*. We denote the set of support vertices of T by $S(T)$. The *root* of the tree is denoted by $r(T)$. We denote by $p(v)$ the parent of vertex v. Also, for a set $S \subset V$ define $w(S) := \sum_{u \in S} w(u)$.

2.1 The Algorithm

We root the tree T at any vertex. Denote by d the depth of T. We build a minimum cardinality global offensive alliance set S. Throughout the algorithm we will label some of the vertices by " $-$ ". These vertices will not be included in the global offensive alliance (i.e., no vertex labelled " $-$ " will ever be put in S.)

Algorithm 1 Algorithm Weighted Offensive Alliance

for $v \in S(T)$ **do**
  for $l \in L(v)$ **do**
    if $w(l) > w(v)$ **then**
      Put l in S.
    else
      Label l with " $-$ "
    if there exists $l \in L(v)$ such that $w(l) \leq w(v)$ **then**
      Put v in S.
for vertices v at depth $d - i$, $i = 1$ **to** d **do**
  if $v \notin S$ **and** v is not labelled " $-$ " **and** all of v's children are labelled " $-$ " **then**
    if $w(p(v)) \geq w(\{N[v] - p(v)\})$ **then**
      put $p(v)$ in S (if it already is not) and label v with " $-$ ".
    else
      put v in S.
  else if $v \notin S$ **and** v is not labelled " $-$ " **and** v has at least one neighbor u already in S **then**
    if $w(N(v) \cap S) \geq w(\{N[v] - S\})$ **then**
      label v with " $-$ ".
    else if $w(N(v) \cap S \cup \{p(v)\}) \geq w(N[v] - S - \{p(v)\})$ **then**
      put $p(v)$ in S and label v with " $-$ " .
    else
      put v in S.

The correctness of the algorithm hinges on the following theorem.

Theorem 2.1. *After each iteration of the algorithm, the set S is contained in some minimum cardinality weighted global offensive alliance R, which also does not contain any of the vertices marked by " $-$ ".*

Proof. The proof is by induction on the number of iterations. First, suppose we are carrying out the first two nested FOR loops of the algorithm, so that a support vertex v is considered. Note that for every $l \in L(v)$, every $\gamma_o(T)$-set must contain v or l. If some vertex $l \in L(v)$ satisfies $w(l) > w(v)$, then clearly l must be put in S for otherwise the alliance condition for l would not be satisfied. Now, suppose there exists a vertex $l \in L(v)$ such that $w(l) \leq w(v)$. This implies that putting v in S satisfies the alliance condition of l. Since at least one of v or l must be put in S, it is safe to put v in S since this can only help to satisfy the alliance conditions of the necessary vertices in $N(v)$. Thus, the vertices added to S in the first two nested FOR loops of the algorithm are safe.

Now, assume we are considering a vertex v in the third FOR loop of the algorithm. By induction, all the vertices in S thus far are contained in some $\gamma_o(T)$-set. We first claim that every child u of v that has label " $-$ " has its

alliance condition already satisfied. This is clear if u is a support vertex or a leaf by the first two nested for loops. Now, suppose that u got labelled " $-$ " in a previous iteration of the third FOR loop of the algorithm. If u got labelled " $-$ " in line 10, then clearly its alliance condition was satisfied. It is similarly seen that if u gets labelled " $-$ " in line 15, then its alliance condition is satisfied as well. This establishes the claim.

If v is already in S or v is labelled with " $-$ ", no further vertices get labelled and the theorem holds. Now, assume we are in line 10. This implies that we must pick at least one vertex to be in the set S from $N[v]$ to satisfy the domination condition for v. If putting $p(v)$ in S is already sufficient to satisfy the alliance condition of v, then this is certainly safe to do so because choosing $p(v)$ can only help to satisfy the alliance conditions of vertices in $N[p(v)]$. Clearly, in this case, we do not take v to be in S. If putting $p(v)$ in S is not sufficient to satisfy the alliance condition of v, then at least one vertex from v and its children must be put in S. Now, it is clear that it is safe to put v in S for in that case we do not have to satisfy the alliance condition of v, and $v \in S$ can only help to satisfy alliance condition of $p(v)$, if necessary.

Now, assume we satisfy line 15 of the algorithm. In this case, v is clearly already dominated. First, we check whether v's alliance condition is already satisfied. If this is the case then we claim it is safe to label v with " $-$ ". Since the alliance conditions of v and all its children are satisfied we only need to consider the effect on $p(v)$. However, when we come to consider $p(v)$, given a choice to satisfy $p(v)$'s alliance condition by picking v, it would still be better to pick $p(v)$.

