EFFICIENT ANYTIME SEARCH ALGORITHM FOR SIMPLIFIED INFLUENCE MAXIMIZATION (VERTEX COVER) PROBLEM IN SOCIAL NETWORKS

Ali Vâlâ Barbaros
avbarbaros@cs.ucla.edu

June 8, 2013

Abstract

In this project, we worked on the simplified influence maximization problem in social networks. We show that it is almost the same as the minimum vertex cover problem which is known to be NP-complete. There are classical approximation algorithms that give near-optimal solutions and four-clique additive pattern database heuristic based search algorithm, which is purposed by Felner et al [1]. Their algorithm is the best-known algorithm that gives optimal solutions for minimum vertex cover problem. In addition to these previous works, we proposed a Depth-First Branch-and-Bound (DFBnB) algorithm with Spanning Tree Heuristic that is a new heuristic evaluation function which is based on computing minimum vertex cover of the spanning tree of the given graph. We also adopted the three pruning techniques of inclusion/ exclusion problem space tree [1]. In addition to the pruning techniques we also used pre-evaluation function, which reduces the given graph before starting the searching of problem space tree. Our experiments show that our DFBnB algorithm with spanning tree heuristic gives better results for running time in social networks that typically have power-law edge degree distribution. However our algorithm couldn't advance the running time of the four-clique additive pattern database heuristic based search algorithm in random graphs.

1. Introduction

In the 1970s, researchers including Mark Granovetter and Thomas Schelling in the mathematical social sciences began trying to develop models of certain kinds of collective human behaviors: Why do particular fads catch on while others die out? Why do particular technological innovations achieve widespread adoption, while others remain focused on a small group of users? What are the dynamics by which rioting and looting behavior sometimes (but only rarely) emerges from a crowd of angry people? They proposed that these are all examples of *cascade processes*, in which an individual's behavior is highly influenced by the behaviors of his or her friends, and so if a few individuals provoke the process, it can spread to more and more people and eventually have a very wide impact. We can think of this process as being like the spread of an illness, or a rumor, jumping from person to person. In recent years, researchers in marketing and data mining have looked at what kind of model could be used to investigate "word-of-mouth" effects in the success of new products (the so-called viral marketing phenomenon). The idea here is that the behavior we are concerned with is the use of a new product; we may be able to convince a few key people in the population to try out this product, and hope to trigger as large a cascade as possible [2]. In a nutshell, given a social network graph, a specific influence cascade model (linear threshold model in our project) and a small number k, the original *influence maximization problem* is to find k vertices in the graph (referred to as *seeds*) such that under the

influence of a cascade model, the expected number of vertices influenced by the k seeds is the largest possible [3]. The previous works about this problem are in [6,7,8,9]. Unfortunately all of them are approximation algorithms that don't guarantee optimal solutions. The formalization of the original version of influence maximization problem as a graph is shown at Figure-1 below:

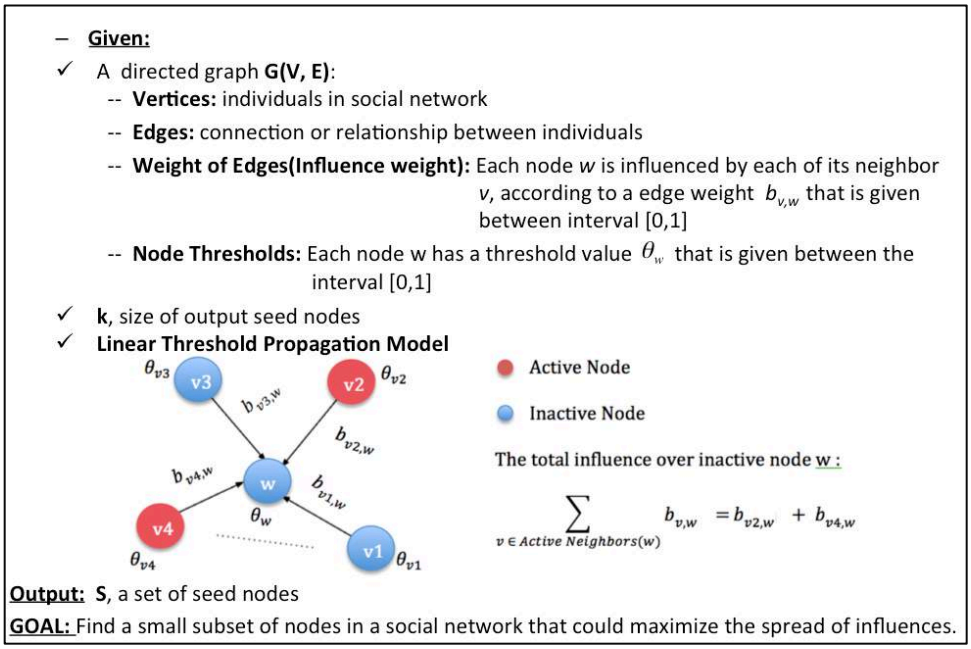


Figure-1 Original Problem Statement

However, we simplified the original problem without violating its NP-completeness, because of its high level of complexity. After simplifying original problem we wind up with well known another NP-complete problem, which is minimum vertex cover problem. The formalization of the simplified version of influence maximization problem as a graph is shown at Figure-2 below:

