4-27-2015

Positive Influence Dominating Set Generation via a New Greedy Algorithm

Matthew Rink
Ursinus College, marink@ursinus.edu

Adviser: Akshaye Dhawan

Positive Influence Dominating Set Generation via a New Greedy Algorithm

Matthew Rink

Submitted to the faculty of Ursinus College in fulfillment
of the requirements for Honors in
Computer Science

April 27, 2015

Acknowledgements

The author would like to thank the Ursinus College Mathematics and Computer Science department for continuous support and encouragement.

Contents

Chapter 0

Abstract

Current algorithms in the Positive Influence Dominating Set (PIDS) problem domain are focused on a specific type of PIDS, the Total Positive Influence Dominating Set (TPIDS). We have developed an algorithm specifically targeted towards the non-total type of PIDS. In addition to our new algorithm, we adapted two existing TPIDS algorithms to generate PIDS. We ran simulations for all three algorithms, and our new algorithm consistently generates smaller PIDS than either existing algorithm, with our algorithm generating PIDS approximately 5% smaller than the better of the two existing algorithms.

Chapter 1

Introduction

When working with a collection of objects, it is frequently useful to identify and select a subset of that collection to concentrate on. Sensor networks, for example, have interesting problems such as communication between sensors and how to choose which sensors will act as communication hubs. In the realm of Social Networks, we might wish to advertise to a group of users that we deem as the most influential. These users usually have the most connections to other users in the social network. The authors of [4] target a group of users to reinforce a positive behavior in the network, e.g. curbing binge drinking. To perform these types of selections, we must first define a way to represent social networks, and then define a selection criteria.

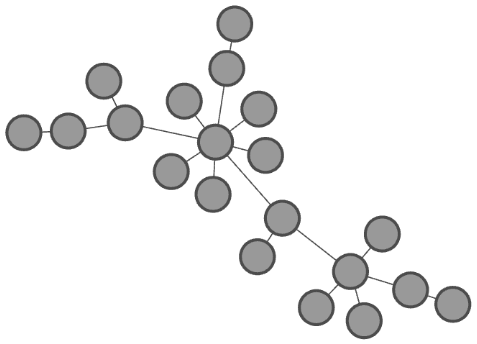
1.1 Preliminaries

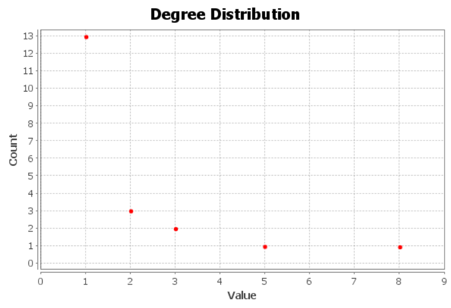
1.1.1 Modeling Social Networks

We model social networks using the mathematical concept of a graph. A graph is a collection of nodes with edges between them. For a social network, the nodes would represent users in the network and the edges would represent the social connections between the users (for example, ”is friends with” in Facebook or ”follows” in Twitter). We represent a graph as follows: $G = (V, E)$, where G is our graph composed of V, the collection of nodes, and E, the collection of edges. The elements in E take the form of a pair (a, b), where $a, b \in V$ and (a, b) represents the notion that a is ”friends with” or ”follows” b. For our research, we are exclusively working with undirected graphs, which means that $(a, b) \rightarrow (b, a)$; an undirected graph in this instance represents a social network where the relationships are always mutual and bidirectional.

Scale-Free Networks

Scale-free networks are a type of graph that follow a power-law degree distribution. This means that there are few nodes with relatively high degree (which can be viewed as "hubs"), and many nodes with low degree. A small example along with the degree distribution is shown in Figure 1.2.


(a) Example graph



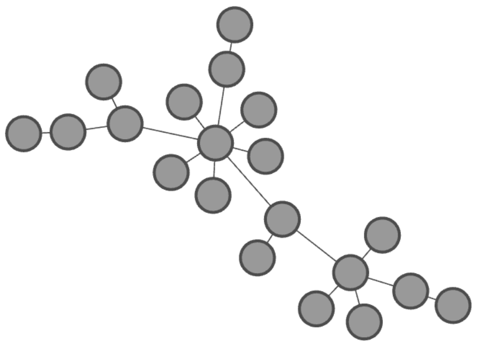
(b) Degree distribution of the graph

Figure 1.1: Example of a power-law graph

[2] showed that social networks tend to form scale-free networks. Thus for our research we model social networks by randomly generating graphs using the BarabsiAlbert model of preferential attachment.

1.1.2 PIDS

A Positive Influence Dominating Set (PIDS) is a subset P of a graph $G = (V, E)$ such that for each node $u \in V$, if less than some fraction (for the purposes of this research and in the literature, this fraction is fixed at 0.5) of the neighbors of u are not in P, then $u \in P$. Figure 1.2 shows an example of a graph and a PIDS on that graph. The nodes in the set P are shown in red.