Second, we check whether putting $p(v)$ in S is sufficient to satisfy the alliance condition of v. As in the case of line 10, it is safe to put $p(v)$ in S. If putting $p(v)$ in S is not sufficient, then we put v in S. By an identical argument as in case of line 10, this is again a safe operation. This completes the proof. □

It is easy to check that the alliance condition of every vertex is satisfied at the end of the algorithm. This fact, combined with Theorem 2.1 proves that the algorithm finds an optimal global offensive alliance.

2.2 Complexity of the Algorithm

Now we consider the running time of the algorithm. Clearly, the first two nested FOR loops take at most $O(n)$ time. In the third FOR loop, at each vertex v, we take at most $O(\deg(v))$ time in each of the three cases.

Therefore, in total we take

$$O(n) + \sum_{v \in V} O(\deg(v)) = O(n)$$

time. Hence, the algorithm runs in $O(n)$ time.

3 Powerful Alliances

In this section, we give a linear time algorithm that finds the weighted global powerful alliance number of any tree. We assume all the weights are positive - the algorithm can be easily modified for the case where the weights are non-negative.

As before, for a set $S \subset V$ define $w(S) := \sum_{u \in S} w(u)$.

It is clear that when the weight function is positive, the global powerful alliance problem can be formulated as follows.

Observation 1. *Let $G = (V, E)$ be a graph, and $w : V \to R^+ \backslash \{0\}$ a weight function. Then a global powerful alliance in G is a set $S \subset V$ such that for all $v \in V$,*

$$\sum_{u \in N[v] \cap S} w(u) \geq \frac{w(N[v])}{2}.$$

Note that the condition that S is a dominating set is automatically guaranteed because all the weights are positive. We will use the above formulation in our algorithm. The following definition will be used throughout the algorithm.

Definition 3.1. *For $v \in V$, the* alliance condition for v *is the condition that $w(N[v] \cap S) \geq \frac{w(N[v])}{2}$.*

3.1 An overview of the algorithm

Now, we give a brief intuition behind the algorithm. As noted above, when all the weights are positive, the global powerful alliance problem is simply finding the smallest set $S \subset V$ such that for every vertex $v \in V$, $w(N[v] \cap S) \geq \frac{w(N[v])}{2}$. Our algorithm is essentially a greedy algorithm. We root the tree at a vertex, and start exploring the neighborhoods of vertices starting from the bottom level of the tree. The vertices which have already been chosen to be included in the alliance set S are labelled with "+". For each vertex v, we find the smallest number of vertices in its closed

neighborhood that need to be added to the vertices labelled "+" in v's closed neighborhood to satisfy the alliance condition for v. We do this using the algorithm FindMinSubset. In some cases we may get more than one optimal solution and it will matter which solution we pick (for example, if there is an optimal solution containing both v and $p(v)$ then this solution is preferable when we consider the neighborhood of $p(v)$). The complication arises when there is an optimal solution containing v, but not $p(v)$, and there is an optimal solution containing $p(v)$. If $w(v) > w(p(v))$, then it is not clear which solution is to be preferred because v is at least as good as $p(v)$ for satisfying the alliance condition for $p(v)$, but choosing $p(v)$ is preferable for satisfying the alliance condition for parent of $p(v)$, $p(p(v))$. In general, when we have to choose between a solution that contains v and one that contains $p(v)$ we give preference to the solution containing $p(v)$ unless: (i) we can immediately gain by choosing the solution with v due to choices made in previous iterations (ii) when satisfying the alliance condition for $p(v)$, it might theoretically be better to have chosen v.

3.2 Labels

In the algorithm, we use five labels for vertices: "+", "?=", "?≠", "?&=" and "?&≠". At each step of the algorithm, the set of vertices labelled "+" is a subset of a minimum cardinality powerful alliance. We assign a vertex v a label "+" when we can claim that it preserves this property. At the end of the algorithm, the set of vertices labelled "+" will be the vertices of a minimum cardinality global powerful alliance. Generally, we assign a vertex v a label "?≠" when we have a choice of taking v or $p(v)$ to satisfy the alliance condition for v (but we cannot choose both v and $p(v)$) and $w(v) > w(p(v))$. For a vertex u, we define $D(u)$ to be the set of all children of u that are labelled "?≠". We will argue that for every vertex u, we eventually must take u or $D(u)$ in our alliance.

