CS224W: Social and Information Network Analysis **Fall 2013**

Problem Set 4

Due 9:30am November 21, 2013

General Instructions

These questions require thought, but do not require long answers. Please be as concise as possible. Please fill the cover sheet (http://cs224w.stanford.edu/cover.pdf) and submit it as a front page with your answers. We will subtract 2 points for failing to include the sheet. Include the names of your collaborators (see the course website for the collaboration or honor code policy). You are allowed to take maximum of 1 late period (see course website for a definition of a late period).

Regular (non-SCPD) students should submit hard copies of the answers either in class or in the submission cabinet (see course website for location). You should also include source code and any other files you used in the paper submission.

SCPD students should submit their answers through SCPD. The submission must include the cover sheet, all the answers, the source code and the usual SCPD routing form (http://scpd.stanford.edu/generalInformation/pdf/SCPD_HomeworkRouteForm.pdf).

Additionally, all students (Regular and SCPD) should upload all code to http://snap.stanford.edu/submit/.

Questions

1 Variations on a Theme of PageRank [25 points – Christie, Yoni]

Personalized PageRank

Personalizing PageRank is a very important real-world problem: different users find different pages relevant, so search engines can provide better results if they tailor their page relevance estimates to the users they are serving. Recall from class that PageRank can be specialized with clever modifications of the teleport vector. In this question, we'll explore how this can be applied to personalize the PageRank algorithm.

Assume that people's interests are represented by a set of representative pages. For example, if Zuzanna is interested in sports and food, then we could represent her interests with the set of pages {`www.espn.com`, `www.epicurious.com`}. For notational convenience, we'll use integers as names for webpages.

(a) **[7 points]** Suppose you have already computed the personalized PageRank vectors for the following users:

- Agatha, whose interests are represented by the teleport set $\{1, 2, 3\}$,
- Bertha, whose interests are represented by the teleport set $\{3, 4, 5\}$,
- Clementine, whose interests are represented by the teleport set $\{1, 4, 5\}$, and
- DeShawn, whose interests are represented by the teleport set $\{1\}$.

Without looking at the graph, can you compute the personalized PageRank vectors for the following users? If so, how? If not, why not? Assume a fixed teleport parameter β.

i. **[2 points]** Eloise, whose interests are represented by the teleport set $\{2\}$.

ii. **[2 points]** Felicity, whose interests are represented by the teleport set $\{5\}$.

iii. **[3 points]** Glynnis, whose interests are represented by the teleport set $\{1, 2, 3, 4, 5\}$ with weights 0.1, 0.2, 0.3, 0.2, 0.2, respectively.

(b) **[3 points]** Suppose that you've already computed the personalized PageRank vectors of a set of users (denote the computed vectors V). What is the set of all personalized PageRank vectors that you can compute from V without accessing the web graph?

Spam Farms

The staggering number of people who use search engines to find information every day makes having a high PageRank score a valuable asset, which creates an incentive for people to game the system and artificially inflate their website's PageRank score. Since the PageRank algorithm is based on link structure, many PageRank spam attacks use special network configurations to inflate a target page's PageRank score. We'll explore these configurations, called *spam farms*, in this part of the question.

(c) **[5 points]** Consider the spam farm shown in Figure 1. The spammer controls a set of *boosting pages* $1, \ldots, k$ (where page i has PageRank score p_i) and is trying to increase the PageRank of the *target page* p_0. The target page receives λ amount of PageRank from the rest of the graph (represented by the dotted arrow). This means that $\lambda = \sum_{i \in S} \frac{r_i}{d_i}$, where S is the set of nodes in the rest of the network that link to p_0 and r_i and d_i represent, respectively, node i's PageRank score and outdegree. Let N denote the number of pages in the entire web, including the boosting pages and target page. Calculate the PageRank of the target page p_0 with this configuration as a function of λ, k, β, and N. Your solution should not include other parameters (such as p_i).

Hint: You can write an expression for p_0 in terms of $p_1, \ldots p_k$, β, k, and N. You can also write the PageRanks of the boosting pages in terms of p_0, β, k, and N.