(a) Example graph

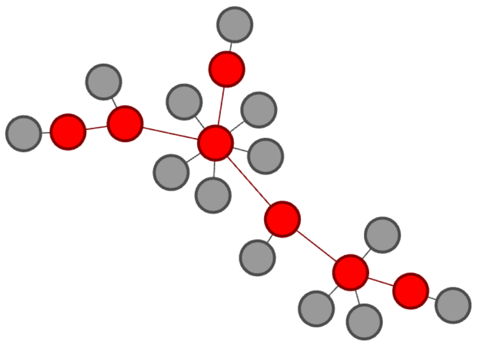
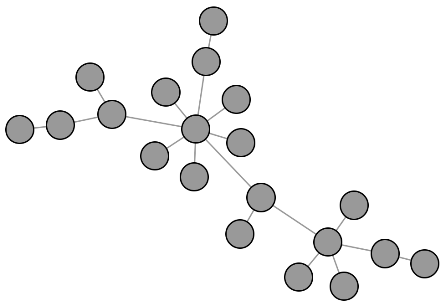

(b) PIDS on the graph

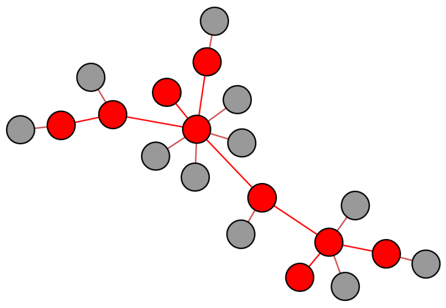
Figure 1.2: Example of PIDS on a power-law graph

A Total Positive Influence Dominating Set (TPID) is a similar to a PIDS, with a single alteration. While in a PIDS P, the nodes in P are exempt from the requirement of needing half their neighbors to be in P, in a TPIDS this is not true. Therefore, in a TPIDS T, *every* node in V has half of its neighbors in T, including the nodes in T itself. Note the difference between 1.3b and 1.2b.

The difference in definition between TPIDS and PIDS is simple, but it is enough that algorithms that are suited to creating one may not be suited for creating the other. We will demonstrate this through simulations in Section 4.2.



(a) Example graph



(b) TPIDS on the graph

Figure 1.3: Example of TPIDS on a power-law graph

Chapter 2

Related Work

While the topic of dominating sets has decades of research behind it, the subproblem of Positive Influence Dominating Sets (PIDS) is relatively recent. The problem was introduced in 2009 in [4].

2.1 Introduction of Positive Influence Dominating Sets

Wang et al. first introduced the PIDS problem in context of looking for a solution to social issues. An example the paper cites is the issue of binge drinking on college campuses - intervention programs may not have the budget to advertise to or take in all binge drinkers. It is then desirable to select a subset of the binge drinkers such that if the binge drinker is not directly participating in an intervention program, at least half of their binge drinking friends are.

For the problem application of binge drinking intervention, Wang et al categorized nodes as positive, neutral, or negative. They collapsed neutral and negative into a single negative category, and then used the two remaining categories (positive and negative) as a starting point for their PIDS. Nodes that are initially positive (in the notation of the paper, the nodes in C) are able to influence the other nodes they are connected to, and in some sense part of the PIDS and in others not. To clarify, the subset P that is output by the algorithm proposed in [4] is not a PIDS by itself, but $P \cup C$ will be the complete PIDS of the graph.

The strategy to select a PIDS used by Wang et al in [4] is to repeatedly select a 1-dominating set of the nodes $V - P \cup C$ that have yet to be dominated, and afterwards dominate the nodes in $P \cup C$ by selecting the highest degree neighbors of those nodes.

The notion of the initial positive compartment C is one that is generally not seen beyond this paper. Wang et al. do mention that a graph with $C = \emptyset$

is one where every node is initially negative, which is the assumption most PIDS research has used. It is important to note that in this paper ([4]), the definition provided for PIDS is what is later called a TPIDS by [1], and so for the purposes of the current work we use the definitions in [1].

2.2 Greedy Algorithm for PIDS Generation

In their follow up to [4], Wang et al. provide their greedy algorithm for generating a PIDS, along with an analysis for the approximation ratio of that algorithm. Their proposed algorithm for generating a PIDS P on a graph $G = (V, E)$ is as follows:

Notation	Definition
$n_P(u)$	denotes the neighbors of u in P
$deg(u)$	is the degree of u
$h(u)$	$\lceil deg(u) \rceil$
$f(P)$	$\sum_{u \in V} min(h(u), n_P(u))$

Algorithm 2.1 WangGreedy algorithm for TPIDS generation

```
1: $P \leftarrow \emptyset$
2: while $f(P) < \sum_{u \in V} h(u)$ do
3:     select $u \in V - P$ to maximize $f(P \cup \{u\})$
4:     and set $P \leftarrow P \cup \{u\}$
5: end while
6: return $A$
```

This algorithm will be referred to as WangGreedy for the purposes of this paper.

The strategy for algorithm 2.1 is to greedily choose the node that maximizes their evaluation function f. This function sums the number of neighbors that each node has in the (T)PIDS-in-progress P, with the stipulation that each node can only contribute up to half its neighbors to the sum. For example, if a node a has 2 of its 8 neighbors in P, then it contributes 2 to the sum; if a node b has 5 of its 6 neighbors in P, it only contributes 3, since 3 is the minimum of 5 and 3. It is important to note that node b in this example is *satisfied*, in that at least half of its neighbors are in P.

We now take a look at how WangGreedy's evaluation function f behaves when generating TPIDS in comparison to generating PIDS. When generating a TPIDS, the node that maximizes f is simply the node that touches the most unsatisfied nodes. To see this, consider first that a TPIDS-in-progress P is empty. Then the node that maximizes f will be the one with the highest degree, since at first f is 0 because no nodes have any neighbors in P and therefore all nodes are unsatisfied. Now consider the case where P already contains some nodes. Then the node u that maximizes f will also be the one that touches the most unsatisfied nodes, since nodes that are already satisfied by P cannot contribute any more to the sum in f because of the *min* limitation, therefore the only way f will increase is if u touches unmet nodes.

We adapt WangGreedy to generate PIDS by altering its evaluation function:

$$f(P) = \sum_{u \in V-P} min(h(u), n_P(u))$$

Likewise, we also change the *while* loop condition to be $f(P) < \sum_{u \in V-P} h(u)$.