Another special case is when we see no optimal solution containing $p(v)$ or v that would satisfy v's alliance condition, but if $p(v)$ were later labelled "+" for other reasons, then there would be an optimal solution containing v that would satisfy v's alliance condition. This solution is preferable since it can decrease the number of vertices required to satisfy $p(v)$'s alliance condition. In the aforementioned special case, we assign v a label "?=", and for every child u of v a label "?&≠" if u is in the optimal solution containing neither v nor $p(v)$, a label "?&=" if u is in the optimal solution containing both v and $p(v)$ and a label "+" if u is in both.

In the algorithm, we might want to label a vertex already labelled, in

which case the new label would replace the old.

3.3 Satisfying the alliance condition for a vertex

Throughout the algorithm, for every vertex v we need to find the smallest number of vertices necessary in the closed neighborhood of v to satisfy the alliance condition for v. To solve this problem, we use the following algorithm.

Given positive numbers $a_1, a_2, ..., a_n$, $FindMinSubset(a_1, a_2..., a_n)$ is the problem of finding the minimum k such that there is a k-subset of $\{a_1, a_2, ..., a_n\}$ the elements of which sum up to at least $\frac{1}{2}\sum_{i=1}^{n} a_i$. We give an algorithm that solves this problem in time $O(n)$. The algorithm also finds an instance of such a k-subset. Furthermore, out of all the optimal solutions it outputs a set with a maximum sum.

Algorithm FindMinSubset[A,n,T].
Input: An array A of size n of positive integers; a target value T.
Output: The least integer k such that some k elements of A add up to at least T.

1. If $n = 1$, return 1.
2. Set $i = \lceil \frac{n}{2} \rceil$.
3. Find the set A' of the i largest elements of A and compute their sum M.
4. If $M > T$ return $FindMinSubset[A', i, T]$; if $M = T$ return i; else return $i + FindMinSubset[A - A', n - i, T - M]$.

Lemma 3.2. *The above algorithm solves the problem $FindMinSubset(a_1, a_2, ..., a_n)$ in time $O(n)$. Furthermore, if the solution is k, the algorithm finds the k largest elements.*

Proof. Let $S = \sum_{i=1}^{n} a_i$. It is clear that $FindMinSubset[A, n, T = \frac{S}{2}]$ where A is the array of the elements $(a_1, a_2, ..., a_n)$ will solve the desired problem. The analysis of the running time is as follows. Note that finding the largest k elements in an array of size n can be done in linear time for any k (see [4]). Therefore, in each iteration of $FindMinSubset$, Step 2 and Step 3 take linear time, say Cn. Since the input size is always going down by a factor of 2, we have that the running time $T(n)$ satisfies $T(n) \leq T(n/2) + Cn$. By induction, it is easily seen that $T(n) = O(n)$. It is clear that out of all the possible solutions, the algorithm picks the one with the largest weight. □

As mentioned before, sometimes it will be useful to see the smallest number of vertices in $N[v]$ that need to be added to the alliance for v's

alliance condition under the condition that we force v or $p(v)$ (or both) to be taken. We will need the following algorithm.

Algorithm 2 FindMin[u, $take\_u$, $take\_p(u)$; $Num, list\_children$].

Input: A vertex u in the tree T; two boolean $take\_u$ and $take\_p(u)$ that indicate if we force u or $p(u)$ to be taken.
Output: The least integer Num such that some Num neighbours of u not already taken by the algorithm add up to at least $w(N[u])/2$; $list\_children$, the set of children of u counted by Num.

```
1: w_total ← w(N[u])
2: Num ← 0
3: w_ally ← 0
4: list_children ←∅
5: if take_u then
6:     Num ← Num + 1
7:     w_ally ← w_ally + w(u)
8: if take_p(u) then
9:     Num ← Num + 1
10:    w_ally ← w_ally + w(p(u))
11: set_children ← set of children of u
12: for each v in set_children labelled "+" do
13:    w_ally ← w_ally + w(v)
14:    remove v from set_children
       /*We don't need to change Num because v was already taken for another vertex.
       Taking it for u is free*/
15: if take_u then
16:    for each v in set_children labelled "?=" do
17:       w_ally ← w_ally + w(v)
18:       remove v from set_children
19:       add v to list_children
          /*We take u, so we can take v, for free, because it allows us to not take children
          of v labelled "?&≠"*/
20: else
21:    for each v in set_children labelled "?&≠" do
22:       w_ally ← w_ally + w(v)
23:       Num ← Num + 1
24:       remove v from set_children
25:       add v to list_children
          /* We do not take u, so we have to take all its children labelled "?&≠"*/
    /* We have taken every vertex which we are forced or free to take, now, we simply
    take enough of the other vertices to satisfy the alliance condition of u */
26: k ← FindMinSubset(set_children, sizeof(set_children), w_total − w_ally)
27: Num ← Num + k
28: add the set obtained by FindMinSubset to list_children
```

We now describe the algorithm for weighted powerful alliances in trees.