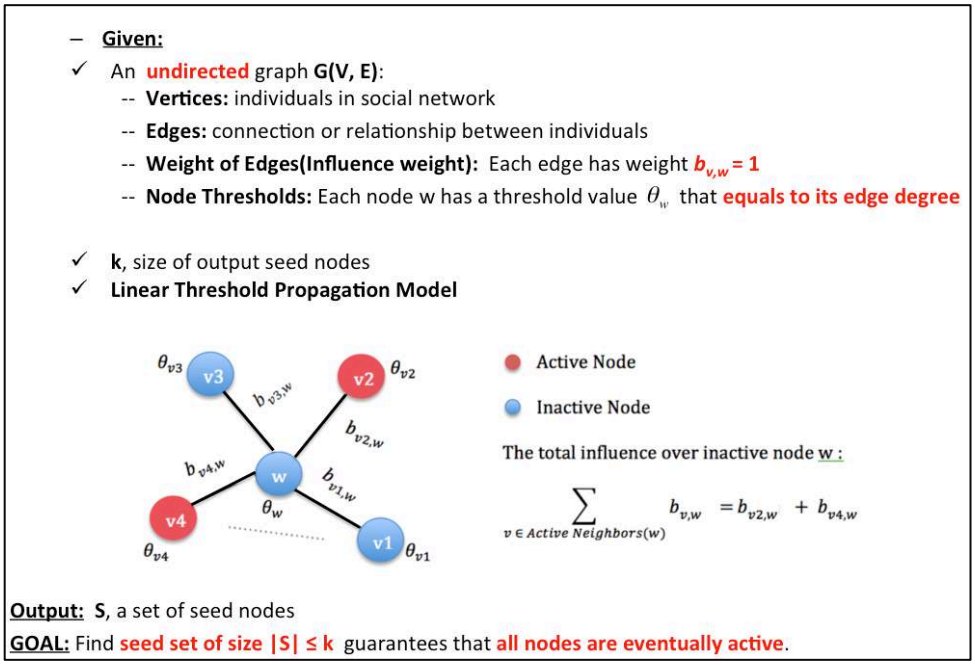


Figure-2 Simplified Problem Statement

In simplified influence maximization problem, we have to cover all the adjacent edges of each node to activate all nodes. This leads us to the minimum vertex cover problem. The vertex cover problem is finding a subset of vertices *C* such that every edge has one of its end points in *C* [1]. Minimum vertex cover is finding the minimum cardinality vertex cover. Therefore simplified influence maximization problem is almost the minimum vertex cover problem except the given parameter *k* that denotes the size of output seed nodes, which equals to number of elements in the set of vertex cover. That means we have an upper bound in our simplified influence maximization problem for seed set. We need to find optimal solutions which include nodes whose count less than or equal to *k*. But for the simplicity, we assume that *k* is equals the number of vertices in the given graph in our experiments. Hence we have two perspectives for this problem: simplified influence maximization and minimum vertex cover. We will use these two perspectives while we are solving the problem for increasing our comprehension about problem and its solution.

1.1 A quick overview of existing algorithms for Minimum Vertex Covering

Many solutions have been purposed for the vertex cover problem. As it is NP-complete, there is no polynomial algorithm constructing an optimal vertex cover. Most of the methods are approximation algorithms and heuristics. In this subsection, we give a rapid overview of existing categories of algorithms.

A well-known heuristic is to select a vertex of maximum degree and delete this vertex and its incident edges from the graph while the degrees of vertices are uploaded, until all edges have been removed. This heuristic called MAXIMUM DEGREE GREEDY [4]. However it doesn't give optimal solutions. It's an approximation algorithm. Another popular solution is EDGE DELETION. Gavril has purposed it in 1979 [5]. It constructs a maximal matching of the input graph and returns the vertices of the matching. It has an approximation ratio at most 2. It is the best-known constant approximation ratio. Inspired by MAXIMUM DEGREE GREEDY, the algorithm GREEDY INDEPENDENT COVER processes as follows: choose a vertex of minimum degree, select its neighbors and delete all these vertices with their incident edges from the graph until all edges have been removed. This also doesn't give optimal solutions and it has an approximation ratio lower than EDGE DELETION.

Besides from approximation algorithms, only algorithm that we can find that gives optimal solutions is depth-first branch and bound algorithm with four-clique additive pattern database heuristic that is purposed by Felner et al [1]. This algorithm utilizes the pattern databases which are precomputed tables of the exact cost of the solving various sub problems of an existing problem. Furthermore unlike the standard database heuristics, they partition their problems into disjoint sub problems, so that the costs of solving the different sub problems can be added together without overestimating the cost of solving the original problem. Pattern database heuristic is the best-known heuristic for the vertex cover problem.

2. Detailed Problem Space

One of the simplest and most effective search space for this problem is an inclusion/exclusion binary tree. This problem space was also used by Felner et al [1]. Each node of the tree corresponds to a different vertex of the graph. The left branch includes the corresponding vertex in the vertex cover, and the right branch excludes the vertex from the vertex cover. The complete tree will have depth *n*, where *n* is the number of vertices, and 2^n leaf nodes, each corresponding to a different subset of the vertices. The inclusion/exclusion binary search tree for sample graph that has five nodes is depicted in Figure-3 below.

