Near Linear Time 5/3-Approximation Algorithms for Two-Level Power Assignment Problems

Benjamin Grimmer
Department of Computer Science,
Illinois Institute of Technology,
Chicago, IL
bgrimmer@hawk.iit.edu

Kan Qiao
Department of Computer Science,
Illinois Institute of Technology,
Chicago, IL
kqiao@iit.edu

ABSTRACT

We investigate the problem of assigning power levels to nodes of an ad hoc network to minimize total power while preserving connectivity. We consider a simplified version of this problem by requiring bidirected input graphs (ie if an arc uv exists, then the arc vu exists and has the same cost) and that all arcs have cost 0 or 1. This corresponds to a network where each transmitter can operate at high and low power.

There are two versions of this problem, a symmetric variant which seeks a connected spanning subgraph and includes an edge in the subgraph if both endpoints have power at least the edge cost, and an asymmetric variant which seeks a strongly connected spanning subgraph and includes an arc in the subgraph if the source endpoint has power at least the arc cost. Both of these have been shown to be NP-Complete. We present 5/3-approximation algorithms for each of these that run in $O(m\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Graph Theory**]: Graph Algorithms

Keywords

Approximation Algorithms; Power Assignment; Network Design

1. INTRODUCTION

The problem of assigning power levels to vertices of a graph to achieve a desired property has important uses in modeling radio networks and ad hoc wireless networks. It is common in this type of problem to minimize total power consumed by the system. This class of problems take as input a directed simple graph $G = (V, E)$ and a cost function $c : E \to R_+$. A solution to this problem assigns every vertex a nonnegative power, $p(v)$. We use $H(p)$ to denote the spanning subgraph of G created by this power assignment as described in the following paragraph. The minimization problem then is to find the minimum power assignment, $\sum p(v)$, subject to $H(p)$ satisfying a specific property.

Two variants of this problem will be discussed in this paper. One is the symmetric variant. The power assignment induces a simple undirected graph $H(p)$ on vertex set V given by $\{x, y\} \in E(H(p))$ if and only if the arc $xy \in E$ and $p(x) \geq c(xy)$ and $p(y) \geq c(xy)$. Then the goal is to minimize the *total power* subject to $H(p)$ being connected. The other one is the asymmetric variant. The power assignment induces a simple directed subgraph $H(p)$ on vertex set V where $xy \in E(H(p))$ if and only if the arc $xy \in E$ and $p(x) \geq c(xy)$. The goal is to minimize the *total power* subject to $H(p)$ being strongly connected. Both of these problems have been shown to be NP-Complete in [6, 4].

The first work on Power Assignment was done by Chen and Huang [5], which assumed that E is bidirected (i.e, $uv \in E$ if and only if $vu \in E$, and if these two edges exist, they have the same cost). This problem has had many different approximations proposed, which are compared in [2]. In this paper, we also make this bidirected assumption. Further, we assume that $c : E \to \{A, B\}$ where $0 \leq A \leq B$. This corresponds to a situation where all wireless nodes operate on one of two power levels. In practice, network nodes cannot operate at an arbitrary power level, but rather have fixed operating powers. We consider networks that have each node able to operate at low or high power, transmitting at short or long range. This problem is equivalent to minimizing the number of maximum power nodes, which allows us to assume $c : E \to \{0, 1\}$ without loss of generality.

Previous works on the symmetric and asymmetric power assignment problems have presented 1.5 and $11/7 \approx 1.571$-approximation algorithms respectively [7, 1]. Although these algorithms are polynomial, their runtimes are high degree polynomials which prevents them from being used in many cases. Many previous papers have given faster variations of their approximation algorithms for these problems. We will use $n = |V|$ and $m = |E|$. On the symmetric problem, a 5/3-approximation algorithm has been published with claimed $O(nm\alpha(n))$ runtime [6]. The asymmetric problem has a 7/4-approximation with claimed runtime of $O(n^2)$[4, 3].

We give a fast algorithm for both of these problems. In Section 2, we present our algorithm and then prove its 5/3-approximation bound. In Sections 3 and 4, we show that our algorithm is able to be implemented in $O(m\alpha(n))$ time for the symmetric and asymmetric problems respectively.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FOMC'14, August 11, 2014, Philadelphia, PA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2984-2/14/08 ...$15.00.
http://dx.doi.org/10.1145/2634274.2634276 .

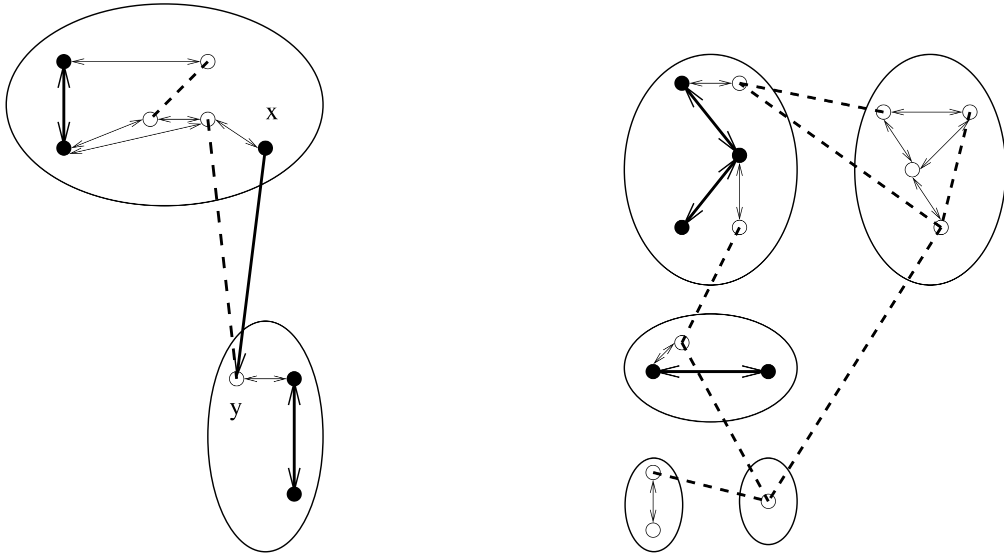

Figure 1: Left: a set S that is not impeccable. Black nodes have power 1 and make up S. All other nodes have power 0. Thin arrows are edges (arcs) of $H(S)$ of cost 0, thicker arrows are edges (arcs) of $H(S)$ of cost 1, and dashed segments are edges of E of cost 1 with no corresponding connection in $H(S)$. Ellipses represent (strong) components of $H(S)$. S is not impeccable as x has power one, y has power zero, and they are in different components. The arc xy is only included in $H(S)$ for the asymmetric problem. Right: an impeccable set.

