UNDERSTANDING APPROXIMATIONS FOR NODE COVER AND OTHER SUBSET SELECTION PROBLEMS

Dan Gusfield and Leonard Pitt

YALEU/DCS/TR-308
March,1984

This work was partially funded by the National Science Foundation under grant numbers MCS-8105894, MCS-8204246, MCS-8002447, and MCS-8116678.

Understanding Approximations for Node Cover and Other Subset Selection Problems

Dan Gusfield and Leonard Pitt

Yale University, Department of Computer Science

Abstract

The node cover problem is a well known NP-hard optimization problem on graphs. In the unweighted version of the problem, the optimal can be approximated to within a factor of two by simple, intuitive algorithms, but in the weighted version, the approximation algorithms are more complex and less intuitive. Two recent algorithms for the weighted problem achieve a factor of two approximation. These algorithms are similar, and in both, the crucial parts of the algorithms rely on operations that are counter-intuitive, at first seeming backwards. The published proofs, while elegant, do not really explain the ideas behind these counter-intuitive operations, or identify exportable techniques that might be used in approximations for other hard problems. In this paper we recast and generalize the above algorithms and provide new combinatorial proofs of their accuracy; we interpret the counter-intuitive parts of the heuristics, expose the combinatorial structure permitting such heuristics, identify exportable ideas embodied in these heuristics, and apply these ideas to obtain bounded approximations for subset selection problems generalizing the node cover problem.

1. Introduction

The node cover problem (defined below) is a well known NP-hard optimization problem on graphs. In the unweighted version of the problem, the solution can be approximated to within a factor of two by simple intuitive algorithms [GJ], [S], but in the weighted version, the algorithms achieving a factor of two approximation are more complex and less intuitive. The first factor of two approximation for the weighted case was implicit in a paper of Nemhauser and Trotter [NT] where it was proved that in the linear programming relaxation of the node cover problem, all variables in the optimal basic solution take on values of either zero, one or one-half. They also presented an efficient method for solving the relaxed linear programming problem. We will discuss this method further in section 5. The first explicit factor of two approximation algorithm, due to Hochbaum [H82], was also based on linear programming relaxation, and the graphical aspects of the problem were not explicitly used. Two subsequent algorithms, one by Bar-Yehuda and Even [BE81] and one by Clarkson [C], refined and sped-up the algorithm of Hochbaum, by exploiting the graphical structure of the node cover problem. Both of those algorithms run in linear time, however, the proofs of the approximation bounds in these papers are still based on linear programming relaxation and duality.

The two latter algorithms are similar, and in both, the crucial parts of the algorithm rely on operations that are very counter-intuitive, at first seeming backwards. The proofs of accuracy based on linear programming, while elegant, do not really explain the ideas behind these counter - intuitive operations, or identify exportable techniques that might be used in heuristics for other hard problems, particularly problems that are not closely associated with linear programming. This is especially disappointing since the algorithms appear novel, and do not at first seem to be based on familiar heuristic techniques.

The goal of this paper is to explain the counter-intuitive parts of the above heuristics, giving more familiar interpretations of the algorithms; to expose the combinatorial structure permitting such heuristics; to identify exportable ideas embodied in these algorithms; and to apply these ideas to obtain bounded approximations for problems generalizing those problems treated in [H82] and [BE81]. We do this in three ways. First, we generalize the algorithms in [BE81] and [C], give a direct combinatorial proof (not based on linear programming) of the factor of two bound, and give a more familiar interpretation of the method and of the counter-intuitive parts. Second, we present a factor of two approximation algorithm that contains none of the non-intutitive parts of the algorithms in [BE81] (we will refer to the algorithm in [BE81] as the BE algorithm) or [C]. Although this algorithm is seemingly different than the BE algorithm, we next show that the BE algorithm is, in fact, just this algorithm with a faster implementation. The counter-intuitive parts of the BE algorithm may therefore be viewed as implementation details with no heuristic meaning. Finally, we re-examine the proof of our first algorithm, making explicit the combinatorial structure that allows approximation results based on similar heuristics for other subset selection problems. We illustrate these observations on a resource sharing problem that properly generalizes the set cover and node cover problems treated in [H82] and [BE81]. In an appendix, we present a simple direct proof that the algorithm of [NT] achieves a factor of two approximation of the node cover problem.

Independent of the work presented here, Bar-Yehuda and Even [BE83] discovered a different combinatorial approach and generalization of their algorithm of [BE81], and Monien and Speckenmeyer [MS] developed ideas similar to theirs, but only for the unweighted problem. We will discuss this more in section 2.5.

1.1. Definitions

Let G be an undirected graph with each node i given weight $w(i) > 0$. A set of nodes S is a node cover of G if every edge of G is incident to at least one node of S. The weight of a node cover is the summation of the weights of the nodes in the node cover, and the weighted node cover problem is to select a node cover with minimum weight. In the unweighted version, all nodes have unit weight, and hence a solution is a node cover of minimum cardinality. Even the unweighted problem is NP-hard [GJ], but in addition to the methods that achieve a factor of two for general graphs, there are better approximations for special classes of graphs [H83], [BE82], [C].

Definition: If f is any weight function defined on nodes of G, and S is a set of nodes, then f(S) is defined as $\sum_{i \in S} f(i)$.

2. First Heuristics and Interpretation

In this section we present a node cover heuristic generalizing the weighted and unweighted heuristics in [BE81], [C], [GJ], [S] and give a combinatorial proof that it approximates the optimal within a factor of two. We then give an intuitive interpretation of the heuristic.