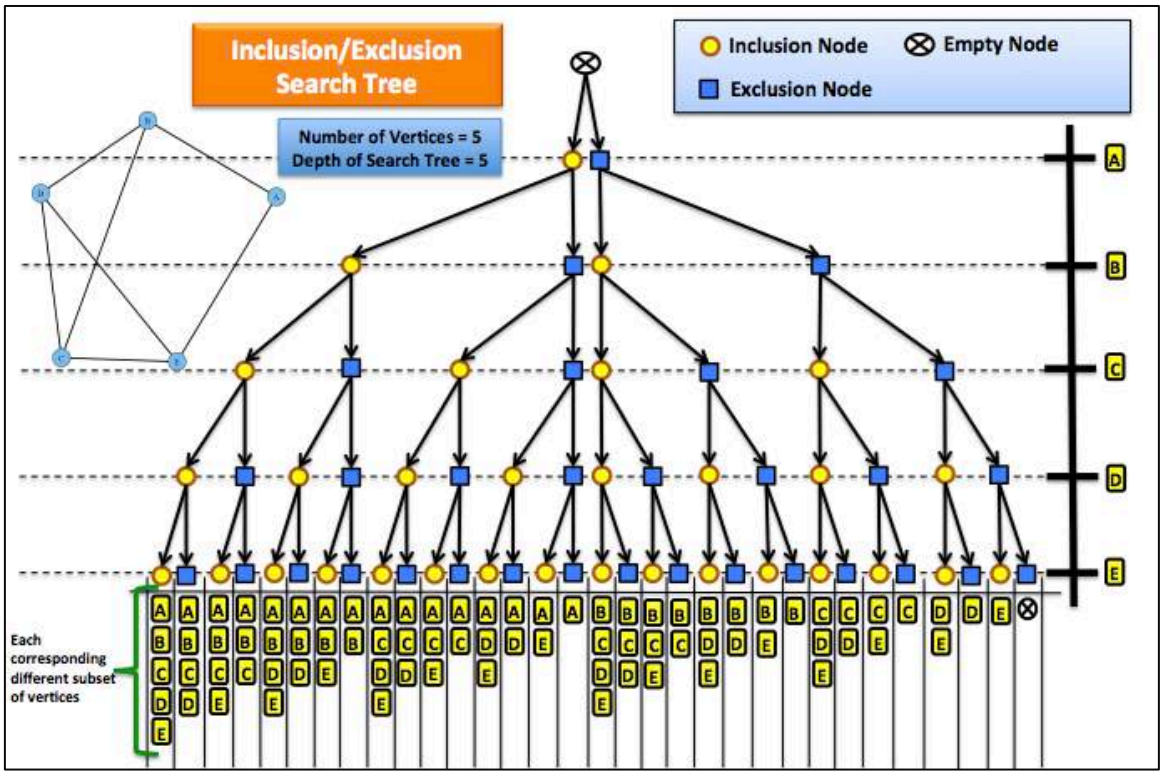


Figure-3 Inclusion/Exclusion Binary Search Tree for sample graph with five nodes

2.1 Pre-evaluation of Problem Graph

Before starting the search in our inclusion/exclusion search tree, in some situations we can decide that some of the nodes have to be in vertex cover set and some of them don't. In a given problem graph, if a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set. In a nutshell, we can exclude the leaf nodes of given graph from vertex cover set and include its neighbors. Then we can continue this iterative process until no nodes which have one neighbor remains. Furthermore we can remove these nodes and their edges from our problem graph and we can start our search with remaining graph.

In Figure-4, we can put nodes 9 and 19 into our vertex cover set directly then we can remove nodes 9,19,19,20 and their incident edges from the problem graph and start our search in remaining graph.

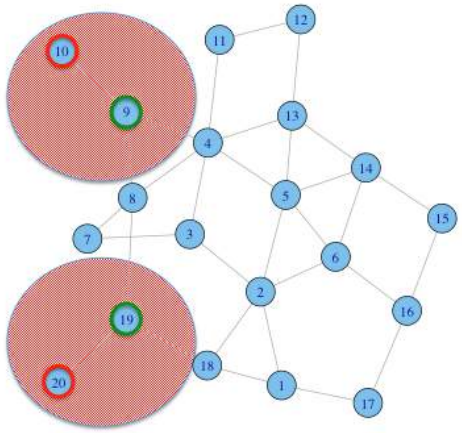


Figure-4 Pre-evaluation of problem graph

3. Pruning of Search Tree

We used three types of pruning techniques for the inclusion/exclusion problem space tree. These techniques are also used in [1].

3.1 Inclusion Pruning

If a node in problem graph is included in the vertex cover set during the course of the search of problem space tree, all of its incident edges are covered. Therefore we don't need to include nodes whose incident edges covered by other included nodes in vertex cover set. Inclusion pruning is depicted in Figure-5 for sample graph that has nine nodes. In this figure, if we include node A in our vertex cover set, we don't need to include node C, so we can prune the inclusion branch of C in the search tree. The same situation is also valid for D and E: if we include D in our vertex cover set we don't need to include node E, so we can prune the inclusion branch of E in the search tree. Furthermore, if we include nodes A and B in our vertex cover set, we don't need to include F, therefore we can prune the inclusion branch of F where nodes A and B included.

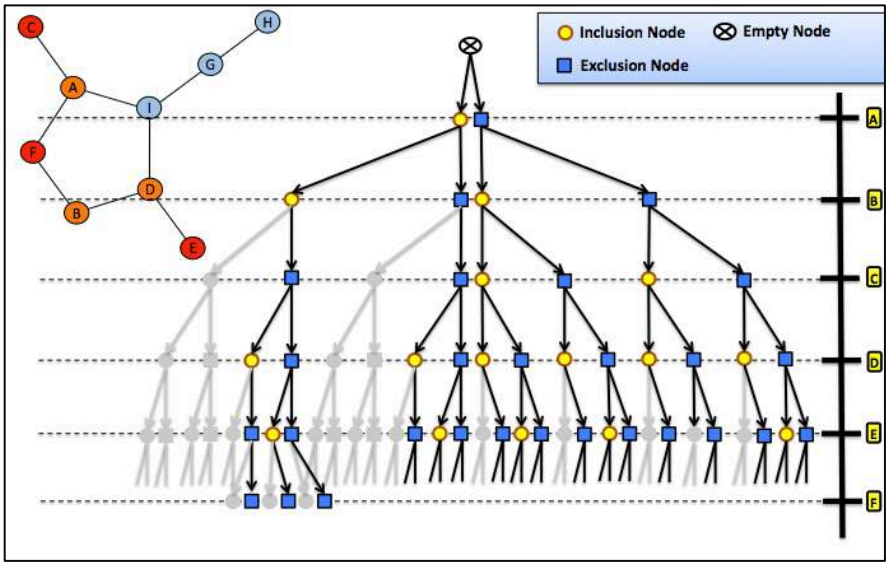


Figure-5 Inclusion Pruning of Problem Space Search Tree

3.2 Exclusion Pruning

If a node in problem graph is excluded from the vertex cover set during the course of search of the problem space tree, all the incident nodes have to be included for covering the edges between them. In these situations, branches, which don't include these incident vertices, are pruned. Exclusion pruning is depicted in Figure-6 for sample graph that has nine nodes. In this figure, if we exclude node A in our vertex cover set, we have to include nodes C, F and I into our vertex cover set for covering the edges between them. Therefore we can prune the exclusion branches of nodes C, F and I. If we exclude node B in our vertex cover set then we have to include nodes F and D so we can prune these nodes' exclusion branches. Lastly if we exclude node B, we have to include E in our vertex cover set, so we can prune the exclusion branches of node E in our problem space tree.