2. GENERAL ALGORITHM

Our general algorithm will iteratively select groups of vertices that if assigned power one, their separate components will merge into a single component. This approach follows from the concept of perfect sets defined in [3]. In Section 2.1, we formalize the definitions related to perfect sets and then in Section 2.2 we present our algorithm and prove its approximation bound and tightness.

2.1 Definitions

Since we only consider graphs where all arcs have either cost 0 or 1, we can replace the notation of a power assignment with a set of maximum power vertices. Given a set of vertices $S \subseteq V$, we define the function $p^S : V \to \{0, 1\}$ as follows: $p^S(u) = 1$ if $u \in S$ and $p^S(u) = 0$ if $u \in V \setminus S$. We will then abuse notation and write $H(S)$ instead of $H(p^S)$. For the symmetric problem $H(S)$ is an undirected graph and for the asymmetric problem $H(S)$ is a directed graph. The distinction between these two functions will always be obvious from context.

For any $H(S)$, we will use $Comp(v)$ for $v \in V$ to be the connected (strongly connected) component of v. Initially the components of $H(\emptyset)$ are defined by the zero cost arcs. We define the component graph of some $H(S)$ to be a graph $G' = (V', E')$ describing the relationship between all current components as follows. V' is the set of connected (strongly connected) components of $H(S)$. E' has an edge $\{Comp(x), Comp(y)\}$ if and only if $\{x, y\} \in E$ $Comp(x) \neq Comp(y)$. For the purposes of our algorithms, we also associate all the $\{x, y\}$ endpoints in V with each edge added to the component graph. Whenever we refer to a vertex $u' \in V'$ of the component graph, we will use a prime mark to distinguish it from vertices of the original graph.

For any $S \subseteq V$, S is *impeccable* if and only if no $uv \in E$ exists with $p(u) = 1$, $p(v) = 0$ and $Comp(u) \neq Comp(v)$ (See Figure 1 borrowed from [3] for an example). Note that $H(\emptyset)$ is impeccable.

We say that $Q \subseteq V$ is *quasiperfect* with respect to impeccable S if the following holds: For all $u' \in V'$, we have that $|Q \cap u'| \leq 1$, and all vertices of Q are in the same connected (strongly connected) component of $H(Q \cup S)$. Therefore, adding a quasiperfect Q to our current impeccable solution will contract $|Q|$ components into one.

However, adding a quasiperfect set may make our current solution no longer impeccable. If Q is quasiperfect w.r.t. impeccable S and $Q \cup S$ is impeccable, then Q is called *perfect* w.r.t. S (See Figure 2 borrowed from [3] for an example).

2.2 Algorithm and Analysis

Our approach to both symmetric and asymmetric problems will maintain an impeccable set, S, that grows until $H(S)$ is connected (strongly connected). This is accomplished by repeatedly adding sets perfect w.r.t. S into S. Adding such a set, Q, to S will decrease the number of components in $H(S)$ by $|Q| - 1$. The greedy heuristic of adding larger perfect sets to S first follows from this idea. Calinescu's $\approx$1.61-approximation of the asymmetric problem uses this heuristic by adding perfect sets of size k or more, then size $k-1$... then size 4, and finally solving the problem exactly once only perfect sets of size 2 and 3 remain [3].

Following this heuristic, our algorithm for both problem variations, shown in Algorithm 1, first adds perfect sets where $|Q| \geq 4$. Once no sets of size four or more exist, it adds sets of size three, and then sets of size two. For the symmetric case, this approach is essentially the same as the algorithm in [6] and our proof of its approximation bound follows closely from their proofs. We begin with sets of size four or more because we are unable to maintain $O(m\alpha(n))$

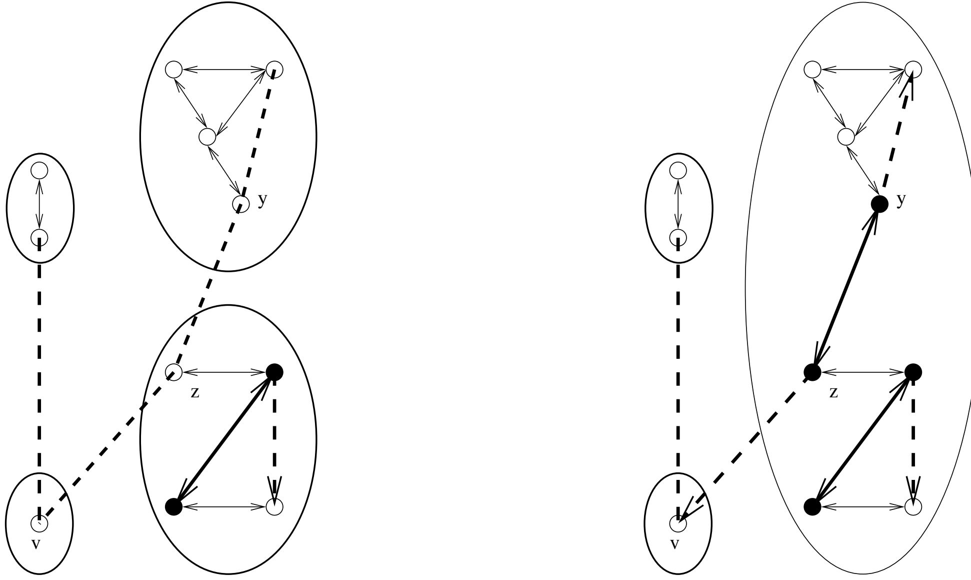

Figure 2: Left: The set $Q = \{y, z\}$ is quasi-perfect w.r.t. S, as explained next. Right: $H(Q \cup S)$. All the vertices of Q are in the same strongly connected component of $H(Q \cup S)$. However, Q is not perfect since z has power one and v has power zero, but they are in different (strong) components of $H(Q \cup S)$. The dashed arc zv and the dashed arc leaving y are only included in $H(S \cup Q)$ for the asymmetric problem.