3.4 The Algorithm

We assume the tree T is rooted and has depth d. For each k, we order the vertices of depth k from left to right. By $C(v)$ we denote the set of children of v. We define $p(v)$ to be the parent of v.

Algorithm 3 *Label*(T).

for $i = 0$ **to** d **do**
  for vertices u at depth $d - i$ **do**
    if u is labelled "+" **then**
      Findmin(u, True, True; $min11$, $list\_11$)
      Findmin(u, True, False; $min10$, $list\_10$)
      $min01 \leftarrow \infty$
      $min00 \leftarrow \infty$
    else
      Findmin(u, True, True; $min11$, $list\_11$)
      Findmin(u, True, False; $min10$, $list\_10$)
      Findmin(u, False, True; $min01$, $list\_01$)
      Findmin(u, False, False; $min00$, $list\_00$)
    /*If u is already labelled "+", there is no need to check if there is a smaller set that avoids taking u */
    $min \leftarrow$ minimum of $\{min11, min10, min01, min00\}$
    if $min11 = min$ **then**
      label "+" every vertex in $list\_11$
    else
      if $min10 = min$ **or** $min01 = min$ **then**
        if $w(u) > w(p(u))$ **and** $min10 = min01$ **then**
          label "+" every vertex in $list\_10 \cap list\_01$
          label "?≠" the vertex u and every vertex in $list\_01 \backslash list\_10$
          label "?=" every vertex in $list\_10 \backslash list\_01$
        else
          if $min = min01$ **then**
            label "+" every vertex in $list\_01$
          else
            label "+" every vertex in $list\_10$
      else
        if $min00 + 1 = min11$ **then**
          label "+" every vertex in $list\_11 \cap list\_00$
          label "?&≠" every vertex in $list\_00 \backslash list\_11$ not labelled "?≠"
          label "?&=" every vertex in $list\_11 \backslash list\_00$
          /*Notice here that $list\_11$ does not contain u and $p(u)$ even when it is counted in $min11$, so $list\_00$ contains one more vertex than $list\_11$. We are labelling one more vertex "?&≠" than "?&="*/
          label "?=" the vertex u
        else
          label "+" every vertex in $list\_00$

Algorithm 4 Algorithm for finding Minimum Cardinality Weighted Powerful Alliance in a tree T.

```
1: Label(T)
2: for i = 0 to d do
3:     for vertices v at depth i do
4:         if v is labelled "?=" then
5:             if p(v) is labelled "+" then
6:                 label "+" the vertex v
7:             else
8:                 unlabel the vertex v
9:         if v is labelled "?≠" then
10:            if p(v) is labelled "+" then
11:                unlabel the vertex v
12:            else
13:                label "+" the vertex v
14:        if v is labelled "?&=" then
15:            if p(v) and p(p(v)) are labelled "+" then
16:                label "+" the vertex v
17:            else
18:                unlabel the vertex v
19:        if v is labelled "?&≠" then
20:            if p(v) and p(p(v)) are labelled "+" then
21:                unlabel the vertex v
22:            else
23:                label "+" the vertex v
```

3.5 Proof of Correctness

Let T be a weighted tree, rooted at a vertex r. It will be convenient to assume that r actually has a parent $p(r)$ of weight 0. We are going to prove by induction the following lemma. Note that MCPA denotes *Minimum cardinality powerful alliance* and for a vertex u, we will call a minimum cardinality powerful alliance in the subtree rooted at u, and denote by $MCPA(u)$, a smallest subset of $V(T)$ satisfying the alliance condition for every vertex in the subtree rooted at u. Note that this definition differs from the general definition of minimum cardinality powerful alliance in that $MCPA(u)$ can contain $p(u)$, yet $p(u)$ is not in the subtree rooted at u.

Lemma 3.3. *For each vertex u, let $P(u)$ be the property that one of the four conditions below hold. Let u be a vertex of T. Then, during the execution of the algorithm, if we skip line 3 to 34 of Algorithm 2 every time the vertex considered by the loop is not in the subtree rooted at u (meaning that we just look at satisfying the alliance condition of vertices in the subtree rooted at u), $P(u)$ holds at the end of Algorithm 2.*

1. *(a) u is labelled "+".*
   (b) The algorithm will yield a MCPA of the subtree rooted in u.
   (c) If $p(u)$ is unlabelled, then there is no MCPA of the subtree rooted in u containing $p(u)$.