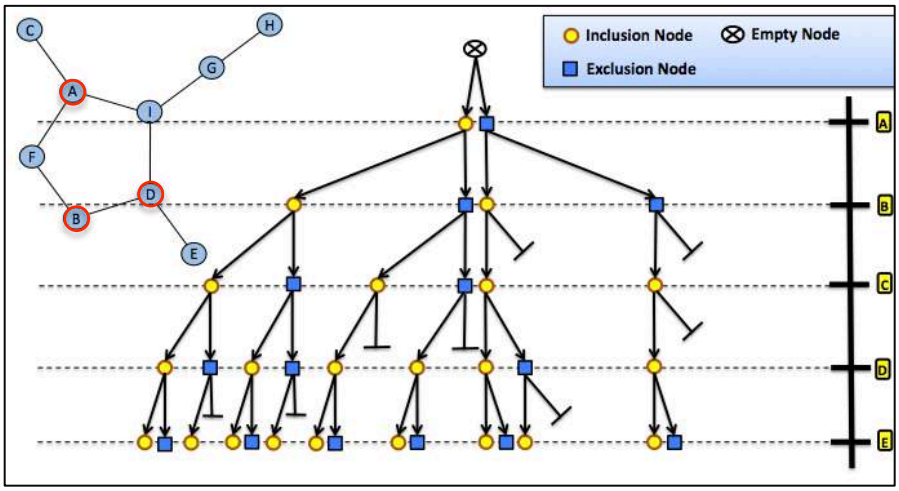


Figure-6 Exclusion Pruning of Problem Space Search Tree

3.3 k – Bound Pruning

If the number of included nodes in the vertex cover set exceeds the given parameter *k*, all the other inclusion node branches will be pruned since *k* is the upper bound of the number of nodes in the vertex cover set or simplified influence maximization seed set. k-bound pruning is depicted in Figure-7 for sample graph that has nine nodes and k equals to 4. In the left most branch, when number of elements exceeds the k, we have to prune next inclusion branches under this set.

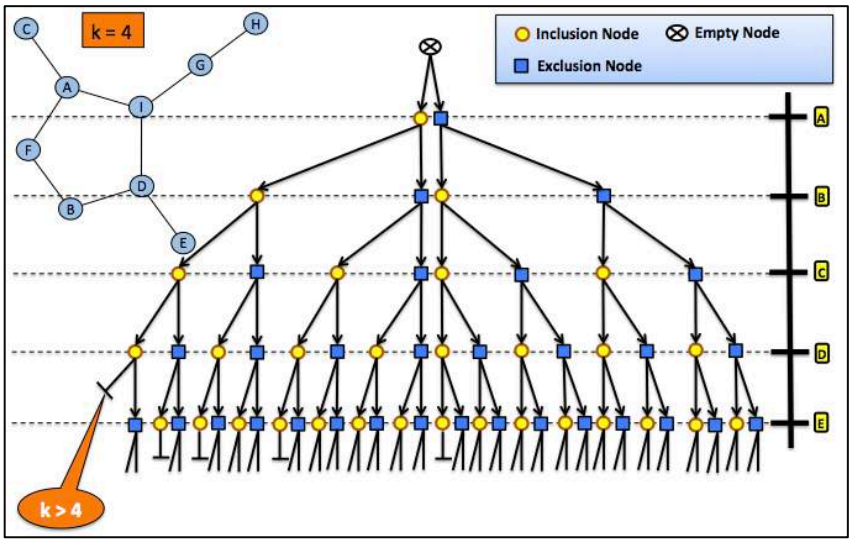


Figure-7 k-Bound Pruning of Problem Space Search Tree

3.4 Node Ordering Heuristic for Improving Pruning Performance

If we look at our problem from the perspective of simplified influence maximization problem, we can easily realize the ways of improving the pruning performance of our search algorithm. For instance, in sociology literature, degree and other centrality-based heuristics are commonly used to estimate the influence of nodes in social networks. Degree is frequently used for selecting seeds in influence maximization. Experimental results show that selecting vertices with maximum degrees as seed nodes, results in larger influence spread than other heuristics.

Ordering the nodes of the graph from maximum edge degree to minimum edge degree in inclusion/exclusion problem space search tree is depicted at Figure-8 below. Our experimental results also show that pruning efficiency is improved greatly by node ordering heuristic.

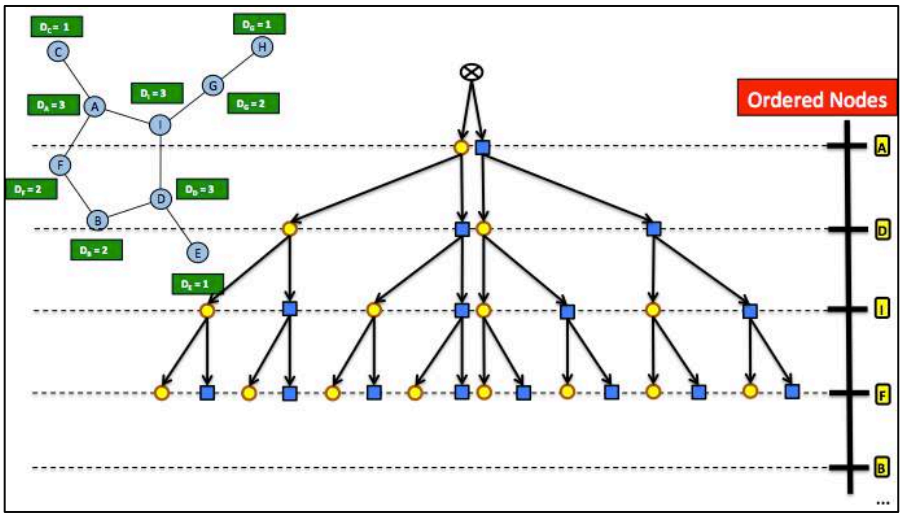


Figure-8 Node Ordering of Problem Space Search Tree with Decreasing Order of Degree

4. Selection of Search Algorithm

We will use the Depth-First Branch-and-Bound (DFBnB) for searching our inclusion/exclusion problem space tree. The inclusion/exclusion scheme guarantees that there is a unique path to each vertex cover; therefore there are no duplicate nodes, and this situation eliminates the need for storing previously generated nodes and allows a linear-space depth-first search. When we reach the first complete vertex cover, we save the number of vertices it includes. At any node *n* of the tree, let *g(n)* represent the number of vertices that have been included so far at node *n*. This is the cost of the partial vertex cover computed so far, and can never decrease as we go down the search tree. Hence, if we encounter a node *n* for which *g(n)* is greater than or equal to the smallest complete vertex cover found so far, we can prune that node from further search, since it cannot lead a better solution. If we find a complete vertex cover with fewer nodes than the best so far, we update the best vertex cover found so far. However, we used A* cost function in our algorithm, therefore we will decide cost of the partial vertex cover with sum of g(n) and h(n) which is the value of the heuristic evaluation function on node *n*.