Algorithm 1 General Approximation Algorithm

1: Set $i = 0$; $S_0 = \emptyset$
2: **while** $\exists$Q perfect w.r.t. S_i and $|Q| \geq 4$ **do**
3: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
4: **end while**
5: **while** $\exists$Q perfect w.r.t. S_i and $|Q| = 3$ **do**
6: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
7: **end while**
8: **while** $\exists$Q perfect w.r.t. S_i and $|Q| = 2$ **do**
9: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
10: **end while**
11: **return** S_i

---

runtime while searching exclusively for larger perfect sets. We denote S at after i sets are added as S_i.

Theorem 1. *Algorithm 1 has a 5/3 approximation ratio.*

Given any graph G, we denote an optimal solution as $OPT(G)$, Algorithm 1's solution as $A(G)$, the number of (strong) components in $H(S)$ before the first loop as K, before the second loop as M, before the third loop as P. We will denote each set of size four or more added as k_i and each set of size three as m_i. It follows immediately that

$$|A(G)| = \sum_i |k_i| + \sum_i |m_i| + 2(P-1) \tag{1}$$

$$\leq \frac{4}{3}\sum_i(|k_i| - 1) + \frac{3}{2}\sum_i(|m_i| - 1) + 2(P-1) \tag{2}$$

$$= \frac{4(K-M)}{3} + \frac{3(M-P)}{2} + 2(P-1) \tag{3}$$

$$= \frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}P - 2 \tag{4}$$

where (2)=(3) holds because the first loop contracts $K - M$ (strong) components by reducing the number of components by $|k_i| - 1$ for each k_i found. Similarly the second loop contracts $M - P$ (strong) components.

Lemma 1. $|OPT(G)| \geq K$

Proof. Any solution has a vertex in each connected (strongly connected) component of $H(\emptyset)$. $\square$

Consider the reduced problem of strongly connecting $H(S_i)$ after the first loop has finished. An optimal solution to the reduced problem will add some perfect sets of size three and some sets of size two. We will denote the number of sets of size two used in the optimal solution as L. Note that $M \geq L$.

Lemma 2. $|OPT(G)| \geq \frac{3}{2}M + \frac{1}{2}L - \frac{3}{2}$

Proof. Strongly connecting $H(\emptyset)$ requires at least as many vertices as strongly connecting $H(S_i)$ in the reduced problem, which requires $\frac{3}{2}(M - L - 1)$ vertices from sets of size three and $2L$ vertices from sets of size two. $\square$

Lemma 3. $\frac{M-P}{2} \geq \frac{M-L}{4}$*, therefore* $P \leq \frac{1}{2}M + \frac{1}{2}L$

Proof. Consider the $\frac{M-L}{2}$ perfect sets of size three added by an optimal solution to the reduced problem. We claim that when Algorithm 1 greedily adds a perfect set, Q, of size three at most two of the optimal perfect sets are no longer perfect. This follows because such a perfect set Q_1 in the optimal solution must have at least two vertices in components contracted by Q. Therefore in $H(S \cup Q_1)$ there will be at most two components with vertices in Q. A second perfect set Q_2 that contracting Q makes imperfect must have vertices in the two remaining components of Q. Therefore, $H(S \cup Q_1 \cup Q_2)$ will leaving Q in a single component, which cannot effect the remaining optimal perfect sets. Therefore the second loop will add at least $\frac{M-L}{4}$ perfect sets of size three before the $\frac{M-L}{2}$ perfect sets of the optimal solution no longer exist. $\square$

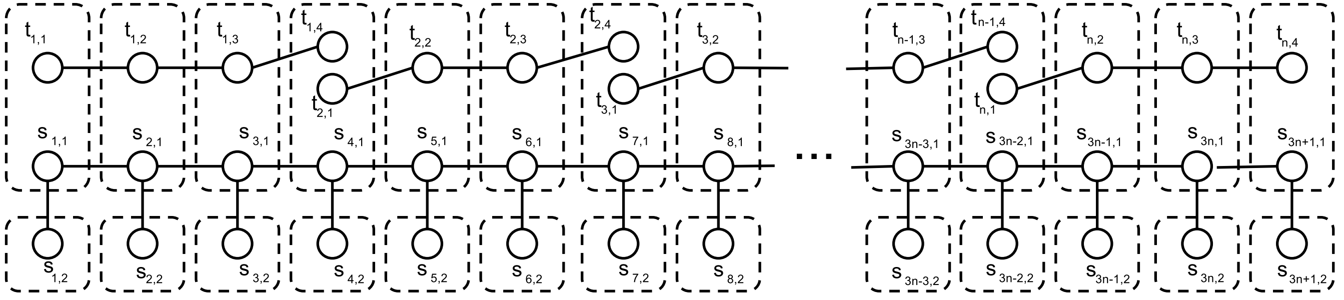

Figure 3: An example for tightness. Dotted boxes represent the (strong) components of $H(\emptyset)$. All edges shown have cost 1.