Algorithm A

```
0)  Set $t(i) \leftarrow w(i)$ for each node $i \in G$.
    Set $H' \leftarrow \emptyset$.

1)  Pick any node i in G. If there are no nodes in G, then go to 4).
    Otherwise, pick a subset $U_i$ of nodes adjacent to i.

2)  IF $t(i) > t(U_i)$ THEN GO TO step 3)
    (Otherwise, $t(i) \leq t(U_i)$, and we do the following:)
    Set $H' \leftarrow H' \cup \{i\}$.
    For each node j in $U_i$, pick any value d(j) such that $0 \leq d(j) \leq t(j)$,
        and $\sum_{j \in U_i} d(j) = t(i)$.
      Set $t(j) \leftarrow t(j) - d(j)$.
    Remove node i and all incident edges from G, and then remove all isolated
     nodes from G. For any node k removed at this step, set $t'(k) \leftarrow t(k)$.
    GO TO step 1).

3)  IF $t(i) > t(U_i)$, then add all nodes in $U_i$ to $H'$.
    Set $t(i) \leftarrow t(i) - t(U_i)$.
    Remove all nodes in $U_i$ and all edges incident to those nodes,
     and then remove all isolated nodes from G. For any node k removed at this step,
     set $t'(k) \leftarrow t(k)$.
    GO TO step 1).

4)  Set $C \leftarrow H'$. If C is not a minimal node cover
    (i.e. some proper subset of C is a node cover of G) then find a
    minimal node cover H in C, and stop. Output H.
```

The set H' contains the contents of the cover as it is being built during the algorithm, the set C contains all nodes placed in the cover by the end of the algorithm, and the set H is a minimal node cover derived from C.

Note that in steps 2) and 3), the t values of the nodes being placed into H' are not reduced.

It should be clear by the termination condition of step 1, that C is, in fact, a node cover of G.

2.1. Specializations

We note that algorithm A is a generalization of the algorithms in [BE81] and [C]. The algorithm of BE results from restricting U_i to be a singleton. We will discuss this algorithm in more detail in section 3. The algorithm of [C] results from making the following restrictions: Alway pick node i in step 1 to be the node with least ratio $t(i)/v(i)$, where $v(i)$ is the current degree of node i; always take U_i to be all the neighbors of node i; always set $d(j) = t(i)/v(i)$ in step 2. In this version, $t(i)$ is always less than or equal to $t(U_i)$ in step 2, and hence the node i selected in step 1 always enters H', and step 3) never occurs. A simpler version of this is to pick the node of least $t(i)$ value in step 1, and then to subtract a total amount of $t(i)$ off of its neighbors in any arbitrary way. This is also a specialization of algorithm A.

Definition: After the execution of algorithm A, let $K = C \cup \{\text{nodes } i \mid t'(i) = 0\}$. Clearly K is a node cover.

There are two published, simple algorithms explicitly for the unweighted version of the node cover problem. The first, due to Gavril in [GJ], finds a maximal matching M, and then uses both endpoints of every edge in M in the node cover. This method is easily seen as a specialization of algorithm A where the node cover used is K instead of H. It will be shown in theorem 1 below that K also also achieves a factor of two approximation. The second method, due to Savage [S], constructs a depth first search tree D of graph G, and then the node cover consists of all non-leaf nodes in D. This method is a straight-forward specialization of algorithm A.

2.2. What is counter-intuitive about algorithm A?

It is first unclear what the role of the t variables are; why are they used in place of the original weights? More puzzling and counter-intuitive is the fact that when a node enters H', the t values of some subset of its neighbors are reduced, making it heuristically more likely that those neighbors will also enter H′. At first, this seems totally backwards; it would seem more reasonable that as edges incident with a node j become covered by nodes entering H′, j should become less likely, not more likely, to subsequently enter H′. In fact, earlier heuristic algorithms [CH], [J] for the node cover problem do discourage a node from entering the node cover as more of its neighbors do, however, these algorithms can perform very badly, being off the optimal by a ratio of as much as log n, for an n node graph. Should we conclude then that a good and possibly generalizable heuristic is to encourage the neighbors of nodes in H′ to also enter H′? This seems like the wrong conclusion to draw, so the questions are how can we interpret the heuristic and the role of the t variables in a more meaningful way, and what is the combinatorial structure of the node cover problem that makes this counter-intuitive heuristic useful for other problems? We begin to answer these questions by first giving a combinatorial proof of the accuracy of algorithm A.

2.3. Accuracy of algorithm A

Definition: Let I be an optimal node cover of G.

Theorem 1: $w(K) \le 2w(I)$, $w(H) < 2w(I)$, and $w(H - I) \le w(I)$.

Proof: The proof is based on the following scenario. Each node must receive a permit to be in the node cover, and a permit for any node i is issued only after a tax of w(i) dollars is paid. We will interpret the above heuristic in terms of the scenario and show that whenever a node is placed in H′ by the algorithm, a permit for that node has been paid for in the scenario, and that the total tax paid is at most 2w(I) dollars.

The scenario unfolds in parallel with the running of the algorithm. At the start of the scenario each node i in I is given 2w(i) dollars, and no money is given to nodes out of I. Every addition to H′, in the algorithm, is accompanied by some tax payments (detailed below), in the scenario. We make two inductive claims about the algorithm-scenario:

1. At every execution of step 1 of the algorithm, and for any node i still in G, the value of t(i) in the algorithm is equal to the remaining tax that must be paid to obtain a permit for node i, in the scenario.
2. At every execution of step 1 of the algorithm, and for any node i still in G, node i will have at least 2t(i) dollars in the scenario, if i is in I.

These claims are clearly true at the onset of the algorithm-scenario, and we will show that they hold inductively throughout.

We now present the details of the scenario. Suppose node i is picked in step 1 of the algorithm, and $t(i) \leq t(U_i)$. The scenario unfolds in one of two ways depending on whether i is in I or not. If i is in I, then in the scenario, node i pays a tax of t(i) dollars for its own permit, and has at least t(i) dollars left over. It generously uses these t(i) dollars to pay part (perhaps all) of the remaining tax of the nodes in U_i. In particular, for each node j in U_i, node i pays d(j) dollars towards j's permit. This corresponds to step 2 in the algorithm, where t(j) is set to t(j) - d(j) for each node j in U_i, and $\sum_{j \in U_i} d(j) = t(i)$. After node i is removed, every remaining node k in I still has at least 2t(k) dollars, and every remaining node k owes additional taxes of t(k) dollars. Hence in the case that $t(i) \leq t(U_i)$, the inductive claims hold.