(d) **[5 points]** It turns out that the structure in Figure 1 is optimal, in the sense that it maximizes the target's PageRank p_0 with the resources available. However, it may still be possible to do better by joining forces with other spammers. It also turns out that the

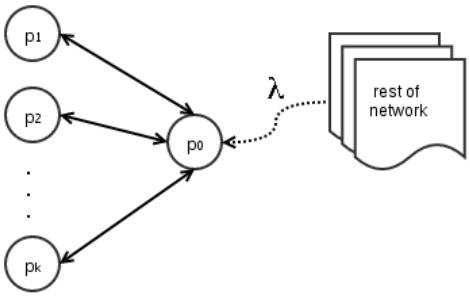

Figure 1: A spam farm.

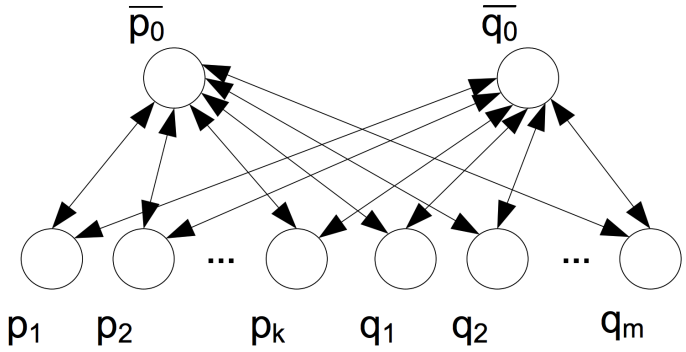

Figure 2: Linked spam farms.

λ contribution from the rest of the graph is mathematically equivalent to having some extra number of boosting pages, so for the rest of this question we'll ignore λ. Consider the case where two spammers link their spam farms by each linking to the other's target page as well as their own, as shown in Figure 2. Let p'_0 and q'_0 denote the PageRank values of the target pages if the spammers use the individual spam configuration that we discussed in the previous question (without λ this time). Calculate the PageRanks of the target pages $\overline{p_0}$ and $\overline{q_0}$ with the new configuration as a function of k, m, β, and N (where again N denotes the number of pages in the web graph). What are $\overline{p_0} - p'_0$ and $\overline{q_0} - q'_0$? Are the spammers better off than they would be if they operated independently (i.e. is $\overline{p_0} + \overline{q_0} > p'_0 + q'_0$)?

(e) **[5 points]** There are other ways spammers can form alliances. Consider the setup shown in Figure 3, where the spammers only link their target pages together. Again let p'_0 and q'_0 denote the PageRank values of the target pages that the spammers would get on their own. Calculate the PageRank of the target pages $\overline{\overline{p_0}}$ and $\overline{\overline{q_0}}$ with this configuration as a function of k, m, β, and N (where again N denotes the number of pages in the web graph). What are $\overline{\overline{p_0}} - p'_0$ and $\overline{\overline{q_0}} - q'_0$? Are the spammers better off than they would be if they operated independently (i.e. is $\overline{\overline{p_0}} + \overline{\overline{q_0}} > p'_0 + q'_0$)?

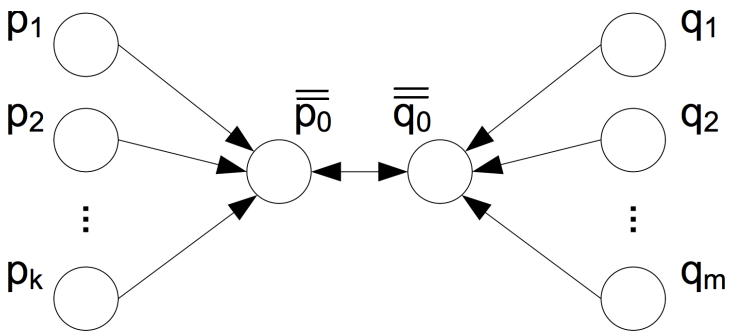

Figure 3: Another way of linking two spam farms.