By taking a convex combination of Lemmas 1 and 2, we find that

$$|OPT(G)| \geq \frac{4}{5}K + \frac{1}{5}(\frac{3}{2}M + \frac{1}{2}L - \frac{3}{2}) \quad (5)$$

$$= \frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{3}{10} \quad (6)$$

From equations (4), (6) and Lemma 3, it follows that

$$\frac{|A(G)|}{|OPT(G)|} \leq \frac{\frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}P - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{3}{10}} \quad (7)$$

$$\leq \frac{\frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}(\frac{1}{2}M + \frac{1}{2}L) - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{3}{10}} \quad (8)$$

$$= \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{3}{10}} \quad (9)$$

$$= \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{5}{20}M + \frac{1}{20}M - \frac{1}{20}L + \frac{3}{20}L - \frac{3}{10}} \quad (10)$$

$$\leq \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{5}{20}M + \frac{3}{20}L - \frac{3}{10}} \leq \frac{5}{3} \quad (11)$$

Therefore Algorithm 1 has a 5/3 approximation ratio. □

Theorem 2. *For both the symmetric and asymmetric problem, the 5/3 approximation ratio of Algorithm 1 is tight.*

Proof. We prove this by giving an class of graphs that Algorithm 1 may assign arbitrarily close to 5/3 times the number of vertices to maximum power as the optimal solution. Consider the example given in Figure 3. This example has $6n+2$ connected (strongly connected) components that could be optimally connected with $6n+2$ maximum power vertices, namely all vertices s_{ij}. Algorithm 1 could choose the set $\{t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}\}$ for each iteration i=1,..,n of the first loop. After this, the only remaining perfect sets are the pairs $\{s_{i,1}, s_{i,2}\}$. Therefore, Algorithm 1 would add each of these pairs in its third loop. This example could have $10n+2$ vertices assigned maximum power. Therefore, we have an approximation ratio of $\frac{10n+2}{6n+2}$, which asymptotically approaches 5/3. □

The tightness example in [6] could also be used to show our approximation is tight.

Theorem 3. *Algorithm 1 has $O(m\alpha(n))$ runtime implementations for both the symmetric and asymmetric problems, where $\alpha(n)$ is the inverse Ackermann function.*

Theorem 3 is our main result. We first show how to implement Algorithm 1 for the symmetric problem in Section 3, and then expand on this method to give an implementation for the asymmetric problem in Section 4.

3. SYMMETRIC IMPLEMENTATION

For this version of the problem, we will consider a simplified version of perfect sets that we will call symmetric sets. A set Q is *symmetric* w.r.t. S if Q is perfect w.r.t. S and the induced subgraph on the vertices of Q in $H(Q \cup S)$ is connected.

Claim 1. *Every perfect set in the symmetric problem is a symmetric set.*

Proof. Consider a perfect set Q in the symmetric problem. Since S is impeccable, each edge added between the components of Q in $H(S)$ must have both endpoints in Q. Using this and the fact that $H(S \cup Q)$ has all components of Q connected, we can conclude that the induced subgraph on the vertices of Q in $H(Q \cup S)$ is also connected. □

Since we only need to consider symmetric sets, we can search for connected subgraphs with each vertex in a different component of $H(S_i)$ with size four or more, then size three, and finally size two.

Any such connected subgraph will have a quasiperfect vertex set. We claim that every quasiperfect set is a subset of some perfect set. Consider the simple construction, called $Augment(Q)$, which takes a quasiperfect Q and will return a set Q' where $Q \subseteq Q'$ and Q' is perfect w.r.t. S.

1: **while** Q is not perfect w.r.t. S **do**
2: $\quad Q = Q \cup \{v\}$ where $u \in Q$, $uv \in E$, and $Comp(u) \neq Comp(v)$ in $H(S \cup Q)$
3: **end while**
4: **return** Q

For both the symmetric and asymmetric problems, we will maintain a Union Find data structure grouping each vertex with all others in the same connected (strongly connected) component of $H(S_i)$. Initializing this data structure only requires computing the connected(strongly connected) components of $H(\emptyset)$, which can be done in $O(m)$. This data structure supports both union and find operations in $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function [8].

When a connected subgraph is found, it will be augmented to make it perfect and then added to our solution. Every vertex processed by Augment will be added to S_i immediately afterwards. Therefore each vertex is inspected at most

once. We can conclude that total time spent augmenting sets is bounded by $O(\sum d(v)\alpha(n)) = O(m\alpha(n))$. For all following implementations we will ignore the cost of the augment function, since this argument can be used to bound its total cost over the entire algorithm.

3.1 Implementation of Loop 1 (lines 2-4)

For the symmetric problem, the first loop needs to find symmetric sets of size four or more. We will process this loop in two phases, first adding sets symmetric w.r.t. S_i that have a vertex with degree three or more in their connected subgraph. Such sets can be easily found by augmenting the set $\{v\}$ for any v with three neighbors in other components (lines 1-5 below).

When no more such vertices exist, all sets symmetric w.r.t. S_i must either have a connected subgraph that is a cycle or a path, because all vertices are adjacent to at most two other components of $H(S_i)$. Therefore all remaining symmetric sets of size four or more contain a path on four vertices in their connected subgraph. We can then find all of these structures in linear time by first precomputing the at most two components adjacent to each vertex, and then checking if each edge can be the middle edge of such a path (lines 6-13 below).

1: **for all** $u \in V$ **do**
2: **if** u adjacent to three or more other components in $H(S)$ **then**
3: $S_{i+1} := S_i \cup Augment(\{\text{u}\})$; $i := i + 1$; Union components of this set
4: **end if**
5: **end for**
6: **for all** $u \in V$ **do**
7: $Adjacent[u] := \{Comp(v) | uv \in E, Comp(v) \neq Comp(u)\}$
8: **end for**
9: **for all** $uv \in E$ **do**
10: **if** $|Adjacent[u] \cup Adjacent[v]| = 4$ **then**
11: $S_{i+1} := S_i \cup Augment(\{u, v\})$; $i := i + 1$; Union components of this set
12: **end if**
13: **end for**

The first two loops will each require iterating through the adjacency list of every vertex doing a find operation on the other end of each edge. Therefore these loops can be implemented in $O(\sum d(v)\alpha(n)) = O(m\alpha(n))$. The final loop inspects each edge and checks if its endpoints are adjacent to two distinct components (four components including the components of the endpoints), taking $O(m\alpha(n))$.

3.2 Implementation of Loop 2 (lines 5-7)

All symmetric sets of size three have a connected subgraph that is either a path or cycle. Both of these can be found in near linear time by searching for the necessary vertices with degree two.

1: **for all** $u \in V$ **do**
2: **if** u adjacent to two other components in $H(S_i)$ **then**
3: $S_{i+1} := S_i \cup Augment(\{u\})$; $i := i + 1$; Union components of this set
4: **end if**
5: **end for**

This loop only needs to read the adjacency list of each vertex, checking the number of neighboring components. So we can implement it in $O(m\alpha(n))$.