2. *(a) u is labelled "? ="*
   (b) The algorithm will yield a MCPA of the subtree rooted in u.
   (c) If we label $p(u)$ or u or both with "+", then the algorithm will yield a powerful alliance of the subtree rooted at u of cardinality $|MCPA(u)|+1$
   (d) There is no MCPA of the subtree rooted in u containing u or $p(u)$

3. *(a) u is labelled "? $\neq$"*
   (b) The algorithm will yield a MCPA of the subtree rooted in u.
   (c) If we label $p(u)$ with "+", the algorithm will yield a MCPA of the subtree rooted in u.
   (d) If we label $p(u)$ and u with "+", the algorithm will yield a powerful alliance of the subtree rooted at u of cardinality $|MCPA(u)|+1$
   (e) There is no MCPA of the subtree rooted in u containing both u and $p(u)$

4. (a) u is unlabelled
(b) The algorithm will yield a MCPA of the subtree rooted in u.
(c) If we label u with "+", the algorithm will yield a powerful alliance of the subtree rooted at u of cardinality $|MCPA(u)| + 1$
(d) If $p(u)$ is unlabelled, then

- *If we label $p(u)$ with "+", the algorithm will yield a powerful alliance of the subtree rooted at u of cardinality $|MCPA(u)| + 1$*
- *If we label $p(u)$ and u with "+", the algorithm will yield a powerful alliance of the subtree rooted at u of cardinality $|MCPA(u)| + 2$*
- *There is no MCPA of the subtree rooted in u containing u or $p(u)$*
- *There is no powerful alliance of the subtree of u containing both u and $p(u)$ of cardinality $|MCPA(u)| + 1$*

(e) If $p(u)$ is labelled with "+", then $w(u) < w(p(u))$ or there is no MCPA of the subtree rooted in u containing u.

Proof. The proof is by induction. The algorithm goes from child to parent. We are trying to prove the lemma for the vertex $u \in T$. By induction, we may assume the lemma to be true for every child v of u. It means that for every child v of u if we apply step 3 to 34 of Algorithm 2 to every vertex in the subtree rooted in v (from child to parent), $P(v)$ holds. Notice that doing so does not label any vertex outside of u or the subtree rooted in v. We must check that if we apply step 3 to 34 of Algorithm 2 to every vertex in the subtree rooted u, $P(u)$ holds.

Let us first prove that if we apply step 3 to 34 of Algorithm 2 to every vertex in the subtree rooted u except u itself, and then label "+" the set of vertices returned by $FindMin(u, true, true)$ (respectively, $FindMin(u, true, false)$), and finally apply step 2 to 23 of Algorithm 3, then, the vertices labelled "+" will satisfy the alliance condition of every vertex in the subtree rooted at u and be of minimum cardinality with respect to the fact that u and $p(u)$ must be labelled "+" (respectively, that u must be labelled "+", but not $p(u)$). Let v be a child of u.

If v is labelled "+" before the execution of $FindMin$, we know that it satisfies the first property of $P(v)$. Thus, by part (c), if $p(v) = u$ was not labelled by step 3 to 34 of Algorithm 2 during the loop where v was considered, we know that there is no MCPA in the subtree rooted at v containing u. But we are forced to take u, so the best we can hope for is to have a

powerful alliance of the subtree rooted in v of cardinality $|MCPA(v)| + 1$. Taking the MCPA of the subtree rooted in v containing v that we have and adding u to it works and it ensure us that we will have both v and $p(v)$ which will allow us to take less vertex for overpowering u. There exist a minimum cardinality powerful alliance of the subtree rooted u which contains v, so $Findmin(u, true, true)$ (respectively $Findmin(u, true, false)$) must take v. It does take v because it is a child of u labelled "+".

If v is labelled "? =" before the execution of $FindMin$, we know that it satisfies the second property of $P(v)$. We know that there is no MCPA of the subtree rooted at v containing $p(v) = u$. But we are forced to take u, so the best we can hope is to have a powerful alliance of the subtree rooted in v of cardinality $|MCPA(v)| + 1$. We know we can do this by having v and u labelled with "+", which will allow us to take less vertices for the alliance condition of u. We want $Findmin(u, true, true)$ (respectively, $Findmin(u, true, false)$) to take v. It does take v because v is a child of u labelled "? =". Moreover, $Findmin$ should consider v as free, as a vertex labelled "+", because there is no powerful alliance of the subtree rooted in v of cardinality $|MCPA(v)|$ containing u, so by taking both u and v, we only augment the cardinality of the whole set by 1. Notice, that the algorithm does consider v as free.