If i is not in I, then certainly every neighbor of i, and hence every member of U_i, must be in I. The scenario in this case is that the nodes of U_i generously pay for i's permit, and while at the tax office they pay off part (perhaps all) of their own taxes as well. In particular, each node j in U_i pays d(j) dollars towards node i's permit, and d(j) dollars towards its own permit. By assumption, $t(i) \leq t(U_i)$, and by induction, each node j in I has at least 2t(j) dollars, hence the nodes in U_i have a total of $2t(U_i)$ dollars--enough to pay for i's permit and part of their own. The payment of d(j) dollars towards j's permit corresponds to setting t(j) to t(j) - d(j) in step 2 of the algorithm. Each node j in U_i now owes t(j) - d(j) dollars and has at least 2[t(j) - d(j)] dollars, hence the inductive claims hold in this case also.

Suppose now that step 3 is executed (hence $t(i) > t(U_i)$), and the U_i nodes are placed in H. As before, either i is in I, or all the U_i nodes are. If the U_i nodes are in I, then they pay their own tax of $t(U_i)$ dollars, and apply the remaining $t(U_i)$ dollars to i's tax. If i is in I, then since $t(i) > t(U_i)$, node i pays off the tax of the U_i nodes, and applies $t(U_i)$ dollars to its own tax. In this case, node i now has at least $2[t(i) - t(U_i)]$ dollars and owes $t(i) - t(U_i)$ dollars, and the inductive claims hold.

In the scenario, no node i enters H unless it has a permit, and since the total payments are at most $2w(I)$ dollars, $w(H) \leq 2w(I)$. By the same reasoning, $w(K) \leq 2w(I)$. Further, at least half of all payments are made for nodes in I, since every time payments are made, at least half go for nodes in I. Therefore, the total payment for nodes not in I is bounded by $w(I)$, and, in particular, $w(H - I) \leq w(I)$. Now $H = (H \cap I) \cup (H - I)$, and so the only way that $w(H) = 2w(I)$ is if $w(H \cap I) = w(H - I) = w(I)$. But, H is a minimal node cover and cannot properly contain I, so if $w(H - I) > 0$, then $w(H \cap I) < w(I)$. Hence, $w(H)$ is strictly less than $2w(I)$. □

Definition: The quantity $\sum_{i \in C} [w(i) - t'(i)] + \sum_{i \in C-H} w(i)$ is called the *excess*. That is, the total payments made for nodes not in H.

Corollary 1: $w(H)/w(I) < w(H)/[\sum_{i \in C} t'(i)] = 2w(H)/[w(H) + \text{excess}] \leq 2$.

Proof: Every tax payment is associated with a node entering H′; When node i enters H′, $2t'(i)$ dollars are paid. Over the scenario, part of the total payment goes to buy permits for nodes actually in H, but the rest is excess. Hence the total payment equals $2 \sum_{i \in C} t'(i) = w(H) + \text{excess} < 2w(I)$, and the corollary follows. This corollary can be used for closer empirical estimates of $w(I)$. □

The following corollary will be important in the next sections.

Corollary 2: At any step in the algorithm, $w(H' - I) \leq \sum_{i \in H'} t'(i)$.

Proof: At any step of the algorithm, the total payments made are exactly $2 \sum_{i \in H'} t'(i)$, and at least half of this is applied to nodes in I, so the total amount which is applied to nodes in H′ - I is at most $\sum_{i \in H'} t'(i)$. All nodes in H′ have permits, so $w(H' - I) \leq \sum_{i \in H'} t'(i)$. □

2.4. An Interpretation of Algorithm A: Error and upper bounds

Recall that $w(H) = w(H \cap I) + w(H - I)$. We call any node in H - I an *error*, and w(H - I) the *size* of the error. For a subset S of H, let e(S) denote w(S - I). A reasonable heuristic approach to the node cover problem is to try to limit the size of the error.[1] We know from theorem 1 and corollary 2 that $w(H' - I) \leq \sum_{i \in H'} t'(i) \leq w(I)$ for any H′. So, $\sum_{i \in H'} t'(i)$ is an *upper bound* on e(H′), and at any execution of step 1), and for any node i still in G, t(i) is the *increment* in the upper bound that would result from next adding i to H′. In selecting between node i and the set U_i for entry into H′ the algorithm makes the choice that *minimizes the increment* in the upper bound. (The specialization of the algorithm of [C], mentioned above, in fact picks the node which minimizes the increment of the upper bound over all available nodes.) Viewed in this way, the heuristic goal of limiting the size of the error is embodied in algorithm A as a greedy hill-climing method for limiting the size of the error *bound*.

This partly explains the counter-intuitive nature of algorithm A. Algorithm A should be viewed not as a heuristic trying to find the best node cover, but rather, as an algorithm trying to find a node cover that is *provably* not "too far" from the optimal. The two objectives, finding the best node cover and finding a provably not too bad node cover, are different, and good heuristics for one might not seem intuitive for the other. The t variables and the subtraction operations can be viewed as mechanisms which record potential error and charge the error to

[1]Note, however, that while limiting the size of the error may be a good heuristic, there may be node covers with larger error than e(H), but with lower total weight.

nodes in I, so that no node is charged more than its own weight, and hence maximum potential error is limited to the weight of the optimal node cover.

In this interpretation of algorithm A, the exportable heuristic ideas include the idea of error; the goal of limiting the size of the error; the efficiently computed upper bound on the size of the error; and the hill-climbing use of the increment of the upper bound in the node selection criterion. We call this heuristic method a ***greedy upper bound heuristic.*** In section 4, we will return to this heuristic interpretation to see in general how approximation bounds can be derived by using greedy upper bound heuristics for other subset selection problems.

The above interpretation of algorithm A raises the question of whether better, polynomially computable, error bounds would lead to better (provable or practical) approximations. The key idea behind the error bound in algorithm A is that for every node i, either i is in I, or the neighbors of i are in I. Clearly, there are situations where more precise statements can be made, and the error bounds consequently improved. For example, if $t(i) < t(U_i)$ then algorithm A will choose node i over the set U_i, since it chooses the lower of the two upper bounds on incremental error. However, if some of the nodes of the set U_i are connected, then the true incremental error in choosing node set U_i is strictly less than $t(U_i)$. A greedy upper bound heuristic with more precise estimates of incremental error might then choose set U_i over node i, even though $t(i) < t(U_i)$. In fact, algorithm A and the tax proof can be easily expanded to allow for such tighter bounds. However, It is an open question whether improved error bounds would improve the resulting node covers.