We make these changes because it accurately represents the notion that in comparison to TPIDS, a PIDS does *not* require the nodes in P to also have half their neighbors in P. However, when adapted to the requirements for PIDS, WangGreedy no longer chooses the nodes that touch the most unmet nodes. This is because the evaluation function f no longer considers nodes in P. For example, take a PIDS-in-progress P that currently only contains a node a. Node a touches 4 unsatisfied nodes b, c, d, e, so $f(P) = 4$. Say node b touches 2 unsatisfied nodes, f and g, and neither f nor g touch a. If we wish to evaluate $f(P \cup \{b\})$ we must note that, unlike the case of TPIDS, we do not simply add the number of unsatisfied nodes to $f(P)$. If we did, $f(P \cup \{b\})$ would be 6. However, note that $f(P \cup \{b\}) = 5$, since c, d, e each touch one node in $P \cup \{b\}$ (they touch a) and f, g each touch one node (b). Note that since we added b to our PIDS, we no longer consider it when evaluating our PIDS via f. We can view this as a subtraction from $f(P)$, in that where b previously contributed 1 to the f sum, it now contributes nothing, unlike our case when building a TPIDS where both a and b would still be able to contribute 1 to $f(P \cup \{b\})$ for a total sum of 7. As we show in our experiments (4), this means WangGreedy doesn't value nodes as efficiently when generating PIDS.

2.3 A New Greedy Algorithm for TPIDS Generation

The authors of [3] take a very different greedy approach to generating a TPIDS. Their algorithm is as follows (from here on referred to as Raei-Greedy):
Notation for a node u:

$$\textit{need-degree}: \left\lceil \frac{deg(u)}{2} \right\rceil$$

$$\textit{cover-degree}: \sum_{w \in n(u)} \textit{need-degree}(w)$$

Their approach is to value a node u based on how needy u's neighbors are. This has the effect of valuing nodes that might not necessarily connect to a large number of unsatisfied nodes, but perhaps instead connect to a few

Algorithm 2.2 RaeiGreedy algorithm for TPIDS generation

```
1: $P \leftarrow \emptyset$
2: compute *need-degree* for each node
3: compute *cover-degree* for each node
4: **while** for any $u \in V(n_P(u) < \left\lceil \frac{deg(u)}{2} \right\rceil)$ **do**
5:     select $u \in V - P$ to maximize *cover-degree*
6:     and set $P \leftarrow P \cup \{u\}$
7:     subtract 1 from *need-degree* of neighbors of $u$
8:     revise *cover-degree* for all $w \in V - P$
9: **end while**
10: **return** $A$
```

very needy nodes. It is important to note that at line 7, we do not allow a node's *need-degree* to go below 0. If we do, the algorithm will devalue nodes that are connected to nodes that are already covered by more than half their neighbors.

The authors of [3] also include a complexity analysis for both their algorithm and for 2.1. Raei et al. made a slight mistake in their analysis however when considering the worst case size of a TPIDS. They declare that in the worst case, they would need $\frac{n}{2}$ nodes, however the example they give actually requires more than half of the nodes to be selected to form a TPIDS. To create a TPIDS on that graph (fig. 2.1), it is necessary to use more than 4 nodes. Their example subset on that graph does not qualify as a TPIDS, because node 8 does not have a neighbor in the TPIDS. Therefore they are not justified in saying that they need at most half the nodes for a TPIDS, since we have an example of a power-law graph in which it is not possible to construct a TPIDS with at most half the nodes.

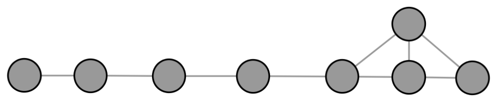


Figure 2.1: Power-law graph that needs more than half of the nodes to form a TPIDS

Chapter 3

New Greedy Algorithm

In this section, we present our algorithm for constructing a Positive Influence Dominating Set (PIDS). Since both WangGreedy [5] and RaeiGreedy [3] were designed with Total Positive Influence Dominating Sets (TPIDS) in mind, we seek to develop an algorithm specifically targeted towards generating a PIDS efficiently. We work off of the framework of the RaeiGreedy algorithm [3] to take advantage of their improved time complexity over WangGreedy, evaluating nodes base on a different criterion.

3.1 Notation

Here we define the notation necessary for our algorithm. Let $G = (V, E)$ be a graph G with nodes V and edges E. Let P be our PIDS on G. For some node $u \in V$:

Notation	Definition
$d(u)$	degree of node u
$h(u)$	$\left\lceil \frac{d(u)}{2} \right\rceil$
$n(u)$	the set of the neighbors of u
$n_P(u)$	$n(u) \cap P$

3.2 Algorithm: AltGreedy

We first define the following functions $s()$ and $g()$ on a node u :

$$s(u) = \begin{cases} 1 & \text{if } n_P(u) \geq h(u) \text{ or } u \in P \\ 0 & \text{otherwise} \end{cases}$$

$$g(u) = s(u) + \sum_{w \in n(u)} 1 - s(w)$$

Our first function, $s(u)$, represents whether a node is satisfied under the definition of a PIDS. To adapt s to test satisfaction under TPIDS, we simply remove the "or $u \in P$" qualification, since in TPIDS *every* node needs to have at least half of its neighbors in our TPIDS subset (sec. 1.1.2).

Our second function, $g(u)$, tallies the number of unsatisfied neighbors a node u touches. To this tally, we add 1 if u is unsatisfied. We do this because as noted in the definition of PIDS (sec. 1.1.2), a node can be satisfied either by having half of its neighbors in our PIDS P, or by being in P itself. Thus we increment the tally by 1 to indicate that selecting this node comes with the added benefit of satisfying it instantly.