4.1 Cost Function

During the course of the search, in our DFBnB algorithm we apply the A* cost function f(n) at each interior node *n*.

$$\mathbf{f(n) = g(n) + h(n)}$$

f(n) = Denotes the cost value of the node n.
g(n) = Denotes the number of nodes that have been included so far at node n of the search tree. This is the monotonic function that can never decrease as we go deep the search tree.
h(n) = Denotes the estimation of the number of additional vertices that must be included to cover the edges of the remaining graph.

In our DFBnB algorithm, during the coarse of the search we apply the cost function f(n) at each interior node n in the search tree. If this cost equals or exceeds the size of the smallest vertex cover found so far, we can prune this node. We are still guaranteed an optimal solution as long as h(n) never overestimates the number of additional vertices we have to add to the current vertex cover.

5. Design of Heuristic Evaluation Function

Our main contribution to this problem is a newly designed heuristic evaluation to estimate the number of additional vertices that must be included to cover the edges of the remaining graph. We designed our heuristic evaluation function in classical way: by producing the functions that returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem [12]. We can relax our original problem by removing the some of the constraints since the constraints increase the cost of the solution. The constraints in minimum vertex cover problem are minimum cardinality of vertex cover nodes and covering the all edges. If we think about these constraints, we realize that we can relax the "covering all the edges " constraint. This relaxation in current problem leads us to two obvious questions: Can we solve the exact cost of the vertex cover problem optimally in a graph, which has fewer edges than original graph? And how can we produce this kind of graphs, which has fewer edges than original graphs? The possible answer for the first question is that it is possible to solve the optimal cost of vertex cover problem if underlying graph is a tree. The naive answer of the second question is that we can find the spanning tree of the graph in polynomial time by simply applying Depth-first search algorithm. Essentially the whichever algorithm we apply in our graph, the edge and node count of the spanning tree will never be changed since the spanning tree must have the edge count exactly (n+1) where n is the number of vertices in the graph. We designed a polynomial time greedy algorithm for solving the minimum vertex cover problem in tree, which gives optimal solutions. We will explain this algorithm and prove its optimality in the next section.

5.1 Spanning Tree Heuristic Evaluation Function

The pseudo code of the algorithm that solves the minimum vertex cover problem optimally when the underlying graph is a tree is shown in the Figure-9. This algorithm has O(n) running time complexity.

Our heuristic evaluation function h(n), firstly produce the spanning tree(depth-first spanning tree) of the remaining graph. Then apply our polynomial time algorithm to spanning tree and eventually returns the estimation of the number of additional vertices that must be included to cover the edges of the remaining graph.

```
Algorithm(input tree G=(V,E)) :
1.  Choose one of the nodes which has the biggest d(v) as a root
2.  Let L , denotes the list of leaves of G=(V,E)
3.  for each node v∈ V
4.      mark[v] = FALSE
5.  endfor
6.  While L is not empty
7.      f = remove the first leaf from L
8.      If f is in G=(V,E)
9.          If mark[f] == FALSE and parent[f] == NULL
10.             mark[f] = TRUE
11.         else if  mark[f] == FALSE and parent[f] != NULL
12.             mark[ parent[f] ] = TRUE
13.         endif
14.         remove f from G=(V,E)
15.         If parent[f] != NULL and  parent[f] != root
16.             If children[ parent[f] ] == NULL
17.                 append parent[f] to L
18.             endif
19.         endif
20.     endif
21. endwhile
22. Return min vertex cover set{ v∈V | mark[v]=TRUE }
```

Figure-9 Pseudocode for algorithm that solves minimum vertex cover in trees

We can also prove our algorithm's optimality:

Theorem: Our algorithm outputs a minimum-sized vertex cover of input tree T=(V, E). A minimum-sized vertex cover of T is a subset of vertices V' of V such that:

1) V' is a cover of the edges of T: for each edge (u,v) in E, either u is in V' or v is in V'(or both):
2) The size of V' should be minimal: There should not exist any vertex subset V'' of V' such that V'' is a vertex cover of T and |V''| < |V'|.

Proof: To establish (1) and (2) above, we need facts (a)-(c) below.

a) Never need to include original leaf of T in vertex cover unless leaf is root. This is because (unless leaf is root) degree of leaf < degree of its parent.
b) Must cover edge from leaf to parent of leaf. Since T is a tree, only one edge comes from a parent. Thus, we must include parent of each leaf in cover if we choose to not include leaf in cover.
c) If parent x of a node is in cover, parent x also covers edges to all children of x and the edge to parent[x].

Due to (a)-(c) above, adding the parent of each unmarked leaf is always better than adding leaves to the cover (note that all leaves are unmarked at the start). We therefore make that choice for each unmarked leaf that has a parent (i.e. we mark its parent as belonging to the vertex cover). After making that choice for all leaves, the edges from leaves to parents are covered.

Since adding to the cover the parent of each unmarked leaf is always better than adding leaves to the cover, and all edges from leaves to parents must be covered, a minimal cover of T is the union of the set of parents of leaves with the set of nodes that form a minimal cover of the modified T consisting of T without the leaves and their edges.

Once we make the choice for each leaf, the leaves are therefore no longer relevant. Therefore, we remove leaves and all incident edges from the tree and from the leaf list. Next, append to the leaf list all nodes that become new leaves in the modified tree. We now apply the same process iteratively to this modified tree as to the original. However, some leaves may already be marked. A marked leaf already covers the edge to its parent, so we need not mark its parent; we simply remove a marked leaf from T and L and check if this creates a new leaf. (Note that this does not increase the size of the vertex cover.)