If v is labelled "? $\neq$" before the execution of $FindMin$, we know that it satisfies the third property of $P(v)$. $p(v) = u$ is already labelled "+". Applying step 2 to 23 of Algorithm 3 without changing the label of v gives us a MCPA of the tree rooted at v. However, if we want to take $p(v) = u$ and v, we know that there is no MCPA of the tree rooted at v containing both v and $p(v) = u$, but labelling them both "+" then applying step 2 to 23 of Algorithm 3 gives us a powerful alliance of the tree rooted at v of cardinality $|MCPA(v)| + 1$. Thus, $Findmin(u, true, true)$ (respectively, $Findmin(u, true, false)$) return a set as claimed.

If v is unlabelled before the execution of $FindMin$, we know that it satisfies the fourth property of $P(v)$. $Findmin(u, true, true)$ (respectively $Findmin(u, true, false)$) can take v, but it will not help to take less vertices elsewhere.

This proves that $FindMin(u, true, true)$ (respectively, $Findmin(u, true, false)$) returns the set intended.

Let us prove now that if we apply step 3 to 34 of Algorithm 2 to every vertex in the subtree rooted u except u itself and u is not labelled, and then label "+" the set of vertices returned by $FindMin(u, false, true)$ (respectively, $FindMin(u, false, false)$), and finally apply step 2 to 23 of Algorithm 3, then, the vertices labelled "+" will satisfy the alliance condition of every vertex in the subtree rooted at u and be of minimum cardinality with respect to the fact that $p(u)$ must be labelled "+", but not u (respectively, that both u and $p(u)$ cannot be labelled "+"). Let v be a child of u.

If v is labelled "+" before the execution of $FindMin$, we know that it satisfies the first property of $P(v)$. As $p(v) = u$ was not labelled previously, we know that there is no MCPA of the subtree rooted in v containing $p(v) = u$. Line 2 to 23 of Algorithm 3 gives us a MCPA of the tree rooted in v containing v. Taking v allows us to take less vertices for satisfying the alliance condition of u. There exist a minimum cardinality powerful alliance of the subtree rooted u which contains v, so $Findmin(u, false, true)$ (respectively $Findmin(u, false, false)$) must take v. It does take v because it is a child of u labelled "+".

If v is labelled "? =" before the execution of $FindMin$, we know that it satisfies the second property of $P(v)$. We know that there is no MCPA of the subtree rooted in v containing v. Step 2 to 23 of Algorithm 3 gives a MCPA of the subtree rooted in v containing neither v nor $p(v) = u$. If we label v with "+", step 2 to 23 of Algorithm 3 gives a powerful alliance of the subtree rooted in v of cardinality $|MCPA(v)| + 1$. $Findmin(u, false, true)$ (respectively $Findmin(u, false, false)$) can take v, but it will not help to take less vertices somewhere else. Thus, $Findmin(u, true, true)$ (respectively, $Findmin(u, true, false)$) return a set as claimed.

If v is labelled "? $\neq$" before the execution of $FindMin$, we know that it satisfies the third property of P(v). $p(v) = u$ is not taken by $Findmin$. Applying step 2 to 23 of Algorithm 3 without changing the label of v gives a MCPA of the tree rooted at v containing v. A minimum cardinality powerful alliance not containing u must contain v, so $Findmin(u, false, true)$ (respectively $Findmin(u, false, false)$) must take v. It does take v because v is a child of u labelled "? $\neq$". However, contrary to other cases, the cost of taking v must still be counted, because taking u costs less than taking all of its children labelled "? $\neq$".

If v is unlabelled before the execution of $FindMin$, we know that it

satisfies the fourth property of $P(v)$.

This proves that $FindMin(u, false, true)$ (respectively, $Findmin(u, false, false)$) returns the set intended. Thus, Findmin take exactly as few vertices as it can to satisfy the alliance condition of u, so it gives a set in $N[u]$ such that labelling this set with "+" and applying step 2 to 23 of Algorithm 3 gives a powerful alliance of the subtree rooted in u of minimal cardinality with respect to the forced vertices.

We look at the minimum of the 4 instances of Findmin(u,.,.) and we know that it is the size of an MCPA(u). If Label() does the case in line 19 of the Algorithm 2, then $p(u)$ is labelled with "+" and u must satisfy the property 1, which it does. If Label() does the case in lines 23-24, then u must satisfy the property 3, which it does. If Label() does the case in line 26, then $p(u)$ is labelled with "+" and u must satisfy the property 4, which it does. If Label() does the case in line 27, then $p(u)$ is unlabelled and u must satisfy the property 1, which it does. If Label() does the case in lines 30-35, then u must satisfy the property 2, which it does. If Label() does the case in line 36, then $p(u)$ is unlabelled and u must satisfy the property 4, which it does.