What to submit

(1a) For each of (i),(ii), and (iii), 'yes' or 'no' and an explanation of why or why not.

(1b) A mathematical expression for the set in terms of V.

(1c) An expression for p_0 in terms of k,β, and N. Also show how you derived the expression.

(1d)
- Expressions for $p'_0 + q'_0$ and $\overline{p_0} + \overline{q_0}$ in terms of k, m, β and N. Show how you derived these expressions.
- A yes/no answer to whether spammers are better off and a brief explanation.

(1e)
- Expressions for $p'_0 + q'_0$ and $\overline{\overline{p_0}} + \overline{\overline{q_0}}$ in terms of k, m, β and N. Show how you derived the expressions.
- A yes/no answer to whether spammers are better off and a brief explanation.

2 Approximate betweenness centrality [20 points – Bell, Zhemin]

Betweenness centrality is an important concept in network analysis. For instance, in class you have seen the Girvan–Newman algorithm for community detection, which uses betweenness centrality as a subroutine.

Given an undirected graph $G = (V, E)$, the betweenness centrality of an edge $\{v, w\} \in E$ is defined as

$$B(\{v, w\}) = \sum_{(s,t)\in V^2} \frac{\sigma_{st}(\{v, w\})}{\sigma_{st}},$$

where σ_{st} is the number of shortest paths between s and t, and $\sigma_{st}(\{v, w\})$ the number of shortest paths between s and t that contain the edge $\{v, w\}$.

In this question, we assume graph G is connected graph. That is, there always exists a path between any two nodes in G.

Betweenness centrality can be computed using this algorithm:

Algorithm 1 (exact betweenness centrality) For each vertex $s \in V$, perform a BFS from s, which induces a BFS tree T_s. For each $v \in V$, let v's *parent set* $P_s(v)$ be defined as the set of nodes $u \in V$ that precede v on some shortest path from s to v in G. During the BFS, also compute, for every $v \in V$, the number σ_{sv} of shortest paths between s and v, according to the recurrence

$$\sigma_{sv} = \begin{cases} 1 & \text{if } v = s, \\ \sum_{u \in P_s(v)} \sigma_{su} & \text{otherwise.} \end{cases}$$

After the BFS has finished, compute the so-called *dependency* of s on each edge $\{v, w\} \in E$ using the recurrence

$$\delta_s(\{v,w\}) = \begin{cases} \frac{\sigma_{sv}}{\sigma_{sw}} & \text{if } w \text{ is a leaf of } T_s, \\ \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \sum_{x : w \in P_s(x)} \delta_s(\{w,x\})\right) & \text{otherwise;} \end{cases}$$

(for the purpose of definition, assume without loss of generality that the shortest path from s to v is shorter than to w). Finally, the betweenness centrality of $\{v, w\}$ equals $B(\{v,w\}) = \sum_{s \in V} \delta_s(\{v,w\})$. □

Algorithm 1 has a time complexity of $O(nm)$, where n and m are the numbers of nodes and edges, respectively, and there is no faster known algorithm for computing betweenness centrality exactly in undirected, unweighted graphs.

Also, using Algorithm 1, one needs to compute the betweenness centrality of *all* edges, even if one is interested only in that of some select edges.

In this problem you'll explore an approximate version of Algorithm 1. It is nearly identical to Algorithm 1, with the difference that it doesn't start a BFS from every node but rather samples starting nodes randomly with replacement. Also, it can be run for any edge $e \in E$ independently, without necessarily having to compute the centrality of all other edges as well:

Algorithm 2 (approximate betweenness centrality) Repeatedly sample a vertex $v_i \in V$; perform a BFS from v_i (as in Algorithm 1) and maintain a running sum Δ_e of the dependency scores $\delta_{v_i}(e)$ (one Δ_e for each edge e you're interested in). Sample until Δ_e is greater than cn for some constant $c \geq 2$. Let the total number of samples be k. The estimated betweenness centrality score of e is given by $\frac{n}{k}\Delta_e$. □

Theorem 1 is the approximation guarantee about Algorithm 2. You shall see during implementation, Algorithm 2 runs much faster.