Our algorithm is as follows:

```
1: $P \leftarrow \emptyset$
2: compute $g(u)$ value for all $u \in V$
3: **while** $P$ is not a PIDS **do**
4:     select $u \in V - P$ to maximize $g(u)$
5:     and set $P \leftarrow P \cup \{u\}$
6:     revise $g(w)$ values for all $w \in V - P$
7: **end while**
8: **return** $P$
```

Our algorithm's strategy for generating a PIDS is to greedily select the node that is connected to the most unsatisfied nodes, and add that to the PIDS-under-construction. To accomplish this, after creating our empty set A that will become our PIDS at termination, we compute g values for all the nodes in our graph V. Our loop simply chooses the node with the maximum g value, adds that to A, and recomputes the g values for the remaining nodes that are not in A. This is very similar in flow to the algorithm presented by Raei et al. (sec. 2.3) [3], but we are substituting RaeiGreedy's cover-degree criterion with our g function.

We chose to use the strategy of selecting the node that is connected to the most unsatisfied nodes, taking into account whether the node in question is unsatisfied or not, because we are specifically targeting PIDS generation with our algorithm. Both algorithms WangGreedy 2.1 and RaeiGreedy 2.2 ([5],[3]) are aiming to construct a TPIDS. Section 3.3.2 simulates those algorithms running on an example graph, and section 4 examines the performance of those algorithms via several experiments.

Complexity Analysis

Since the psuedocode is structured much like that of RaeiGreedy 2.2 [3], we are able to reference their time complexity proof ([3]) to show that our algorithm is $O(n^2)$.

The loop in line 3 will run at most n times, since we can only add as many nodes to our PIDS A as we have altogether, and each iteration of the loop adds a node from V to A.

The first thing we do in the loop is to check if A is a PIDS or not, and only continue if the latter is true (this also takes place on line 3). In the worst case, we need to evaluate every node to check if it satisfies the PIDS conditions, so the complexity of this step is $O(n)$.

Lines 4 and 6 of our algorithm both take $O(n)$, since we will have to choose from and revise first n nodes, then $n-1$ nodes, etc. Line 5 takes constant time ($O(1)$).

Therefore, our algorithm's time complexity is $O(n^2)$.

3.3 Example

We will demonstrate our algorithm on a small example graph, and afterwards demonstrate the other two algorithms, WangGreedy and RaeiGreedy, to better illustrate the differences between them.

3.3.1 Our Algorithm

We choose a node $u \in V$ to maximize $g(u)$. For the first round, the $g(u)$ values are as follows (along with the f values for WangGreedy and the cover-degrees for RaeiGreedy):

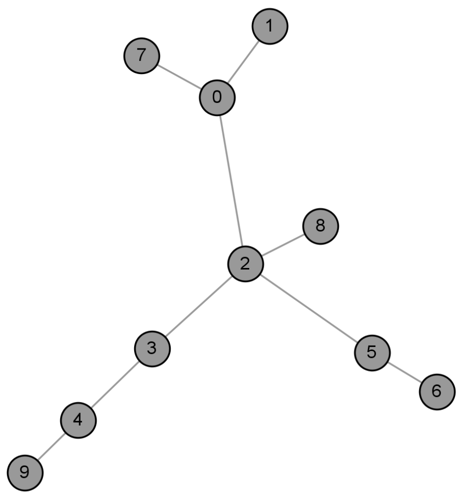


Figure 3.1: Small graph V before selecting any nodes

Node u	g(u)	Node u	g(u)
0	4	5	3
1	2	6	2
2	5	7	2
3	3	8	2
4	3	9	2

As seen, node 2 is the most desirable node, since it touches the most unmet nodes (4), and is unsatisfied currently (+1). We thus add node 2 to our work-in-progress PIDS A.

We now recalculate g values:

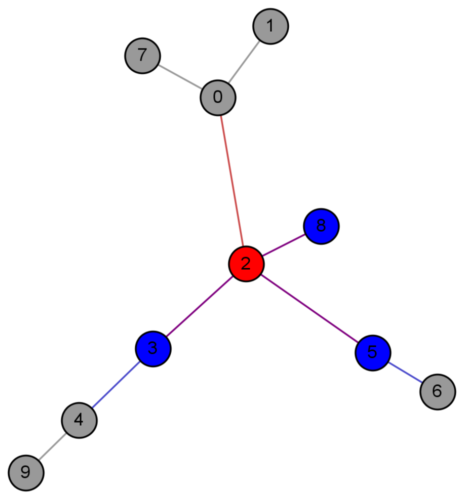


Figure 3.2: After selecting node 2. Red nodes are part of the PIDS A, blue nodes are satisfied by A, and grey are unsatisfied nodes.

Node	g(u)	Node	g(u)
0	3	5	1
1	2	6	1
2	N/A	7	2
3	1	8	0
4	2	9	2

Node 2's score is not considered since since it is already in A. Therefore our best choice for this round is node 0, since it is connected to two other unsatisfied nodes and is unsatisfied itself.

Once again, we recalculate g values:

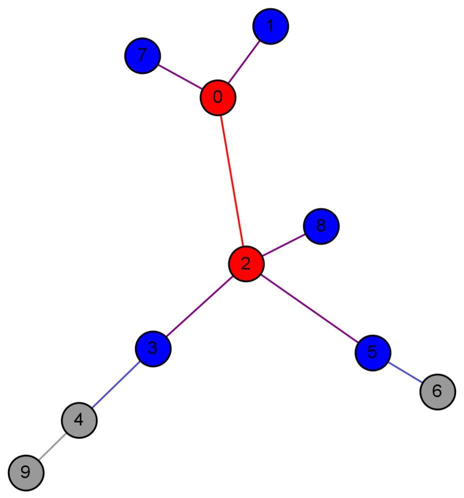


Figure 3.3: After selecting node 0. Red nodes are part of the PIDS A, blue nodes are satisfied by A, and grey are unsatisfied nodes.