The choice made at a leaf requires no consideration of sub problems and it leads to an optimal solution. The algorithm therefore has the greedy choice property. (2) is satisfied by establishing that the problem has optimal substructure and the algorithm has the greedy choice property. (1) is satisfied because each edge is covered before it is removed from the tree. The pseudo code is correct because it is consistent with the above description.

Since the while loop in our algorithm executes |V| times and each iteration requires O(1) time, total while loop time is O(|V|). Therefore we can claim that running time complexity of our algorithm is O(n) where n is the number of nodes in the tree(n=|V|).

6. Implementation Setup and Empirical Results

Most of the communication and social networks have power-law link distributions, containing a few nodes that have a very high degree and many with low degree [10,11]. Therefore we also produce random graphs with power-law edge distribution besides other random graphs. We conduct several experiments on different graphs that have different node count, edge count and edge degree to evaluate the performance of our DFBnB with Spanning Tree Heuristic algorithm and compare it with the state of the art DFBnB with additive pattern database heuristic algorithm.

The code is written in Python2.7 using the "igraph" library, and all the experiments are run on a Mac OS X 10.7.5 machine with 2.5Ghz Intel Core i5 CPU and 4GB memory. The code is available in Appendix-A. It includes the implementation of our algorithm and producing the random graphs. We created our graphs with "igraph graph generators". These generators use approaches in [13][14]. Our test graphs and their edge degree distributions are depicted in Figure-10 below.

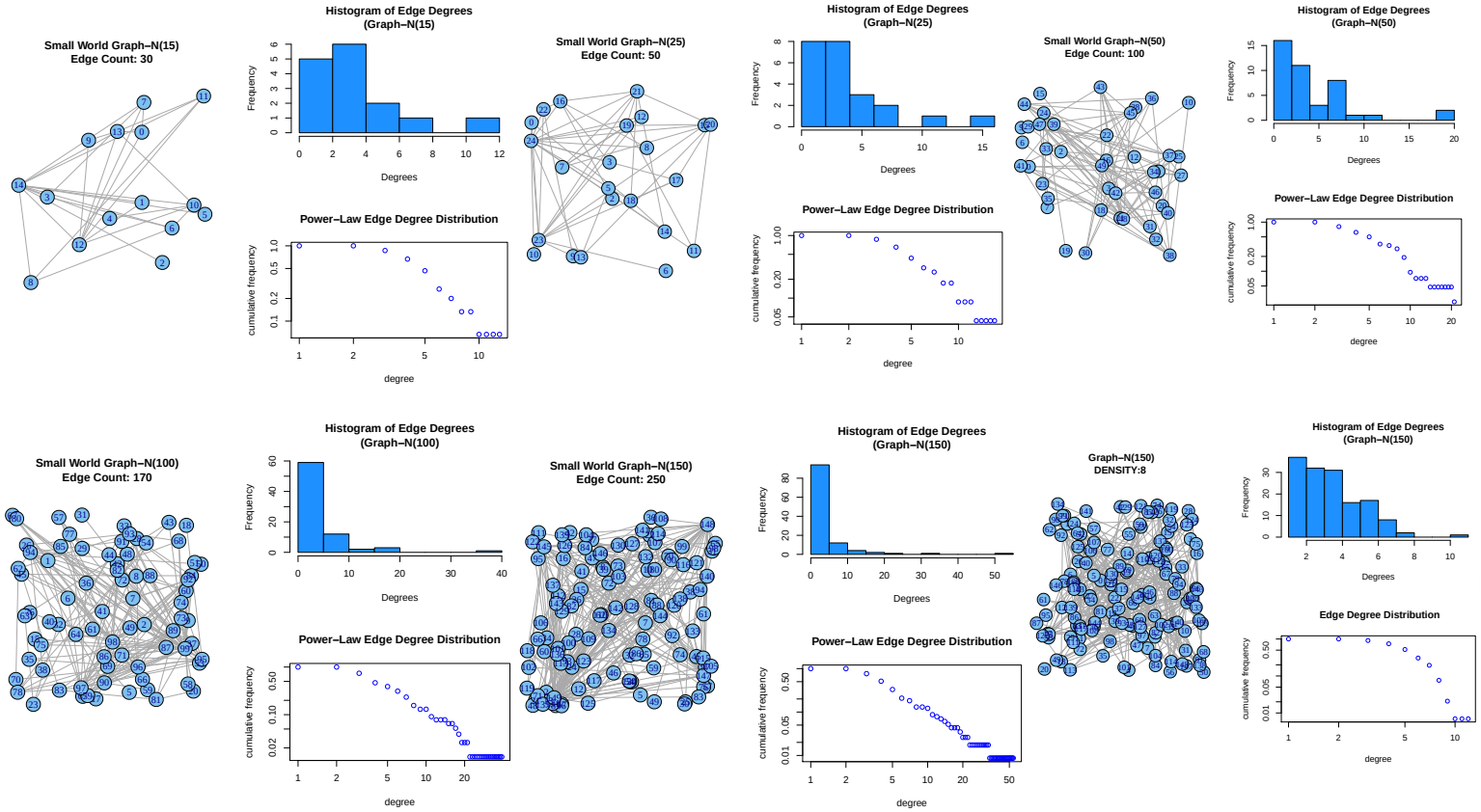


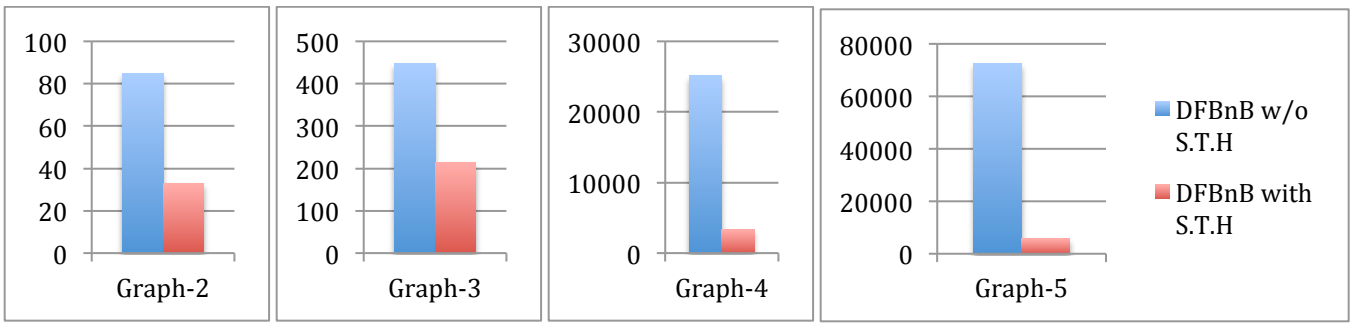
Figure-10 Test Graphs
(First five graphs are power-law graphs last one is random graph with density=8, n=150)