2.5. The Local Ratio Theorem

After the publication of [BE81], and independent of our work, Bar-Yehuda and Even [BE83] discovered a different generalization and combinatorial proof of their algorithm. Monien and Spekenmeyer [MS] developed a similar idea for the unweighted case. The fundamental observation in [BE83] is contained in a theorem called the local ratio theorem, which allows a more "liberal" subtraction of t values than in the original BE algorithm.

Let G' be a subgraph of G with k nodes, such that the minimum *unweighted* node cover of G' has h nodes, and let d be the minimum of any node weight in G'. The local ratio theorem states that if A^* is an algorithm that achieves an approximation of the optimal node cover within a factor of k/h, it will also achieve this ratio when run on graph G, where the weight of *each* node in G' is first reduced by d. Of course the theorem might be applied repeatedly, until G contains a node cover all of whose nodes have zero weight. This generalizes the original BE algorithm, where G' is successively a single edge, and $k/h = 2$.

The tax arguments used in section 2.3 are easily extended to prove the local ratio theorem, and can even generalize it. If G' is a subgraph of G with k nodes, where the optimal weighted node cover has weight w, and A^* is an algorithm that achieves an approximation of the optimal node cover within a factor of c, then A^* will also achieve this ratio when run on graph G, where the weight of *each* node i in G' is first reduced by $\min[t(i), (cw)/k]$. The local ratio theorem is a specialization of this where $c = k/h$, and w is approximated to be at least dh (d and h defined above). In that case, $t(i) \leq d \leq (cw)/k$, and $\min[t(i), d]$ is always equal to d. The idea of the tax proof is that if I is the optimal weighted node cover of G, then $w(I \cap G') \geq w$, and so if each node in $I \cap G'$ has money equal to c times its own weight, then those nodes can pay at least cw dollars for permits of nodes in G'. There are k nodes in G', hence the nodes in $I \cap G'$ can pay to reduce the tax on each node in G' by cw/k, and then every node in $I \cap G'$ will then have money at least equal to c times the tax it still owes. Only nodes with permits are taken into the heuristic node cover, and no node in $I \cap G'$ will ever pay more than c times its own weight.

We have not seen a way to use or generalize the local ratio theorem to prove the factor of two bound for algorithm A, and there are simple examples showing that algorithm A can find node covers within a factor of two of the optimal, that cannot be obtained by using the algorithm implied by the local ratio theorem. Hence the ideas in algorithm A and its proof are different and seem to be more general than those in the local ratio theorem.

3. Deriving the BE Algorithm

Above we gave an interpretation of algorithm A as a greedy upper bound heuristic. This interpretation is "after the fact", and does not completely clarify the role of the t variables. In this section we further clarify the role of the t variables and the subtraction operations in the BE algorithm by reversing our orientation. We start with a seemingly different node cover heuristic containing no unintuitive steps and prove it obtains a factor of two approximation of the optimal node cover. We then show that the algorithm of BE is simply an optimized version of this heuristic, where the t variables and the subtraction operations are used as straight-forward book-keeping mechanisms to speed up the implementation of the heuristic. Thus, in this interpretation of the BE algorithm, the t variables lose their heuristic importance.

The heuristic, A', is based on two observations. The first is that if S is any subgraph of G, then the weight of the optimal node cover of S is a lower bound on the weight of the optimal node cover of G. Therefore a way to achieve a factor of k approximation is to find a subgraph S, such that some subset K of its nodes forms a node cover of G, and such that the optimal node cover of S has weight greater or equal to $w(K)/k$. The second observation is that when S is a forest, the optimal node cover can be efficiently computed, and easily described. The heuristic presented below combines these two observations; it finds a forest S of node disjoint rooted subtrees of G, where the union of the nodes of these trees minus the root nodes forms a node cover of G, and where the weight of the optimal node cover of each tree is at least half the weight of the tree minus the root. Hence the nodes of S minus the roots is a node cover of G with total weight no more than twice that of the optimal.

To begin, let T_v denote a tree, where the subscript v indicates that v is the root node of T_v. For any rooted tree T_v, we denote the immediate descendants of v in T_v by subscripts on v, i.e. by $v_1, v_2, \ldots, v_i$, etc. For any tree T, let d(T) denote the weight of the optimal node cover of T, and let d′(T) denote the minimum weight of all node covers of T that contain the root of T. For a tree T, define w(T) as the sum of the weights of the nodes in T.

Algorithm A′

```
0)  S starts with n rooted trees $T_v$, each a single node v of G.
    For each node v, $d(T_v) = 0$.

1)  Select any two trees $T_x$ and $T_y$ in S, such that there exists
      an edge (x, y) in G between x and y.
    If there are none, then stop.

2)  Join $T_x$ and $T_y$ by adding (x, y), and call the new tree $T_r$.
    To determine the root of $T_r$, first compute the weight of the optimal
      node cover of $T_r$ using the relation:

    (*) $d(T_r) = \mathrm{Min}[w(x) + \sum_{x_i} d(T_{x_i}) + d(T_y),\ \ w(y) + \sum_{y_i} d(T_{y_i}) + d(T_x)]$.

    If the first term is the maximum, then x is made the root r of $T_r$,
      otherwise y is.

3)  Delete $T_x$ and $T_y$ from S, and add $T_r$ to S.
    Go to 1).
```

Comment: The first term of (*) corresponds to the case that node x is definitely in the optimal node cover of T_r, and the second term corresponds to the case that node y is definitely in the optimal node cover. The relation (*) correctly computes $d(T_r)$ since one of the nodes x or y must definitely be in the optimal node cover.

Lemma 1: At any point in the execution of A′, and for any tree T_r in S, the optimal node cover of T_r omits r, i.e. $d(T_r) = \sum_{r_i} d'(T_{r_i})$. Further, each tree T_r in S has the property that $d(T_r) + d'(T_r) = w(T_r)$.