Node	g(u)	Node	g(u)
0	N/A	5	1
1	0	6	1
2	N/A	7	0
3	1	8	0
4	2	9	2

Note now that nodes 4 and 9 are valued equally by our evaluation function g. For this example, we break the tie randomly, choosing node 9 to add to A.

We revise our g scores a final time:

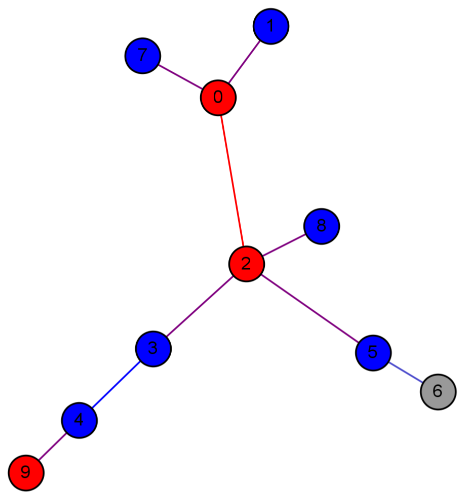


Figure 3.4: After selecting node 9. Red nodes are part of the PIDS A, blue nodes are satisfied by A, and grey are unsatisfied nodes.

Node	g(u)	Node	g(u)
0	N/A	5	1
1	0	6	1
2	N/A	7	0
3	0	8	0
4	0	9	N/A

Once again we have a tie, so for this example we break it randomly and choose node 5.

We now have a PIDS A. We can verify this by observing that every blue node has at least half of its neighbor's colored red (which represents that they are in A).

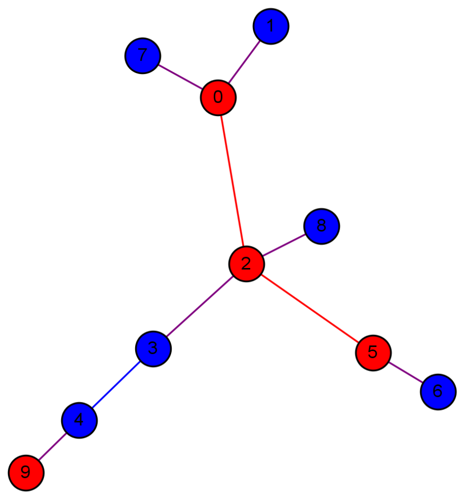


Figure 3.5: After selecting node 5. Red nodes are part of the PIDS A, blue nodes are satisfied by A, and grey are unsatisfied nodes.

3.3.2 WangGreedy and RaeiGreedy

We now demonstrate the differences between the decisions our algorithm made compared to WangGreedy and RaeiGreedy. The tables below each figure feature a column for WangGreedy (the f column) and two columns for RaeiGreedy (the need-degree and cover-degree columns).

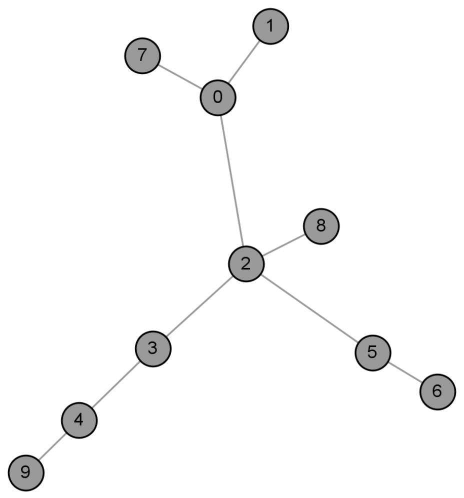


Figure 3.6: Small graph V before selecting any nodes

Node	$f(P \cup u)$	need-degree	cover-degree
0	3	2	4
1	1	1	2
2	4	2	5
3	2	1	3
4	2	1	2
5	2	1	3
6	1	1	1
7	1	1	2
8	1	1	2
9	1	1	1

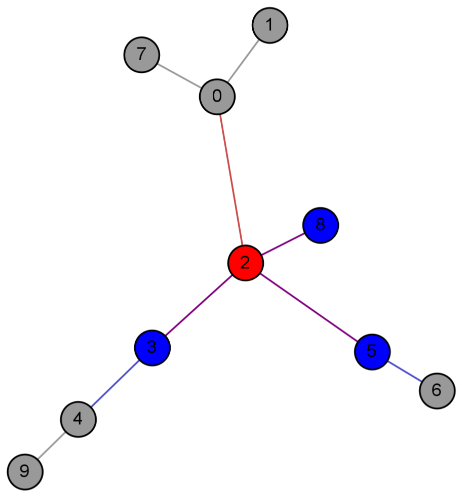


Figure 3.7: Node 2 added to P

Node	$f(P \cup u)$	need-degree	cover-degree
0	5	1	2
1	5	1	1
2	N/A	0	N/A
3	4	0	1
4	5	1	1
5	4	0	1
6	4	1	0
7	5	1	1
8	3	0	0
9	5	1	1

Note that WangGreedy values many nodes equally when both RaeiGreedy and AltGreedy valued a few nodes higher than the rest. Specifically note that nodes 1 and 7 are valued the same as node 0 in WangGreedy, when RaeiGreedy and AltGreedy valued 0 more than 1 and 7. For the sake of this example, we say WangGreedy gets lucky and randomly breaks the tie selecting node 0.