6.1 Pruning Performance

We compared the pruning performance of DFBnB without spanning tree heuristic to pruning performance of DFBnB with spanning tree heuristic. We experimented on random graphs, which have power-law edge degree distribution that were shown above. Table-1 presents the results of pruning performance of algorithms. These numbers in the table are represents the remaining number of leaf nodes (nodes on the level of n in search tree where n is the number of vertices in the graph). As we explained in the section-2 (detailed problem space) that each leaf of the problem space tree corresponds to different subset of vertices when we assume that leaf node represent the all nodes in the branch which is from root node to leaf node. In the course of searching in the problem space tree some of the branches are pruned so pruning decreases the number of leaf nodes in the tree. If the problem space tree has fewer leaf nodes after the search is completed that means higher amount of pruning was occurred during the search and higher number of leaf nodes means pruning performance was poor during the search. At the brute force search row of the Table-1, normally there is no pruning and each cell equals to exactly 2^n number of leaf nodes. In the column of Graph-1 there isn't any leaf node since minimum vertex cover was found during the pre-evaluation stage of our algorithm so in graph-1 we found the vertex cover set without searching. Our experiment results show that our spanning tree heuristic evaluation function always improved the pruning performance in each case and the difference between the pruning performances of two algorithms becomes larger as the number of vertices of the graph is increasing. Numbers inside the parenthesis under the graph names represents the node number of the graphs.

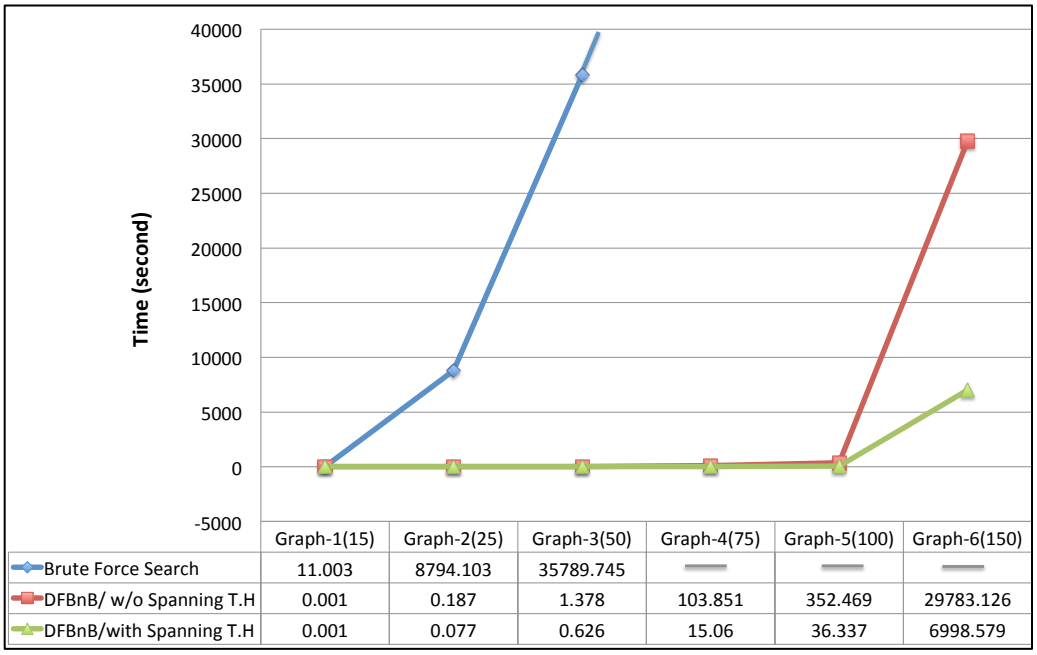
Table-1 Pruning Performance of Algorithms and their representations by histograms

Algorithm	Graph-1 (15)	Graph-2 (25)	Graph-3 (50)	Graph-4 (75)	Graph-5 (100)
Brute Force Search	32768	33.55×10^6	11.26×10^{14}	12.68×10^{29}	14.27×10^{44}
DFBnB / Without Spanning Tree Heuristic	0	85	449	25148	72713
DFBnB/ With Spanning Tree Heuristic	0	33	214	3361	5896



6.2 Running Time Performance

We compared the running time performance of DFBnB without spanning tree heuristic to running time performance of DFBnB with spanning tree heuristic. We experimented these algorithms on our test graphs mentioned in the previous section. It is obvious that our DFBnB spanning tree heuristic algorithm performs better than DFBnB without spanning tree heuristic algorithm in every graph. Moreover the difference between the running time performances of two algorithms become larger as the number of vertices in the graphs is increasing. The running time comparison of brute force search and other two algorithms is shown in Figure-11 below.