Proof: Both of the claims are clearly true for one and two node trees in S. We assume they hold inductively up to an arbitrary point in the execution of A′, and show they hold after a join operation in step 2 of A′. Suppose, in step 2, that y is made the root of T_r, i.e. the first term of expression (*) is the minimum term. Then the optimal node cover of T_r consists of the optimal node cover of T_y (the tree rooted at node y before the join) plus some nodes in T_x. But, by the induction hypothesis, y is not in the optimal node cover of T_y, hence it is not in the optimal node cover of T_r, and therefore the inductive step for the first claim holds. It follows that $d(T_r) = d'(T_x) + \sum_{y_i} d'(T_{y_i})$, i.e. $d(T_r)$ is the sum of the d′ values of all of the children of r in T_r. Now

$$d'(T_r) = w(r) + d(T_x) + \sum_{y_i} d(T_{y_i}), \quad \text{so}$$

$$d(T_r) + d'(T_r) = [d'(T_x) + d(T_x)] + w(r) + \sum_{y_i} [d(T_{y_i}) + d'(T_{y_i})].$$

Using the induction hypothesis for the second claim, and the fact that $w(r) = w(y)$, this is $w(T_x) + w(y) + \sum_{y_i} d(T_{y_i}) = w(T_x) + w(T_y) = w(T_r)$, and the second inductive claim holds. The case when x is made the root of T_r is the same. □

The following corollary gives a precise charactorization of node covers of weight $d(T_r)$ and $d'(T_r)$.

Corollary 3: For any tree T_r constructed by algorithm A′, the set of all nodes of odd distance from the root of T_r form an optimal node cover of weight $d(T_r)$, and the set of all nodes of even distance from the root of T_r form a node cover of weight $d'(T_r)$. This is of course not generally true for arbitrary weighted trees.

Theorem 2: For each tree T_r in S, $d(T_r) \geq w(T_r - r)/2$.

Proof:

By lemma 1,

$$d(T) + d'(T) = w(T) \text{ for any tree } T \text{ in } S,$$

and since

$$d'(T) \geq d(T), \; d'(T) \geq w(T)/2.$$

Then

$$d(T_r) = \sum_{r_i} d'(T_{r_i}) \geq \sum_{r_i} w(T_{r_i})/2 = w(T_r - r)/2.$$

□.

Let H be the nodes of S minus the roots at the termination of algorithm A′. H contains all of the nodes of G except the root nodes of the trees in S. By the termination condition of A′, there are no edges between roots of these trees, hence H is a node cover of G. Therefore S is a set of trees where the optimal node cover of S is of weight at least half the weight of H, and since the weight of the optimal node cover of S is a lower bound on the weight of the optimal node cover of G, H is within a factor of two of the optimal node cover of G.

3.1. Optimizing algorithm A′

We will show that algorithm BE can be viewed as an optimized version of algorithm A′. As a first step, we modify algorithm A′so that H is built up during the running of A′: when T_x and T_y are joined in step 2, place into H the node, x or y, which does not become the root of T_r. Hence x goes into H at this step if and only if

$$w(x) + \sum_{x_i} d(T_{x_i}) + d(T_y) \leq w(y) + \sum_{y_i} d(T_{y_i}) + d(T_x),$$

if and only if

$$w(x) + \sum_{x_i} d(T_{x_i}) - d(T_x) \leq w(y) + \sum_{y_i} d(T_{y_i}) - d(T_y).$$

But, $w(x) + \sum_{x_i} d(T_{x_i}) = d'(T_x)$, and $w(y) + \sum_{y_i} d(T_{y_i}) = d'(T_y)$, so, at this step of A′, node x goes into H if and only if $d'(T_x) - d(T_x) \leq d'(T_y) - d(T_y)$, otherwise y goes into H.

Now recall that the BE algorithm is algorithm A with the restriction that the neighbor set A in step 1 is a singleton. Then the BE algorithm can more usefully be described as:[2]

```
0)    $H' \leftarrow \emptyset$, and $t(v) \leftarrow w(v)$ for each node v.
LOOP   Until G contains no nodes:
  1)   Pick an edge (x,y) in G;
  2)   Compare t(x) to t(y) and place the node with the smaller t value in H′;
  3)   Subtract the smaller of t(x) and t(y) from the t value of the other node;
  4)   Delete from G the node added to H′, all incident edges
       and all isolated nodes.
```

After each iteration of BE, let S′ be the subgraph of G formed by the edges picked so far in step 1, and consider the set H′ at that point. Clearly, S′ is a forest, and in every tree T of S′, all nodes but one are in H′. For every tree T in S′, if we designate the unique node in T but not in H′ as the root of T, and consider nodes not in S′ to be rooted trees with one node each, then algorithm BE forms rooted subtrees of G in a manner similar to that in algorithm A′. That is, each iteration joins two trees at their roots; every node but the root is in the current H′; and the process continues until there are no edges between any pair of roots. Therefore, if the criterion for putting a node into H′ in algorithm BE is the same as the criterion for putting a node into H in algorithm A′, then the two algorithms are equivalent[3] . We show this in theorem 3 below. We first need the following definition and lemma.

Definition: To distinguish the trees built by algorithm BE from the trees built by algorithm

[2]This is actually the original description given in [BE81].

[3]We could provide a precise definition of equivalence, but rely on theorem 3 instead.

A′, we use $\bar{T}_r$ to designate a tree with root r built by algorithm BE.

Lemma 2: At any point in the BE algorithm, let S′ be the set of trees defined as above. Then for any tree $\bar{T}_r$ in S′, $t(r) = w(r) - \sum_{r_i} t(r_i)$.

Proof: By the way trees are built and roots selected, when each child r_i of r (in $\bar{T}_r$) is placed in H′, the value of $t(r_i)$ at that point is subtracted from node r. All subtractions from w(r) are made in this way, and since the t values of any nodes in H′ remain fixed at the point where they enter H′, the lemma follows.