Theorem 1 For $0 < \epsilon < \frac{1}{2}$, if the centrality of an edge e is n^2/t for some constant $t \geq 1$, then with probability $\geq 1 - 2\epsilon$ its centrality can be estimated within a factor of $1/\epsilon$ with ϵt samples of source vertices.

2.1 Simulation

This part asks you to implement Algorithms 1 and 2 and compare them. Proceed as follows:

- Generate a random graph following the Barabási–Albert preferential attachment model, on $n = 1000$ nodes and attaching each node to 4 existing nodes when adding it to the network (`barabasi_albert_graph(1000, 4)` in NetworkX; `snap.GenPrefAttach(1000, 4, ...)` in SNAP).
- Compute the exact betweenness centrality for all edges $e \in E$ using Algorithm 1. (Implement it yourself.)
- Compute the approximate betweenness centrality for all edges $e \in E$ using Algorithm 2. Use $c = 5$ and sample at most $n/10$ random starting nodes v_i.
- Each algorithm induces an ordering over edges with respect to betweenness centrality. Hand in the following plot: One curve for Algorithm 1, one for Algorithm 2, overlaid in the same plot. If the edge with the x-th largest betweenness centrality (according to the respective algorithm) has betweenness centrality y, draw a dot with co-ordinates (x, y). Please use a logarithmic y-axis. Comment very briefly on the curves.

What to submit

A plot, a brief comment and your code (printed, and submitted online).

3 Stochastic Kronecker Graphs [20 points - Ashley]

In this problem, we will study some mathematical properties of the stochastic Kronecker Graphs.

3.1 [**2 points**] Recall from the lecture, a Kronecker Graph can be viewed as a special case of Multiplicative Attribute Graph. Given the symmetric 2×2 probability matrix Θ_1, let $\Theta_1[1,1] = \alpha$, $\Theta_1[1,0] = \Theta_1[0,1] = \beta$ and $\Theta_1[0,0] = \gamma$, where $0 \leq \gamma \leq \beta \leq \alpha \leq 1$. The stochastic Kronecker graph based on the k^{th} Kronecker power Θ_k has 2^k nodes. Now you are asked to label each node by a unique bit vector of length k. How can you label the nodes so that the probablity of an edge between node $(u_1u_2...u_k)$ and $(v_1v_2...v_k)$ existing is $\prod_i \Theta_1[u_i, v_i]$?

3.2 [**5 points**] We define the weight of a vertex to be the number of 1's in its label. Given a node u with weight l. For any node v, let i be the number of bits where $u_b = v_b = 1$, and let j be the number of bits where $u_b = 0$, $v_b = 1$ where u_b is some bit in u and v_b is the corresponding bit in v. There are k bits in each node representation. What is $P[u, v]$? Note: Utilize the representation from 3.1.

3.3 [**5 points**] What is the expected degree of node u with weight l represented using k bits? [Hint: you may need to use the binomial formula, $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$]

Based on the result from (3.3), you can easily calculate out other properties of the graph. Assuming the graph is undirected:

3.4 [**4 points**] What is the expected number of edges in the graph? (You do not need to treat the self-loop as a special case for this part.)

3.5 [**4 points**] What is the expected number of self-loops?

What to submit

3.1) Short description (1-2 sentences).

3.2) $P[u, v]$ in terms of the variables given and show/explain work.

3.3) Expected degree of node u and show/explain work.

3.4) Expected number of edges in the graph

3.5) Expected number of self-loops in the graph

4 Anchored k-cores in Social Networks [35 points - Justin, Ashwin]

In class, we previously explored cascading behavior and the diffusion of information or behavior in a network. In other words, we usually are interested in trying to influence as much of a network as possible to adopt a certain behavior. But even after people adopt a behavior, they might later stop engaging in this behavior, causing the network of "active" users to contract. In this problem, we study how we could minimize the contraction of this active network. As our motivating example, we consider the use (or lack thereof) of Facebook among the general population.