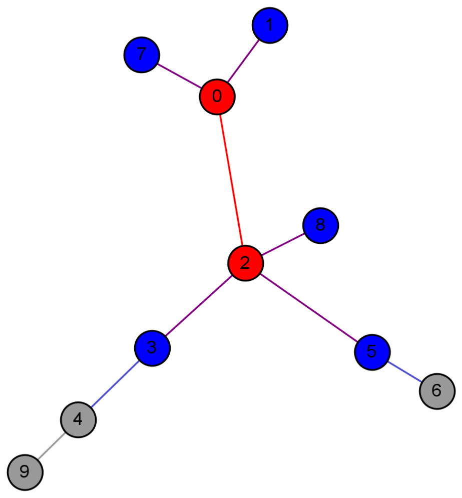


Figure 3.8: Node 0 added to P

Node	$f(P \cup u)$	need-degree	cover-degree
0	N/A	0	N/A
1	4	0	0
2	N/A	0	N/A
3	5	0	1
4	6	1	1
5	5	0	1
6	5	1	0
7	4	0	0
8	4	0	0
9	6	1	1

Here RaeiGreedy is unable to value nodes 4 and 9 higher than node 3. For the sake of this example, we once again say that both algorithms got lucky and broke the tie to select node 9.

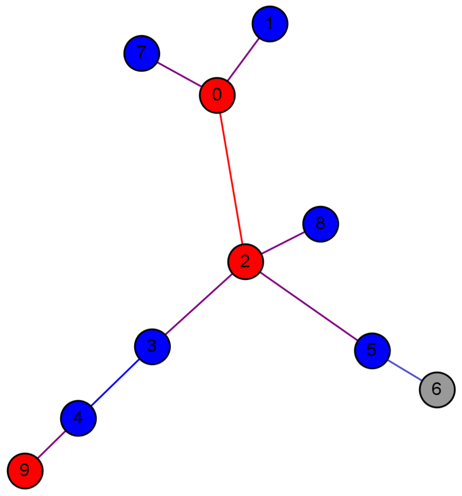


Figure 3.9: Node 9 added to P

Node	$f(P \cup u)$	need-degree	cover-degree
0	N/A	0	N/A
1	5	0	0
2	N/A	0	N/A
3	5	0	0
4	5	0	0
5	6	0	1
6	6	1	0
7	5	0	0
8	5	0	0
9	N/A	0	N/A

For our final node, we say that WangGreedy randomly breaks the tie choosing node 5.

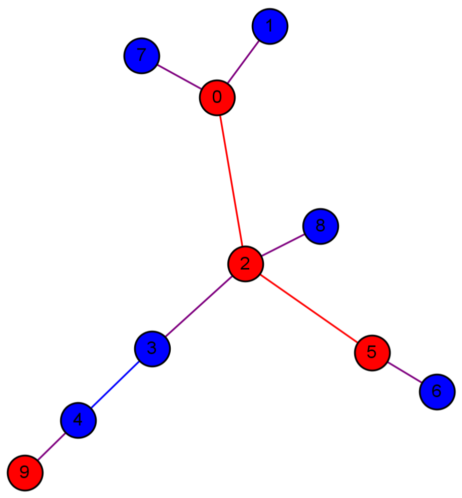


Figure 3.10: Node 5 added to P

Our final PIDS ends up being the same, but only because the ties were broken in very specific ways. It is easy to see that if they had been broken in other ways, our PIDS would have been larger. To illustrate this, here 3.11 shows the final PIDS if WangGreedy had broken the tie at 3.7 differently. While in this small graph, the result is just one node too many, on larger graphs the difference is much more noticeable, and this is seen in our results in Section 4.

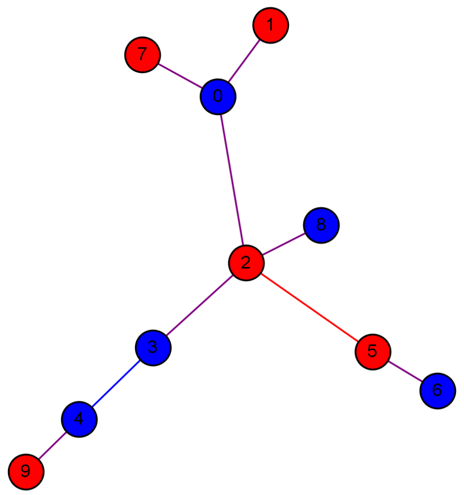


Figure 3.11: The result of WangGreedy with different random tie breakers

Chapter 4

Simulation and Results

All three algorithms were tested by having them generate PIDS on random scale-free graphs. The simulations were designed to allow us to compare our algorithms with those in [3].

4.1 SNAP Library

We used the Stanford Network Analysis Platform (SNAP) to implement and perform our simulations. The C++ SNAP library allowed us to easily generate scale-free networks using their implementation of the BarabsiAlbert model. It allowed us to quickly implement the algorithms in question and scale to larger graphs easily.

4.2 Simulations

Using the SNAP library to generate graphs with power-law degree distribution (via the BarabsiAlbert algorithm), we ran several simulations to compare the PIDS and TPIDS generated by the three algorithms (WangGreedy, RaeiGreedy, and AltGreedy).

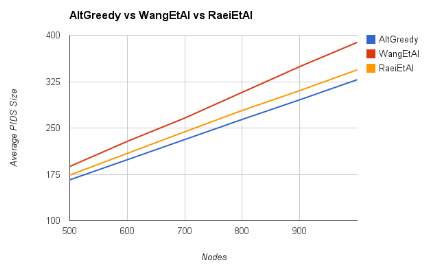
All data for our simulations is included in section 6. Figures have the table number included in the caption.