3.3 Implementation of Loop 3 (lines 8-10)

After the loop 2 of Algorithm 1 completes, all symmetric sets are either a single vertex or a single edge. So we only need to find edges with both endpoints in different components of $H(S_i)$.

1: **for all** $uv \in E$ **do**
2: **if** $Comp(u) \neq Comp(v)$ **then**
3: $S_{i+1} := S_i \cup \{u, v\}$; $i := i + 1$; Union components of this set
4: **end if**
5: **end for**

This loop's implementation has $O(m\alpha(n))$ runtime. Therefore Algorithm 1 can be implemented in $O(m\alpha(n))$ for the symmetric problem. □

4. ASYMMETRIC IMPLEMENTATION

For the asymmetric version of this problem, we need to consider all perfect sets, not only symmetric sets. We will use the concept of a component cycle as a new structure for finding perfect sets.

We define C to be a *component cycle* if the following holds: $C = \{c_1, c_2, \ldots, c_k\}$ is quasiperfect w.r.t. S, and for all for $1 \leq i < k$ there is an arc $c_i u \in E$ where $Comp(u) = Comp(c_{i+1})$ and there is an arc $c_k u \in E$ where $Comp(u) = Comp(c_1)$. We will then use the following lemma to find perfect sets.

LEMMA 4. *Given Q perfect w.r.t. S, Q has a subset that is a component cycle of size at least two.*

PROOF. Let Q be perfect w.r.t. some S. Let u and v be different vertices of Q. Since Q is perfect, there must be paths from u to v and vice versa in $H(S \cup Q)$. Consider the finite sequence of vertices in Q in each of these paths. If these sequences have no common vertices other than u and v, then combined they make a component cycle. Otherwise, let w be the first vertex of Q in the u,v-path other than u that is common to both paths. Then combining the sequence of vertices in Q from u to w with the vertices from w to u will create a component cycle of size at least two. □

Since any set perfect w.r.t. S_i has a component cycle, we can extend the implementation for the symmetric problem to build perfect sets based on their component cycles. Our approach to the asymmetric problem is given in Algorithm 2. The first four loops of Algorithm 2 (lines 2-13) add perfect sets with size at least four. Then loops five and six (lines 14-19) will handle all perfect sets of size three. The final loop (lines 20-22) will finish connecting $H(S_i)$ by adding the remaining size two sets.

The asymmetric approximation algorithm has considerable overlap with simpler algorithm for the symmetric variant. Fast implementations of loops 1, 5 and 7 of Algorithm 2 have already been shown in Section 3. So we only need to show that Loops 2, 3, 4 and 6 can be implemented under our runtime bound.

These four loops are all based on finding component cycles. Each vertex in a component cycle could have arcs going to a component that is not in the cycle. We will refer to any arc that goes from a quasiperfect set Q to a vertex in a different component of $H(Q \cup S)$ as a *branch*. After the first loop of Algorithm 2 finishes, we can conclude multiple constraints on the structure of all branches, shown in the following claim.

Algorithm 2 5/3-Approximation of Asymmetric Problem

1: Set $i = 0$; $S_0 = \emptyset$
2: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| \geq 4$ **do**
3: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 1
4: **end while**
5: **while** $\exists Q$ perfect w.r.t. S_i and $\exists C \subseteq Q$ component cycle and $|C| \geq 4$ **do**
6: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 2
7: **end while**
8: **while** $\exists Q$ perfect w.r.t. S_i, $|Q| \geq 4$ and $\exists C \subseteq Q$ component cycle and $|C| = 3$ **do**
9: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 3
10: **end while**
11: **while** $\exists Q$ perfect w.r.t. S_i, $|Q| \geq 4$ and $\exists C \subseteq Q$ component cycle and $|C| = 2$ **do**
12: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 4
13: **end while**
14: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| = 3$ **do**
15: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 5
16: **end while**
17: **while** $\exists Q$ perfect w.r.t. S_i and $\exists C \subseteq Q$ component cycle and $|C| = 3$ **do**
18: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 6
19: **end while**
20: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| = 2$ **do**
21: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 7
22: **end while**
23: **return** S_i

CLAIM 2. *After loop 1 of Algorithm 2 finishes, for any Q quasiperfect w.r.t. S_i, $q \in Q$ and branch qu: All other branches qv have $Comp(u) = Comp(v)$, and u has no adjacent components other than ones with a vertex in Q.*

PROOF. If a branch qv existed leading to a different component, then augmenting $\{q\}$ will create a symmetric set with at least four vertices. If u had another adjacent component, then augmenting $\{q, u\}$ will create a symmetric set with at least four vertices. Neither of these can occur after the first loop finishes. □

4.1 Implementation of Loop 2 (lines 5-7)

The second loop of Algorithm 2 will contract perfect sets that contain a component cycle of size four or more until none exist. To implement this, we will use a modified depth first search algorithm on the component graph. When our search finds a cycle of size four or more, the DFS will find a corresponding component cycle and build a perfect set containing that component cycle. Then it will add this set into S, contract the components of the perfect set, and continue processing the DFS.

The DFS will start at an arbitrary vertex of the component graph called $root \in V'$ that is assigned $Depth(root)$=0. When a new component u' is reached from some component v', it will have $Discovered(u')$ change from $false$ to $true$. Further, it will have $Parent(u') = v'$ and $Depth(u') = Depth(v') + 1$. This parent relationship defines a path at any time from the current component being processed to the root. The set of edges each component still has to process will be denoted by $E_{new}(u')$, which initially has value $E'(u')$, namely all edges incident to u' in the component graph.

As the DFS runs, there are three ways it could find a cycle of size four or more (shown in Figure 4 (a)). A Type 1 cycle is found when a back edge extends more than two components up the path. A Type 2 cycle is found when two three cycles are found in the path that share an edge. Note that the DFS may process either of these two back edges first. A Type 3 cycle is found when a single component in the path is in the middle of two three cycles.