Theorem 3: At the start of any iteration of BE, and for any tree $\bar{T}_r$ in S′, $t(r) = d'(\bar{T}_r) - d(\bar{T}_r)$. Further, for any tree $\bar{T}$ in S′, the same sequence of joins that created $\bar{T}$ could have been executed by algorithm A′.

Proof: This is clearly true for the first iteration, since t(r) = w(r) for every node r, and $d(\bar{T}_r) = 0$, while $d'(\bar{T}_r) = w(r)$. Suppose it is true up to some iteration of BE, and edge (x,y) is next picked by the BE algorithm. Then t(x) and t(y) are compared, and the node with smallest t value enters H′. In the above interpretation of BE, this means that trees $\bar{T}_x$ and $\bar{T}_y$ of S′ will next be joined by edge (x,y). Call the resulting tree $\bar{T}_r$, where r is either node x or node y. The root of $\bar{T}_r$ is set to x if algorithm BE places node y in H′, otherwise the root is set to node y. Suppose that r is set to node y, and let t(y) be the t value of y before the join, and let t(r) be its t value just after the join. That is, t(r) = t(y) - t(x). Also, recall that r_i designate the children of r in $\bar{T}_r$. By lemma 2, $t(r) = w(r) - \sum_{r_i} t(r_i)$. By the induction hypothesis,

$t(r_i) = d'(\bar{T}_{r_i}) - d(\bar{T}_{r_i})$ for each r_i, so

$$t(r) = w(r) + [\sum_{r_i} d(\bar{T}_{r_i}) - \sum_{r_i} d'(\bar{T}_{r_i})] = [w(r) + \sum_{r_i} d(\bar{T}_{r_i})] - \sum_{r_i} d'(\bar{T}_{r_i}) =$$

$$d'(\bar{T}_r) - \sum_{r_i} d'(\bar{T}_{r_i}).$$

The proof would be finished if $\sum_{r_i} d'(\bar{T}_{r_i}) = d(\bar{T}_r)$. This is the form of lemma 1, but lemma 1 only applies to trees constructed by algorithm A′. However, by the inductive hypothesis, both $\bar{T}_x$ and $\bar{T}_y$ could be constructed by A′, and so could be joined by A′. By the induction hypothesis again, $t(x) = d'(\bar{T}_x) - d(\bar{T}_x)$, $t(y) = d'(\bar{T}_y) - d(\bar{T}_y)$, and hence both algorithms BE and A′choose the same root in joining $\bar{T}_x$ and $\bar{T}_y$, so $\bar{T}_r$ could be constructed by algorithm A′. Now applying lemma 1, $t(r) = d'(\bar{T}_r) - d(\bar{T}_r)$. □

Theorem 3 says that any execution of algorithm BE can be interpreted as an execution of algorithm A′. The converse is proved in the same way.

Theorem 3 yields the interpretation that the t variables and the operations on them in the BE algorithm are simply bookkeeping mechanisms for a faster implementation of algorithm A′; updating t values requires somewhat less arithmetic than updating d and d′ values. In this interpretation, the t variables lose their heuristic importance. However, heuristic interpretations of the t variables are still useful, especially since we have not succeeded in describing algorithm A without using t variables, and, as we show in the next section, the proof of algorithm A suggests additional generalizations of the heuristic.

The papers [BE83] and [MS] also (implicitly) use ideas about trees, but different trees than used here; the manner and detail of their construction and use is very different, and in the weighted case, t variables are also used and subtracted. Readers interested in these additional interesting applications of trees to the node cover problem are referred to the above papers.

4. Other Subset Selection Problems

In this section we isolate the structure needed to generalize and extend the approximation algorithm to other "subset selection" problems.

We've seen an approximation algorithm for the node cover problem, and interpreted it as a greedy upper bound hueristic. What combinatorial structure does the node cover problem have which allows the hueristic to achieve a factor of 2 approximation? The key property is that at each execution of step 1 of algorithm A, we are able to pick 2 subsets of nodes, X_1 and X_2 (one set containing a single node i, and the other consisting of some neighbors of i), such that if I is *any* optimal node cover, either $X_1 \subseteq I$ or $X_2 \subseteq I$. In the scenario-argument, each node i in some arbitrary optimal cover I received 2w(i) dollars, and at all times it had 2t(i) dollars, where t(i) was the "tax remaining to be paid". The fact that either $X_1 \subseteq I$ or $X_2 \subseteq I$ guarantees that there are certain nodes (those in either X_1 or X_2) which can "pay" for the set of nodes entering the cover, and the associated reductions in the values of the t-variables of the other set of nodes.

If S is the set of nodes in a node cover problem, and $X \subseteq S$, then we say that X is *2-partitionable* if we can find sets X_1 and X_2 which partition X, and where at least one is contained in any optimal cover. Note that at each execution of step 1 of the node cover algorithm A, the set G contains some 2-partitionable subset. Hence we say that the node cover problem is *stepwise 2-partitionable.*

We will generalize this property in two ways. We will allow greater than 2 sets to be found, and we will allow them to have non-empty intersection. We will apply this generalization to approximate a more general problem, called the *subset selection problem.*

Definition: Let S be a finite set, and P a polynomially-testable boolean predicate on subsets of S, such that if P(C) is true, then if $C \subseteq D$, then P(D) is true. Assigned to each element s of S is a weight $w(s)$. The subset selection problem (S,P,w) is to select a subset C of S such that $P(C)$ is true, and $\sum_{c \in C} w(c)$ is minimal. Clearly the node cover problem is simply a restricted version of the subset selection problem, hence the subset selection problem is NP-hard.

Given a subset selection problem (S,P,w), a subset $S' \subseteq S$ is said to be *d-coverable* if it is possible (in polynomial time) to form a collection of k non-empty subsets of $S' = \{X_1,X_2,...,X_k\}$, such that $1 \leq k \leq d$, and $\bigcup_{i=1}^{k} X_i = S'$, and if I is any optimal solution of (S,P,w), then for some i, $X_i \subseteq I$.