4.2.1 PIDS Generation

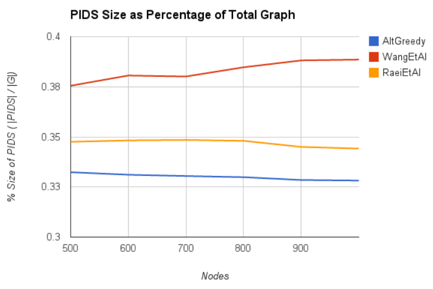
PIDS were generated on graphs of sizes 500-1000 (increments of 100), all with an average degree of 10, 600 graphs in total. The average size of the PIDS generated by the algorithms during this simulation are displayed in fig. 4.1.

In addition to these results, the RaeiGreedy and AltGreedy algorithms were run on datasets of larger graphs of sizes 5000-25000 (increments of 5000, average degree 10), seen in 4.2.

The data shows that for our simulations the PIDS generated by RaeiGreedy and AltGreedy are smaller than those generated by WangGreedy. In addition, on average the PIDS generated by our AltGreedy algorithm are

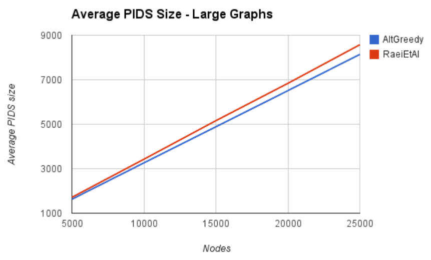


(a) Average PIDS size

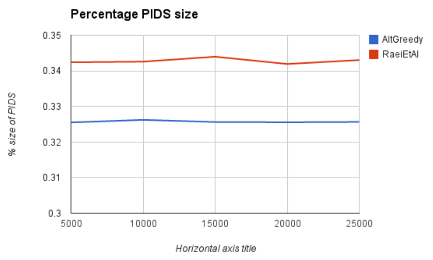


(b) Average PIDS %

Figure 4.1: PIDS Results for # nodes $n = 500 \rightarrow 1000$, 100 iterations per size (table 6.1)



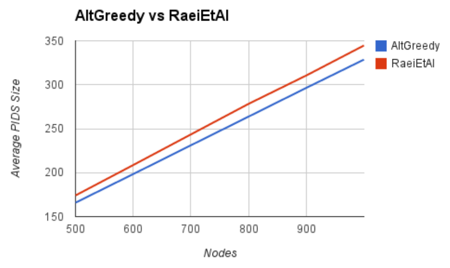
(a) Average PIDS size



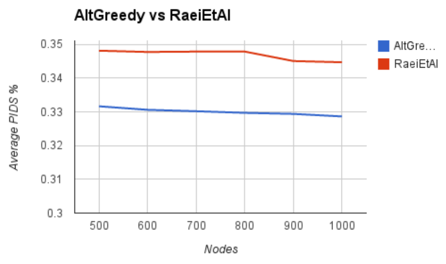
(b) Average PIDS %

Figure 4.2: PIDS Results for # nodes $n = 5000 \rightarrow 25000$ (table 6.2)

smaller than those generated by RaeiGreedy. To further demonstrate this, we ran the simulation on a larger dataset of 6000 graphs, sizes 500-1000 in increments of 100 with an average degree of 10. This data confirms the aforementioned results that AltGreedy produces smaller PIDS than RaeiGreedy.



(a) Average PIDS size



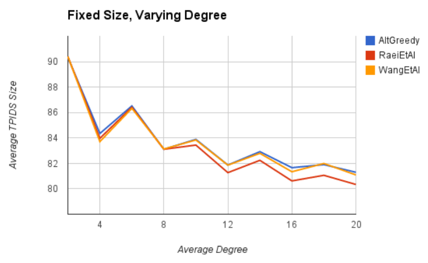
(b) Average PIDS %

Figure 4.3: PIDS Results for # nodes $n = 500 \rightarrow 1000$, 1000 iterations per size (table 6.3)

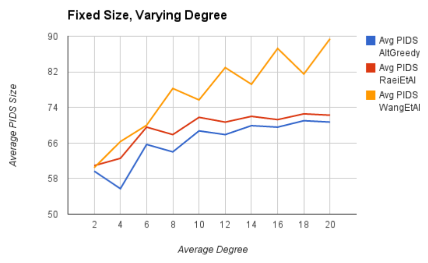
4.2.2 TPIDS Generation

We also ran simulations for the Total Positive Influence Dominating Set (TPIDS) generation. As expected, on average the TPIDS were larger than the PIDS for the same graph. Interestingly, TPIDS and PIDS exhibit opposite behaviors for certain tests. To replicate an experiment from [3], we used all three algorithms to generate both TPIDS (as in the original experiment) and also PIDS for graphs of fixed size 200, while incrementing the average degree by 2.