As our DFS runs, whenever a three cycle is found, we will mark that each of its components are in a three cycle. The component with the greatest depth has $Bottom3$ set to $true$. The middle component has $Middle3$ set to $true$. Finally the component closest to the root has $Top3$ set to $true$. These values are initially $false$ for every component. We will use $Bottom3$ and $Top3$ to find type 2 cycles and $Middle3$ to find type 3 cycles. However, for any $x' \in V'$, we only want $Top3(Parent(x'))$ to indicate that $Parent(x')$ is the top of a three cycle containing x'. To maintain this, we will reset $Top3(x')$ to $false$ whenever we backtrack to a vertex x'.

By associating the edges that make these three cycles with each of these boolean flags, we can easily detect and construct type 2 and 3 cycles while the DFS runs. While processing a component x', if we find a three cycle and have $Bottom3(Parent(x'))$ or $Top3(Parent(x'))$ we can construct the type 2 cycle shown in Figure 4 (a). Similarly if we find a three cycle and have $Middle3(Parent(x'))$, we can construct the type 3 cycle shown in Figure 4 (a).

Whenever a cycle is found, we will expand the corresponding component cycle into a perfect set. As shown in Figure 4 (b) and (c), our component cycle could have two different types of branches. There could be a branch yz that leads to a component higher in the path. In which case, we can expand our component cycle into a larger quasiperfect set by adding a vertex in each component between $Comp(z)$ and the highest component with a vertex in our current set. Iterating this expanding process must eventually terminate in a quasiperfect set with no branches into the path. This procedure will be called $Expand(Q)$, which takes a quasiperfect Q, and returns a quasiperfect Q' such that $Q \subseteq Q'$ and there are no branches from Q' to components in the path. $Expand(Q)$ is defined as follows (An example is shown in Figure 5):

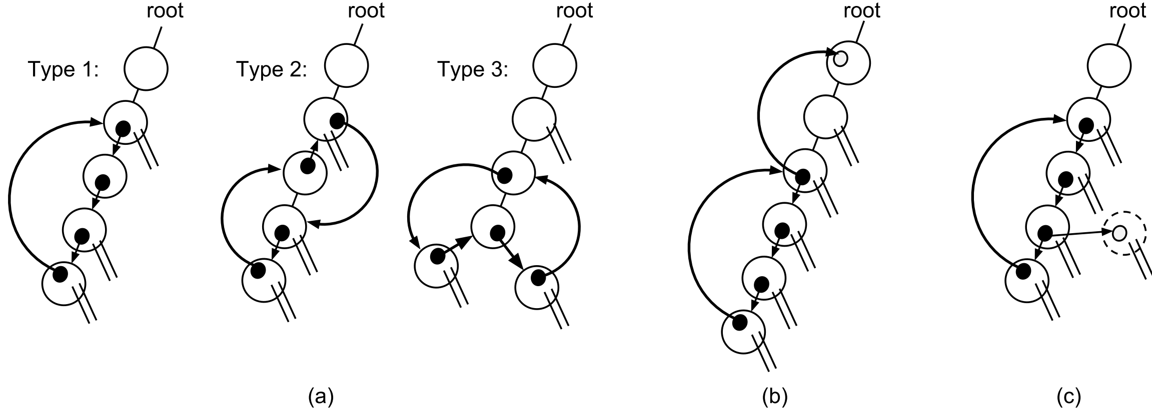

Figure 4: (a) The three types of cycles in the component graph contracted during the DFS. (b) A component cycle with a branch to a component in the path. (c) A component cycle with a branch to an undiscovered component (dashed).

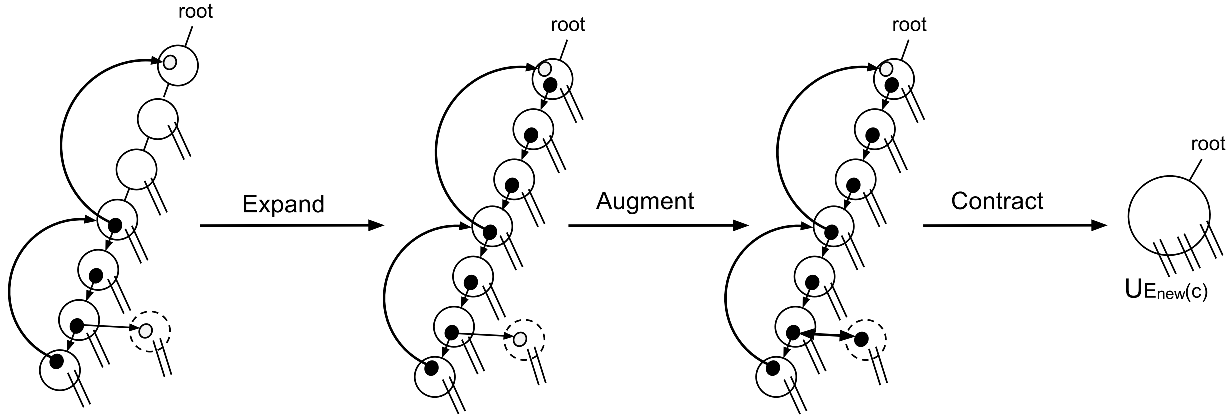

Figure 5: When a component cycle is found, any branch to the path can be removed as shown by the Expand operation. After Expand runs, all remaining tails to undiscovered components are handled by Augment. The final perfect set can be contracted, combining all E_{new}

.

1: $t' :=$ component with a vertex in Q of minimum depth
2: **while** Q has a branch uv and Discovered(v) **do**
3: **while** $t' \neq Comp(v)$ **do**
4: Add a vertex in $Parent(t')$ adjacent to t' into Q
5: $t' := Parent(t')$
6: **end while**
7: **end while**
8: **return** Q

The same type of runtime argument made for $Augment(Q)$ can be applied to $Expand(Q)$. Every vertex needs to be inspected for branches, and finding a branch will add more vertices that need to be inspected. However, since we will add all vertices found by expand into S_i, we will never inspect a vertex twice. Therefore, we can amortize the cost of $Expand(Q)$ to $O(m\alpha(n))$.