By induction P(u) is true for every $u \in T$. □

If we apply this lemma with $u = r$ the root, It means that $P(r)$ is satisfied after having apply the algorithm. Given item 1)b, 2)b, 3)b, and 4)b, of $P(r)$, the algorithm yield a MCPA of the subtree rooted in r. This MCPA does not contain $p(r)$ because $w(p(r)) = 0$ and it would contradict the minimality, so it is a MCPA of T.

3.6 Complexity of the Algorithm Powerful Alliances for Trees

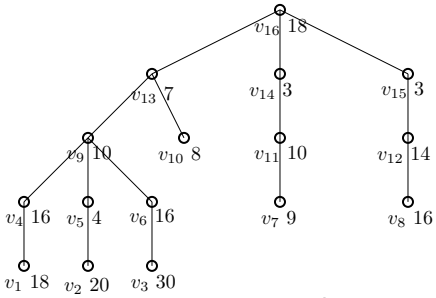
Here we analyze the running time of the algorithm.

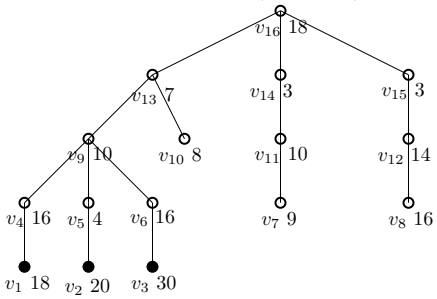
Lemma 3.4. *Algorithm 1 runs in time* $O(\deg(u))$.

Proof. It is clear that $FindMin$ will solve the desired problem. The analysis of the running time is as follows. It is easily seen that everything until line 26 of the algorithm is done in $O(\deg(u))$. The input size of FindMinSubset is $O(\deg(u))$ so the running time is $O(\deg(u))$. □

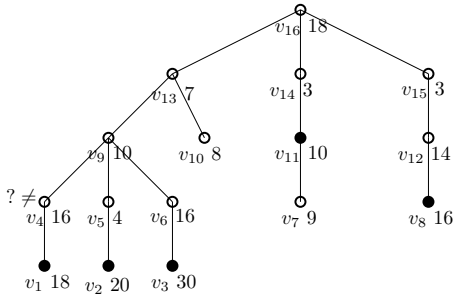
Theorem 3.5. *The running time of the algorithm Powerful Alliances for Trees is* $O(n)$.

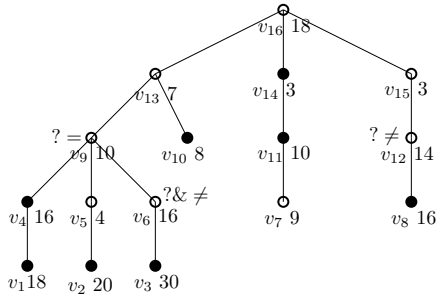
An illustration of the algorithm by an example (Fig. 1).



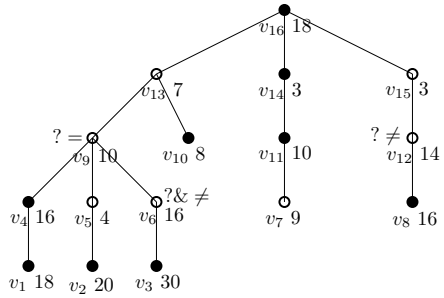


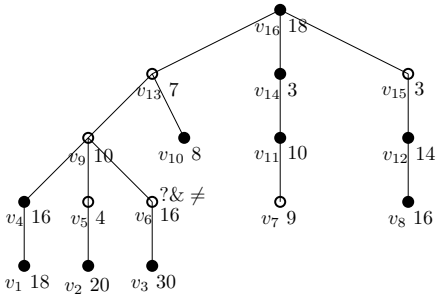
After vertices v_1,v_2,v_3 have been considered ($i = 0$ in the first loop)



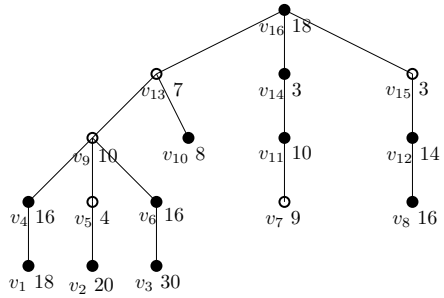


After vertices v_4,v_5,v_6,v_7,v_8 have been considered ($i = 1$) After vertices v_9,v_{10},v_{11},v_{12} have been considered ($i = 2$)





After vertices v_{13},v_{14},v_{15},v_{16} have been considered After vertices v_{16} to v_9 have been unlabelled ($i = 2$ in the second loop)



After vertices v_8 to v_4 have been unlabelled ($i = 3$)

Proof. At each step of Label(T), we compute Findmin(u) four times, then label all the neighbours of u accordingly. Findmin(u) has a running time in $O(\deg(u))$ so each iteration of the loop has a running time in $O(\deg(u))$. That loop visit each vertex once and has a running time of $\sum_{v \in V} O(deg(v)) = O(n)$. Algorithm 3 visits each vertex again once and relabels it. It also have a running time of $O(n)$.