Whether or not a subset $S' \subseteq S$ is d-coverable will depend on the structure of the predicate P. For example, in the node-cover problem, the predicate P is essentially "for every pair of adjacent nodes, at least one must be chosen". Hence we can easily find many 2-coverable subsets: if $e = \langle u, v \rangle$ is an edge, then $\{u,v\}$ is a 2-coverable subset, since $\{u\}$, and $\{v\}$ are 2 sets of nodes whose union is $\{u,v\}$, and for any optimal cover I, either $\{u\} \subseteq I$, or $\{v\} \subseteq I$.

Many natural subset selection problems have the property that they are "stepwise" d-coverable; *i.e.* that we can successively find a d-coverable subset of S, $S' = \{X_1,X_2,...,X_k\}$; eliminate some X_i from S, and iterate until S is empty. It is this property which allows a greedy upper bound hueristic generalizing algorithm A to obtain a factor of d approximation. Note that there is no restriction on the size of each X_i, or on how it intersects with other X_j in S'. The key point is that there are at most d subsets in S'. Below, we formalize this algorithm, and demonstrate how it may be used for two different problems.

ALGORITHM A"

```
0) $t(s) \leftarrow w(s)$ for each $s \in S$.
   $H' \leftarrow \emptyset$.

1) If $P(H')$ is true, then go to 6).

2) Find k nonempty subsets of S, $X_1, X_2, \ldots, X_k$, $1 \leq k \leq d$,
    all with $t(X_i) > 0$, such that if I is any optimal solution to (S,P,w),
    then for at least one i, $X_i \subseteq I$.

3) Let $X_{min}$ be the set that minimizes $\{t(X_i)\}$.
   For each $X_i \neq X_{min}$, subtract at most $t(X_{min})$
     collectively from the t values of the elements of $X_i$
     as long as no value $t(x)$ for $x \in X_i$ becomes negative.
     Note that since $X_i \cap X_j$ need not be empty, some
     value $t(x)$ may be decremented several times.
   For $x \in X_{min}$, $t(x) \leftarrow 0$.

4) $H' \leftarrow H' \cup \{x \mid t(x) = 0\}$

5) GO TO 1)

6) $H \leftarrow$ an arbitrary minimal subset of $H'$ which
   satisfies the predicate P.
```

We note that in algorithm A", we do not eliminate elements from the set S as they are chosen into H', as we did in algorithm A'. Instead, we simply decrement $t(x)$ to 0, and form H' from those elements. The condition in step 2, that each $t(X_i) > 0$, guarantees that the algorithm never attempts to "cover" subsets which consist entirely of elements which have been previously covered.

Theorem 4. If (S,P,w) is a subset selection problem, and it is possible to execute step 2 of the algorithm A" at every iteration of the loop, then $w(H) \leq d \cdot w(I)$, where I is any minimum solution to (S,P,w).

Proof sketch: The proof that the above algorithm yields a factor of d approximation to an optimal solution is analagous to the node cover approximation proof presented in section 2.3. Let $I \subseteq S$ be any minimum solution of (S,P,w). We have a scenario in which every element $i \in I$ receives $d \cdot w(i)$ dollars, and all other elements of S receive 0 dollars. We show that there is enough money distributed $(d \cdot w(I))$ to pay for all reductions in taxes (t variables). This is easily proved by a tax argument generalizing the proof of theorem 1. A simple argument also shows that the conditions of step 2 are sufficient to guarantee that if the hypothesis of the theorem are true, then the algorithm terminates via step 1, *i.e.* $P(H')$ is true. We omit the details.

We illustrate the usefulness of this algorithm with 2 applications.

1) The Set Cover problem

Let $E = \{e_1,e_2,\ldots,e_k\}$ be a finite set of elements. Let $\mathcal{C}$ be a collection of subsets of E. Associated with each $C \in \mathcal{C}$ is a weight $w(C)$. A ***set cover*** of E is a subcollection of sets $\mathcal{D} \subseteq \mathcal{C}$ such that $\bigcup_{D \in \mathcal{D}} D = E$. The set cover problem is to select a set cover $\mathcal{D}$ with minimum possible weight $\sum_{D \in \mathcal{D}} w(D)$. This problem is NP-hard [GJ]. If we restrict the collection $\mathcal{C}$ so that each element of E is in ***exactly*** 2 elements of $\mathcal{C}$, then we have the node cover problem.

Suppose that each element of E is in at most d subsets of the collection $\mathcal{C}$. A factor of d approximation to this restricted version was first shown in [H82], and later modified in [BE82].

We now note that algorithm A'' will yield a factor of d approximation: Let $S = \mathcal{C} = \{C_1,C_2,\ldots,C_n\}$. Let P be the predicate $P(\mathcal{D})$ is true iff $\bigcup_{D \in \mathcal{D}} D = E$. We observe that each element $e \in E$ ***must*** be "covered" by one of the (at most d) sets C of $\mathcal{C}$ which contain e. Hence step 2 of the general algorithm becomes: Pick an element e not yet covered, let $D_1,D_2,\ldots,D_k$ be all sets of $\mathcal{C}$ containing e $(k \leq d)$. Then $\{D_1,D_2,\ldots,D_k\}$ is d-coverable by $\{D_1\},\{D_2\},\ldots \{D_k\}$. This algorithm is essentially the same as that presented in [BE82].

As a final, and more interesting example, we consider the following problem:

2) The Shared Resource Minimization problem

Let J be a set of jobs to be done, and R a set of resources. Associated with each resource $r \in R$ is a cost $w(r)$. Associated with each job $j \in J$ are (at most) d sets of resources, $R_{j,i} \subseteq R$, $(1 \leq i \leq d)$. In order to complete job j, *all* of the resources from at least one of these at most d sets must be chosen.

For example, if associated with job j are the three sets of resources

$$R_{j,1} = \{r_1, r_2, r_5\}$$
$$R_{j,2} = \{r_1, r_3, r_7\}$$
$$R_{j,3} = \{r_2, r_3\}$$

then the job j could be completed as long as all of the resources from at least one of $\{R_{j,1}, R_{j,2}, R_{j,3}\}$ are chosen. The Shared Resource Minimization problem is to choose a subset $R' \subseteq R$ such that for each job $j \in J$, for some i we have $R_{j,i} \subseteq R'$, and R' is a subset of minimum total weight satisfying this property.