For TPIDS, the data corresponds to the results published in [3], in that the higher the average degree, the smaller the average TPIDS. However, the opposite behavior is exhibited when we generate a PIDS, i.e., a higher average degree means we select more nodes. This behavior makes sense, because when



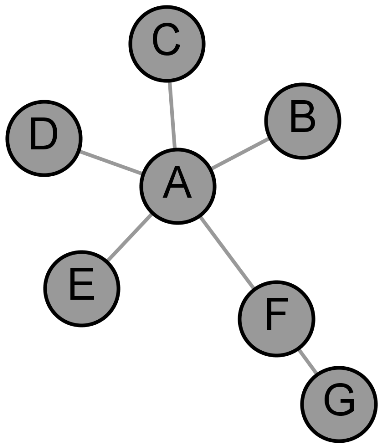
(a) Average TPIDS Size



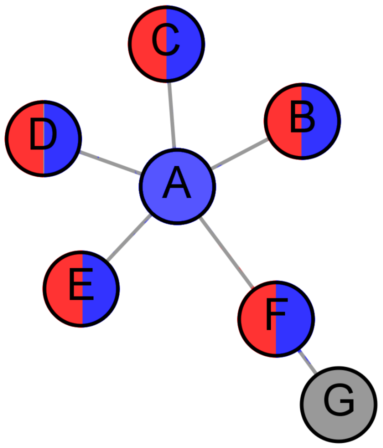
(b) Average PIDS Size

Figure 4.4: PIDS and TPIDS results for fixing number nodes $n = 200$, varying average degree $2 \to 20$ in increments of 2 (tables 6.5 and 6.4)

generating a TPIDS (T), not only do the nodes not in T need to have half of their neighbors in T, but so do the nodes in T itself. When there is a low average degree, and the graphs follow a power-law degree distribution, the graphs have only a few nodes with high degree and many nodes with low degree. Figure 4.5 demonstrates this effect on a small scale, with blue nodes being satisfied according to PIDS and red nodes being satisfied according to TPIDS, red/blue nodes satisfied under both, and grey nodes unsatisfied. When we select the node A, all of its neighboring nodes are now satisfied under both TPIDS and PIDS. However, A itself is not satisfied under TPIDS, so if we were trying to generate a TPIDS we would need to select some of A's neighbors. This explains why there is a reverse trend in TPIDS and PIDS when the average degree is small.



(a) Graph with small average degree



(b) Graph after selecting node A

Figure 4.5: PIDS vs TPIDS satisfaction

Chapter 5

Conclusion and Future Work

Our results show that our algorithm produces smaller PIDS than WangGreedy and RaeiGreedy; however it produces TPIDS of similar size as those generated by WangGreedy. This suggests that any further algorithmic development must be tested against both TPIDS and PIDS, since their different qualifications require different approaches. In addition, our PIDS selection function g could be refined through further experimentation (for example, weighting nodes more heavily than simply adding 1 if they are unsatisfied).

Chapter 6

Appendix: Data

Nodes	AltGreedy	WangGreedy	RaeiGreedy
500	166.18	187.75	173.76
600	198.66	228.36	208.94
700	231.32	266.08	243.92
800	263.91	307.77	278.42
900	295.63	349.32	310.49
1000	328.19	388.55	344.15

Table 6.1: PIDS Results for # nodes $n = 500 \rightarrow 1000$, 100 iterations per size (figure 4.1)

Nodes	AltGreedy	RaeiEtAl
5000	1627.52	1712.03
10000	3262.28	3425.9
15000	4884.39	5159.58
20000	6511.08	6838.65
25000	8141.11	8575.88

Table 6.2: PIDS Results for # nodes $n = 5000 \rightarrow 25000$ (figure 4.2)

Nodes	AltGreedy	RaeiGreedy
500	165.809	174.045
600	198.348	208.628
700	231.115	243.479
800	263.747	278.262
900	296.436	310.527
1000	328.619	344.667

Table 6.3: PIDS Results for # nodes $n = 500 \rightarrow 1000$, 1000 iterations per size (figure 4.3)

Degree	AltGreedy	RaeiGreedy	WangGreedy
2	59.68	60.92	60.45
4	55.74	62.56	66.33
6	65.68	69.58	70
8	64.01	67.89	78.27
10	68.73	71.78	75.69
12	67.89	70.7	82.96
14	69.91	72	79.21
16	69.57	71.26	87.26
18	71.02	72.56	81.51
20	70.71	72.26	89.44

Table 6.4: PIDS results for fixing number nodes $n = 200$, varying average degree $2 \rightarrow 20$ in increments of 2 (figure 4.4b)

Degree	AltGreedy	RaeiGreedy	WangGreedy
2	90.34	90.4	90.4
4	84.35	83.97	83.71
6	86.52	86.41	86.33
8	83.11	83.11	83.13
10	83.89	83.44	83.84
12	81.87	81.27	81.86
14	82.92	82.24	82.79
16	81.66	80.62	81.34
18	81.9	81.06	81.99
20	81.3	80.34	81.09

Table 6.5: TPIDS results for fixing number nodes $n = 200$, varying average degree $2 \to 20$ in increments of 2 (figure 4.4a)

Bibliography

[1] Thang N Dinh, Yilin Shen, Dung T Nguyen, and My T Thai. On the approximability of positive influence dominating set in social networks. *Journal of Combinatorial Optimization*, 27(3):487–503, 2014.

[2] Stephen Eubank, VS Kumar, Madhav V Marathe, Aravind Srinivasan, and Nan Wang. Structural and algorithmic aspects of massive social networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 718–727. Society for Industrial and Applied Mathematics, 2004.

[3] Hassan Raei, Nasser Yazdani, and Masoud Asadpour. A new algorithm for positive influence dominating set in social networks. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pages 253–257. IEEE Computer Society, 2012.

[4] Feng Wang, Erika Camacho, and Kuai Xu. Positive influence dominating set in online social networks. In *Combinatorial Optimization and Applications*, pages 313–321. Springer, 2009.

[5] Feng Wang, Hongwei Du, Erika Camacho, Kuai Xu, Wonjun Lee, Yan Shi, and Shan Shan. On positive influence dominating sets in social networks. *Theoretical Computer Science*, 412(3):265–269, 2011.