The other possible type of branch goes to an undiscovered component. Once these are the only type of arcs leaving our quasiperfect set, we can augment it to get a perfect set. By Claim 2, know that augmenting a branch will only add that component to our quasiperfect set. Therefore we can expand and augment our component cycle to find a perfect set that only contracts vertices at the end of our path and undiscovered components.

When we contract a perfect set Q that merges vertices in the path from the current component x' to the highest component with a vertex of Q, t', we must modify the depth first search state to stay valid. The new component must have all of the remaining E_{new} of components of Q in the path and every edge incident to an undiscovered component of Q. This new component is the bottom or middle of a three cycle that we have already found if and only if t' was the bottom or middle of this cycle before contraction. As with backtracking, we will set the $Top3$ flag of our new component to $false$. Our full implementation of this modified depth first search follows:

1: $Discovered(root) := true$
2: $x' := root$
3: **while** $E_{new}(x') \neq \emptyset$ **or** $x' \neq root$ **do**

4: **if** $E_{new}(x') = \emptyset$ **then**
5: $x' := Parent(x')$
6: $Top3(x') := false$
7: **else**
8: Pick $\{x', u'\} \in E_{new}(x')$
9: Remove $\{x', u'\}$ from $E_{new}(x')$
10: **if not** $Discovered(u')$ **then**
11: $Parent(u') := x'$
12: $Depth(u') := Depth(x') + 1$
13: $Discovered(u') := true$
14: $x' := u'$
15: **else**
16: **if** $Depth(x') - Depth(u') = 1$ **then**
17: Continue to Next Iteration
18: **end if**
19: **if** $Depth(x') - Depth(u') = 2$ **then**
20: **if** $Bottom3(Parent(x'))$, $Middle3(Parent(x'))$ **or** $Top3(Parent(x'))$ **then**
21: Construct a type 2 or 3 component cycle C
22: **else**
23: $Bottom3(x') := true$
24: $Middle3(Parent(x')) := true$
25: $Top3(u') := true$
26: Continue to Next Iteration
27: **end if**
28: **end if**
29: **if** $Depth(x') - Depth(u') \geq 3$ **then**
30: Construct a type 1 component cycle C
31: **end if**
32: $Q := Expand(C)$
33: $Q := Augment(Q)$
34: t' := Minimum depth component of Q
35: $E_{new}(t') := \bigcup_{q \in Q} E_{new}(Comp(q))$
36: Contract the components of Q into t'
37: $S_{i+1} = S_i \cup Q$; $i = i + 1$
38: $x' := t'$
39: $Top3(x') := false$
40: **end if**
41: **end if**
42: **end while**

Claim 3. *After the modified depth first search for loop 2 terminates, no cycle of size four or more exist in the component graph.*

Proof. Assume to the contrary that cycles of size four or more exist in the component graph after the DFS terminates. Then let C be such a cycle in the component graph of minimum size, and let x' be the last component in $V(C)$ discovered by the DFS. Then let u' and v' be the predecessor and successor of x' in C. Since there are no cross edges in a DFS of an undirected graph, we can conclude u' and v' are both above x' in the path. If u' and v' are not the $Parent(x')$ and $Parent(Parent(x'))$, then when one of the edges $\{x', u'\}$ or $\{x', v'\}$ was processed the if-statement on line 29 was entered and a type 1 cycle was found contracting multiple components of C. This contracts our assumption.

We can conclude that u' and v' must been $Parent(x')$ and $Parent(Parent(x'))$. Since they are adjacent in the path, the edge $\{u', v'\}$ exists. Since we chose C to be a minimum cycle of size four or more, C must have size exactly four because a smaller cycle is created by replacing u', x', v' in C with u', v'. We denote the fourth component of this cycle by w'. Then w' is either in the path when x' is being processed or not.

Suppose w' is in the path when x' is processed. Then w' must immediately proceed u' and v' in the path, since w' being any higher would create a type 1 cycle. We consider the two three cycles v', w', u' and x', v', u' in the path. If the cycle v', w', u' is found first, we will mark $Parent(x')$ with $Bottom3 = true$. Then when processing x' we will enter the if-statement on line 20 and contract C. Alternatively, if we find the cycle x', v', u' first, we will mark $Parent(Parent(x'))$ with $Top3$. $Top3$ is only reset when we backtrack, which cannot happen before $Parent(x')$ finishes processing. Therefore, when we are processing $Parent(x)$ and find the cycle v', w', u', we will enter the if-statement on line 20 and contract C. Both of these results contradict our assumption.

Finally, suppose w' was not in the path when x' was processed. Then when w' was processed, u' and v' must have been its two predecessors in the path by the same argument used for x'. The DFS must have finished processing all of w''s edges before discovering x'. So the cycle v', w', u' was found and $Parent(x')$ marked with $Middle3$ before x' was discovered. Then when the cycle x', v', u' was found, the if-statement on line 20 must have been entered and C contracted. This result also contradicts our assumption.

We can conclude that after the depth first search terminates no component cycles of size four or more exist. □

Each iteration of our DFS will consider a new edge in the graph. This bounds the number of iterations at $O(m)$ in the same way that a regular DFS takes $O(m)$ steps. Every operation in the in the loop takes $O(1)$ or $O(\alpha(n))$ except for lines 32-39. However, we have already shown that we can amortize the cost of Expand and Augment over the entire algorithm to get our $O(m\alpha(n))$ runtime. Each of lines 34-39 have an straightforward implementation with runtime $O(|Q|\alpha(n))$, which is bounded by $O(\sum |Q_i|\alpha(n)) = O(n\alpha(n))$ over the entire algorithm. It follows then that our modified DFS maintains near linear runtime, $O(m\alpha(n))$.

4.2 Implementation of Loop 3 (lines 8-10)

The third loop of Algorithm 2 iteratively adds perfect sets of size four or more that have a component cycle of size three. Thus every cycle added by this loop must have at least one branch. We will again use a modified depth first search to find cycles, but our search is made easier since no cycles of size four or more exist after the second loop has finished.

When our new DFS finds a three cycle, we will check all of the endpoints in the original graph of edges between the three components for a branch. If a branch is found, the DFS will expand and augment a component cycle with this branch, contract the resulting perfect set, and then resume the search. We use the same definitions for our DFS as in the second loop of Algorithm 2.