The running time of the algorithm is $O(n)$. □

4 Concluding Remarks

In the paper we gave a linear time algorithm that finds global offensive and global powerful alliances of any weighted tree $T = (V, E)$. Combining our results with the obvious lower bounds we actually show that the problems of finding global offensive and global powerful alliances of any weighted tree $T = (V, E)$ are $\Theta(|V|)$. The problem of finding the global defensive alliance number of a tree quickly seems to be more difficult. The difficulty is in the fact that not all vertices have to satisfy the same condition (as in the powerful alliance), and more importantly, we cannot claim that adding a vertex to a global defensive alliance will preserve the defensive alliance (in contrast to offensive alliances).

4.0.1 Acknowledgements

We wish to thank Leonid Chindelevitch for indicating Lemma 3.2. We would also like to thank an anonymous referee whose comments improved the presentation of this paper.

References

[1] R. Aharoni, E. C. Milner, K. Prikry, Unfriendly partitions of a graph. J. Combin. Theory Ser. B 50 (1990), no. 1, 1–10.

[2] H. Balakrishnan, A. Cami, N. Deo, and R. D. Dutton, On the complexity of finding optimal global alliances, J. Combinatorial Mathematics and Combinatorial Computing, Volume 58 (2006), 23–31.

[3] R.C. Brigham, R. D. Dutton, T. W. Haynes, S. T. Hedetniemi, Powerful alliances in graphs, Discrete Mathematics Volume 309 (2009), 2140–2147.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein Introduction to Algorithms, McGraw-Hill, 2002.

[5] B.C. Dean, L. Jamieson, Weighted Alliances in Graphs, Congressus Numerantium 187 (2007), 76–82.

[6] A. Harutyunyan, Some bounds on alliances in trees, Cologne Twente Workshop on Graphs and Combinatorial Optimization 2010: 83-86.

[7] A. Harutyunyan, Some bounds on global alliances in trees, Discrete Applied Mathematics 161 (12), 2013, 1739–1746.

[8] A. Harutyunyan, A fast algorithm for powerful alliances in trees, 4th International Conference on Combinatorial Optimization and Applications (COCOA 2010), LNCS 6508 (1) 2010, 31–40.

[9] T. W. Haynes, S. T. Hedetniemi, and M. A. Henning, Global defensive allliances in graphs, Electronic Journal of Combinatorics 10 (2003), no. 1, R47.

[10] T. W. Haynes, S. T. Hedetniemi, and M. A. Henning, A characterization of trees with equal domination and global strong alliance numbers, Utilitas Mathematica, Volume 66 (2004), 105–119.

[11] S. M. Hedetniemi, S. T. Hedetniemi, and P. Kristiansen, Alliances in graphs, Journal of Combinatorial Mathematics and Combinatorial Computing, Volume 48 (2004), 157–177.

[12] E. C. Milner and S. Shelah, Graphs with no unfriendly partitions. A tribute to Paul Erdos, 373–384, Cambridge Univ. Press, Cambridge, 1990

[13] J. A. Rodriguez-Velazquez, J.M. Sigarreta, On the global offensive alliance number of a graph, Discrete Applied Mathematics, Volume 157 (2009), 219–226.

[14] J. A. Rodriguez-Velazquez, J.M. Sigarreta, Spectal study of alliances in graphs, Discussiones Mathematicae Graph Theory 27 (1) (2007), 143–157.

[15] J. A. Rodriquez-Velazquez and J. M. Sigarreta, Offensive alliances in cubic graphs, International Mathematical Forum Volume 1 (2006), no. 36, 1773–1782.

[16] J. A. Rodriguez-Velazquez, J.M. Sigarreta, Global Offensive Alliances in Graphs, Electronic Notes in Discrete Mathematics, Volume 25 (2006), 157–164.

[17] J. A. Rodriguez-Velazquez, J. M. Sigarreta, On defensive alliances and line graphs. Applied Mathematics Letters, Volume 19 (12) (2006), 1345–1350.