If each job is associated with exactly 2 sets of resources, each containing exactly one resource, then this is the node cover problem. If each job is associated with at most d sets of resources, each containing exactly one resource, then this is the set cover problem (with the d restriction described earlier).

Algorithm A″ may be used to achieve a factor of d approximation for this problem. To cast it as a subset selection problem, we let $S = R$, and let P be the predicate $P(R')$ is true iff for all jobs j, there exists a resource set associated with j, $R_{j,i}$, such that $R_{j,i} \subseteq R'$. Step 2 of the algorithm is executed as follows: pick a job, find its at most d resource sets and let these be the d-cover. Step 3 will guarantee that one of the resource sets is chosen for that job. Continue picking jobs until all have been chosen.

In summary, we began with what appeared to be a counter-intuitive algorithm for the node cover problem. With a more intuitive proof of the algorithm than was previously known, we have isolated several key hueristic techniques: That of miminizing the increment of a particular upper bound on error (greedy upper bound heuristic), and that of using clever book-keeping mechanisms to implicitly carry out these computations. We've noted a combinatorial structure which allows successfull utilization of these heuristics, and concluded with a general factor of d appoximation algorithm for NP-hard subset selection problems which have this structure. We believe there are many more problems for which these techniques are useful.

5. Appendix: A very simple factor of two approximation

All of the approximation algorithms discussed above run in linear time, and none is proved to achieve a factor of less than two. Nemhouser and Trotter [NT] presented a slower algorithm that produces a node cover with very interesting and potentially powerful properties. They did not discuss bounded approximations, but, as pointed out in [BE83] and [H83], the Nemhouser and Trotter theorems immediately imply that their algorithm also achieves a factor of two approximation. We present here a simple direct proof that the Nemhouser Trotter algorithm (actually a simplification of their original algorithm) achieves a factor of two approximation. This is useful for people interested in the approximation result alone, since Nemhouser and Trotter's proofs of their general theorems are difficult[4].

Define an optimal *two cover* of G as a minimum weight *multi-set* M of nodes in G, such that every edge in G is incident with at least two nodes of M. The weight of the optimal two-cover is not more than twice the weight of the optimal node cover of G (since a feasible solution to the two-cover problem is to duplicate a node cover), so it provides a factor of two approximation to the optimal node cover; any node used once or twice in the optimal two-cover solution is used once in the heuristic node cover.

We can solve the two-cover problem optimaly as follows. Given G = (V,E), form the bipartite graph G′ = (V,V,E′) where <a,b> and <b,a> are edges in G′ if and only if (a,b) is an edge in E. Hence G′ contains two copies of every node and every edge of G. Assign weight w(i) to both copies of every node i in G′.

Clearly, a node cover of G′ provides a two-cover of G of equal weight, and we claim that an

[4]Bar-Yehuda and Even, in [BE83], present a proof of most of Nemhouser and Trotter's general theorems that is much simpler than the original proofs.

optimal node cover of G′ provides an optimal two-cover of G. To prove this, it is enough to show that a two-cover of G provides a node cover of G′ of equal weight. Let M be an two-cover of G, let S be the *set* of nodes in M, and let S′ = M - S, the set of nodes in M that appear twice. To form a node cover of G′, choose the nodes in S on the left of G′, and the nodes in S′ on the right of G′. To see this is a node cover of G′, consider an edge (a,b) in G and the two edges <a,b> and <b,a> in G′. At least one of a or b is in S, and if both are, then <a,b> and <b,a> are both covered by S. If only one of a or b is in S, then that node must appear twice in M, and hence once in S′ as well. In this case, <a,b> and <b,a> are covered by S ∪ S′. Clearly, the node cover of G′ defined this way has weight equal to the two cover of G. So the optimal node cover of G′ provides an optimal two-cover in G.

The optimal node cover of a bipartite graph can be found by a straight-forward reduction to a minimum cut problem, as detailed in [FF], and it is well known that the minimum cut in a graph can be found in $O(n^3)$ time [E]. □

6. References

[BE81] Bar-Yehuda, R. and S. Even, "A Linear Time Approximation Algorithm for the Weighted Vertex Cover Problem", J. Algorithms, vol. 2, 1981, 198-203.

[BE82] Bar-Yehuda, R. and S. Even, "On Approximating a Vertex Cover for Planar Graphs", Proc. 14th Ann. ACM Symp. on Theory of Computing, 1982, 303-309.

[BE83] Bar-Yehuda, R. and S. Even, "A Local-Ratio Theorem for Approximating the Weighted Cover Problem", manuscript, 1983.

[CH] Chvatal, V., "A Greedy-Heuristic for the Set-Covering Problem", Math. of Operations Research, vol. 4. no. 3., 1979, 233-235.

[C] Clarkson, K., "A Modification of the Greedy Algorithm for Vertex Cover", Information Processing Letters 16 (1983) 23-25.

[E] Even, S. *Graph Algorithms*, Comp. Sci. Press, 1979.

[FF] Ford and Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[GJ] Garey, M. and D. Johnson, *Computers and Intractability; a Guide to the Theory of NP-Completeness*; Freeman, 1979.

[H82] Hochbaum, D.S., "Approximation Algorithms for the Set Covering and Vertex Cover Problems", Siam J. Computing, vol. 11, 3, 1982, 555-556.

[H83] Hochbaum, D.S., "Efficient Bounds for the Stable Set, Vertex Cover and Set Packing Problems", Dis. App. Math, vol. 6, 1983, 243-254.

[J] Johnson, D., "Approximation Algorithms for Combinatorial Problems", J.C.S.S. 9 (1974) 256-278.

[MS] Monien, B. and E. Speckenmeyer, "Some Further Approximation Algorithms for the Vertex Cover Problem" 8th Coll. on Trees in in Algebra and Programming, Univ. degli. Studi de L'Aquila, Italy, March 1983.

[NT] Nemhauser, G.L. and R.E. Trotter, "Vertex Packing Structural Properties and Agorithms", Math. Programming 8 (1975) 232-248.

[S] Savage, C., "Depth First Search and the Vertex Cover Problem", Information Processing Letters, vol. 14, no. 5, 1982.