1: $Discovered(root) := true$
2: $x' := root$
3: **while** $E_{new}(x') \neq \emptyset$ **or** $x' \neq root$ **do**
4: **if** $E_{new}(x') = \emptyset$ **then**
5: $x' := Parent(x')$
6: **else**
7: Pick $\{x', u'\} \in E_{new}(x')$
8: Remove $\{x', u'\}$ from $E_{new}(x')$
9: **if not** $Discovered(u')$ **then**

10: $\quad\quad\quad Parent(u') := x'$
11: $\quad\quad\quad Depth(u') := Depth(x') + 1$
12: $\quad\quad\quad Discovered(u') := true$
13: $\quad\quad\quad x' := u'$
14: **else**
15: **if** $Depth(x') - Depth(u') = 2$ **then**
16: Set C to the three cycle using the edge $\{x, u\}$
17: **if** an endpoint in V of the cycle edges is adjacent to component not in C **then**
18: Construct component cycle C' with at least one branch
19: $Q := Expand(C')$
20: $Q := Augment(Q)$
21: t' := Minimum depth component of Q
22: $E_{new}(t') := \bigcup_{q \in Q} E_{new}(Comp(q))$
23: Contract the components of Q into t'
24: $S_{i+1} = S_i \cup Q$; $i = i + 1$
25: $x' := t'$
26: **end if**
27: **end if**
28: **end if**
29: **end if**
30: **end while**

CLAIM 4. *After the modified DFS for loop 3 terminates, no component cycles of size three with at least one branch exist in $H(S_i)$.*

PROOF. Assume to the contrary that there exists a component cycle $\{u, v, w\}$ with a branch ut after this DFS finishes. Let u', v', w' and t' be the components of u, v, w and t, respectively. Then u', v' and w' must follow each other immediately in the path when the last of them is processed, otherwise a cycle of size four or more exists. Since u is an endpoint in V of the edges of this cycle, the DFS will enter the if-statement on line 18 and add some perfect set contracting u', v' and w'. Therefore $\{u, v, w\}$ is not a component cycle after the DFS finishes. Contradiction. □

The runtime of this DFS follows from the same argument made for the implementation of the second loop of Algorithm 2. The only addition work is searching for a branch in the endpoints of our cycle. We can reduce this cost by precomputing the at most two components that every vertex in the original graph can be adjacent to. Then checking a single endpoint for a branch will take $O(\alpha(n))$. Thus checking for a branch in the endpoints between two components that are connected by k edges in the underlying graph will take time $O(k\alpha(n))$. We will never check for such an endpoint between the same two components again, because two distinct three cycles sharing an edge implies the existence of a cycle of size four. Thus we can bound the total time checking endpoints by $O(\sum k_i\alpha(n)) = O(m\alpha(n))$.

4.3 Implementation of Loop 4 (lines 11-13)

The fourth loop of Algorithm 2 will find and contract component cycles of size two that have two branches. A component cycle of size two is just an edge in the component graph. Our implementation will iterate through all the edges in the component graph and check if it has a branch on each end of the edge that go to different components.

As in loop 3 of Algorithm 2, we can reduce the runtime of this implementation by precomputing the at most two components that every vertex in the original graph can be adjacent to. Our implementation follows:

1: **for all** $u'v' \in E'$ **do**
2: **if** $\exists c'_1c'_2$ with c'_1 adjacent to an endpoint of uv in u', c'_2 is adjacent to an endpoint of uv in v' and $\{u', v', c'_1, c'_2\}$ are distinct **then**
3: Construct a component cycle C with two branches
4: $Q := Augment(C)$
5: $S_{i+1} := S_i \cup Q$; $i := i + 1$; Union components of Q
6: **end if**
7: **end for**

By precomputing the adjacency of each vertex, the time spent checking the if-condition for one iteration is at most $O(k\alpha(n))$ where k is the number of edges between the two components in the underlying graph. Then the amortizing argument used for loop 3 also implies this implementation has most $O(m\alpha(n))$ runtime.

4.4 Implementation of Loop 6 (lines 17-19)

This loop is simpler than the three previous loop, since no branches can exist after loop 5 finishes. A simplified version of the third loop can be used to run in $O(m\alpha(n))$ time.

We find that all seven loops and therefore Algorithm 2 run in $O(m\alpha(n))$.□

5. CONCLUSION

We have presented an algorithm for approximating the minimum-cost power assignment problems for network connectivity. Our algorithm gives a 5/3-approximation ratio in near linear, $O(m\alpha(n))$, runtime for both symmetric and asymmetric versions of this problem. This improves upon previous fast approximations, namely, 5/3-approximation in $O(nm\alpha(n))$ and 7/4-approximation in $O(n^2)$, respectively.

6. REFERENCES

[1] A. Karim Abu-Affash, Paz Carmi, and Anat Parush Tzur. Dual power assignment via second Hamiltonian cycle. *CoRR*, abs/1402.5783, 2014.

[2] G. Calinescu and K. Qiao. Asymmetric topology control: Exact solutions and fast approximations. In *INFOCOM, 2012 Proceedings IEEE*, pages 783–791, March 2012.

[3] Gruia Calinescu. 1.61-approximation for min-power strong connectivity with two power levels. *Journal of Combinatorial Optimization*, pages 1–21, 2014.

[4] Paz Carmi and Matthew J. Katz. Power assignment in radio networks with two power levels. *Algorithmica*, 47(2):183–201, 2007.

[5] Wen-Tsuen Chen and Nen-Fu Huang. The strongly connecting problem on multihop packet radio networks. *Communications, IEEE Transactions on*, 37(3):293–295, Mar 1989.

[6] Errol L. Lloyd, Rui Liu, and S. S. Ravi. Approximating the minimum number of maximum power users in ad hoc networks. *Mob. Netw. Appl.*, 11(2):129–142, April 2006.

[7] Zeev Nutov and Ariel Yaroshevitch. Wireless network design via 3-decompositions. *Information Processing Letters*, 109(19):1136 – 1140, 2009.

[8] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.