	Graph-1(15)	Graph-2(25)	Graph-3(50)	Graph-4(75)	Graph-5(100)	Graph-6(150)
Brute Force Search	11.003	8794.103	35789.745			
DFBnB/ w/o Spanning T.H	0.001	0.187	1.378	103.851	352.469	29783.126
DFBnB/with Spanning T.H	0.001	0.077	0.626	15.06	36.337	6998.579

Figure-11 Running time performance comparison
Results are shown in seconds

We compared the running time performance of our DFBnB with spanning tree heuristic to running time performance of DFBnB with (Edge count of remaing graph / Max edge degree in remaining graph) heuristic as Prof. Korf recommended during our presentation. We added this approach into our implementation (see Appendix-A) and we also tested this heuristic besides our algorithm. We created more test graphs to test two algorithms for recognizing the difference between well. Besides the previous graphs, which have the power-law edge degree, we produce three more graphs by Erdos/Renyi graph generator [13]. These graphs have node counts: 15,30 and 50. Each graph has the same density value: 4. Therefore, each edge was added to these graphs with probability D/N where N denotes the number of nodes and D is denotes the density (average degree of the graph). This approach also experimented by Spencer W. Sutterlin from class during his project. Table-2 shows the running time performance of two algorithms. These experiments shows that running time of our DFBnB with spanning tree heuristic algorithm is better than other algorithm in every case. Moreover the difference between the performances of two algorithms becomes larger when the number of nodes in the graph is increasing. This results are not surprising because our spanning tree based heuristic evaluation function solves optimally the minimum vertex cover problem in the spanning tree of the remaining graph and that leads us to

more accurate estimation of remaining number of vertices to reach a minimum vertex cover of given graph.

	Graph1 (15)	Graph2 (25)	Graph3 (50)	Graph4 (75)	Graph5 (100)	Random Graph(15)	Random Graph(30)	Random Graph(50)
DFBnB-1	0.001	0.247	1.297	56.326	209.825	0.458	16.834	1143.57
DFBnB-2	0.001	0.077	0.626	15.06	36.337	0.438	10.670	618.220

Table-2: Comparison between two algorithms:
DFBnB-1: DFBnB with (Edge count of remaining graph / Max edge degree in remaining graph) heuristic
DFBnB-2: DFBnB with Spanning Tree Heuristic
"Results are shown in seconds."

6.3 Comparison with 4-clique Additive Pattern Database Heuristic

The best-known admissible heuristic for minimum vertex cover problem is a four-clique additive pattern database heuristic that was purposed by Felner et al [1]. They also purposed to dynamically partitioning the problem for each state of the search. This approach also known as the most effective approach for large problems. They experimented their algorithm on random graphs, which were built as follows. Given two parameters, N to denote the number of vertices and D to denote the density (average degree of the graph), they created a graph with N vertices such that each edge was added to the graph with probability D/N. Their test graphs are not available therefore we created the random graph with same features with igraph graph generation functions. These codes are also available in the Appendix-B. Their experiments show that their algorithm can solve minimum vertex cover problem on a graph that has 150 nodes and D=8 in 26 seconds. Unfortunately our DFBnB algorithm with spanning tree heuristic algorithm couldn't advance this state-of the art. The main reason for that spanning tree heuristic evaluation function spends O(n) time at each calculation of heuristic value but getting heuristic value in pattern databases takes only O(1) time. Figure-12 depicts the comparison between two algorithms.

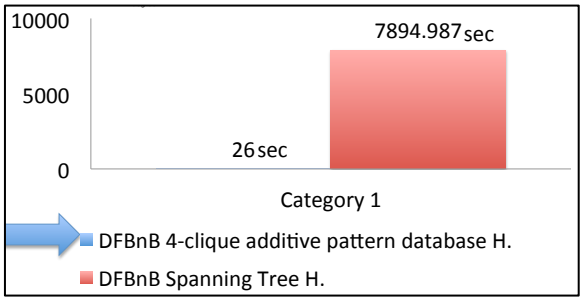


Figure-12: Running time performance comparison with 4-clique pattern database heuristic

7. Conclusion

In this project, we worked on the simplified influence maximization problem in social networks. We show that it is almost the same as the minimum vertex cover problem which is known to be NP-complete. There are classical approximation algorithms that gives near-optimal solutions and four-clique additive pattern database heuristic based search algorithm, which is purposed by Felner et al [1]. Their algorithm is the best-known algorithm that gives optimal solutions for minimum vertex cover problem. In addition to these previous works, we proposed a

Depth-First Branch-and-Bound (DFBnB) algorithm with Spanning Tree Heuristic that is a new heuristic evaluation function which is based on computing minimum vertex cover of the spanning tree of the given graph. We also adopted the three pruning techniques of inclusion/ exclusion problem space tree [1]. In addition to the pruning techniques we also used pre-evaluation function, which reduces the given graph before starting the searching of problem space tree. Our experiments show that our DFBnB algorithm with spanning tree heuristic gives better results for running time in social networks that typically have power-law edge degree distribution. However our algorithm couldn't advance the running time of the four-clique additive pattern database heuristic based search algorithm in random graphs.

References

[1] A.Felner, R.E.Korf,S.Hanan, "An Additive Pattern Database Heuristic", Journal Of Artificial Intelligence Research, Volume 22, pages 279-318, 2004; doi:10.1613/jair.1480

[2] J. Kleinberg and E. Tardos. *Algorithm Design*. (Pages: 522, 523, 524), Pearson Education/Addison Wesley, first edition,2006.

[3] D. Kempe, J. M. Kleinberg, E. Tardos, "Maximizing the spread of influence through a social network" in *Proceedings of the 9th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2003, pp. 137-146.

[4] Christos Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. In *STOC 88: Proceedings of the twentieth annual ACM symposium on Theory of comput- ing*, pages 229–234, New York, USA, 1988. ACM Press

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., New York, 1979.

[6] M. Granovetter. Threshold models of collective behavior. *American Journal of Sociology* 83(6):1420-1443, 1978.

[7] W.Chen, Y.Wang, and S.Yang, "Scalable influence maximization in social networks under the linear threshold model", in *ICDM 2010*.

[8] Natallia Kokash, "An introduction to heuristic algorithms", *University of Trento,Italy*.

[9] A.Goyal, W.Lu, L.V.S.Lakshmanan. "SIMPATH: AN Efficient Algorithm for Influence Maximization under the Linear Threshold Model"

[10] D.Easley, J.Kleinberg, "Networks, Crowds, and Markets: Reasoning about Highly Connected World", Cambridge University Press.

[11] L.Adamic, R.Lukose,A.Puniyani,B.Huberman. "Search in Power-Law Networks", Physical Review E, Volume-64

[12] R.E.Korf, Heuristic Search, CS 261A UCLA Course Reader

[13]Erdos, P. and Renyi, A., On random graphs, *Publicationes Mathematicae* 6, 290–297 (1959)

[14]Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. *Phys Rev Lett* 87(27):278701, 2001