Assume a threshold model of behavior, where a person uses Facebook only if at least k of the person's friends also use the service. If less than k friends use Facebook, then a person stops using it.

Figure 4 shows a snapshot of activity in a network of Facebook users, where edges represent friendships between users. Notice that the network of interest only consists of current Facebook users and edges between them — people who have stopped using Facebook are removed from the network. For this network, assume that $k = 3$. However, this network

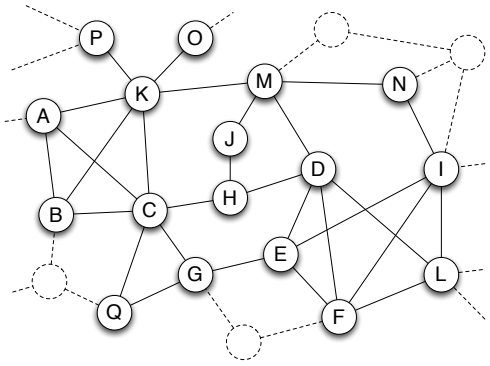

Figure 4: A Facebook network. Solid circles and edges make up the active network (people who are currently using Facebook). Dashed lines and edges indicate people who have dropped out of the network (they do not use Facebook).

is unstable (or not at equilibrium): node Q only has 2 friends using Facebook, so she will stop using Facebook (and is thus removed from the graph). This removal may in turn cause other users to also stop using the service. If every node in the graph will not change its mind about whether to use Facebook, we then say that the network is at *equilibrium*.

4.1 The equilibrium network [2 points]

We now allow nodes to make decisions on whether to stop using Facebook (and remove themselves from the network), until an equilibrium is reached. Again, $k = 3$. **Which nodes in Figure 4 make up the equilibrium network?**

4.2 Finding the k-core [3 points]

The remaining nodes that you found in Part 4.1 make up the *k-core* of a network. In the example above, you found the 3-core of the network. **Describe a simple algorithm for finding the k-core of a graph $G = (V, E)$.** You may either explain your algorithm in words or use pseudo-code.

4.3 Saving nodes [2 points]

As you might notice, the 3-core of the Facebook network is significantly smaller than the size of the original network. To protect the network from shrinking too much, suppose that you now can influence b nodes in the network so that they always use Facebook, regardless of what their friends are doing[1].

[1] You might be showing them TV advertisements, or wining-and-dining them ;)

```
V ← the set of nodes in G
A ← ∅
while b > 0 do
    V ← {u | u ∈ V, u ∉ A, u is not saved}
    for u ∈ V do
        c_u ← the number of additional nodes saved by anchoring u, including itself
    end for
    v ← arg max_u c_u    // Break ties randomly
    A ← A ∪ {v}
    b ← b − 1
end while
return A
```

Figure 5: *NaiveGreedy*(G, b).

As your time and budget are extremely constrained, suppose that you can only influence/brainwash two nodes (i.e., $b = 2$). **Which two nodes in Figure 4 should be picked to maximize the number of nodes in the remaining graph (i.e. at equilibrium)? Including the two that were picked, how many additional nodes did you "save"?**

4.4 Naive Greedy [4 points]

The two nodes that you picked in Part 4.3 correspond to *anchors* in the graph, as they can act to prevent a network from contracting too far. We now describe the *anchored k-core problem*. Consider an undirected graph $G = (V, E)$, and two parameters k and b, corresponding to the k-core and budget b you have for anchors. A solution to the *anchored k-core problem* involves picking, or *anchoring* a set of b nodes, such that the remaining size of the graph $|V|$ is maximized.

For the remainder of this question, assume that $k = 2$. That is, we are only interested in the 2-core of a graph[2].

Recall the greedy hill-climbing for influence maximization that we saw in class. We are interested in whether we can apply a similar approach in greedily picking anchors to maximize the size of the remaining graph. This greedy algorithm (*NaiveGreedy*) proceeds as follows: we pick a node to anchor that maximizes the number of additional nodes saved (breaking ties at random), and keep doing this until we have picked b nodes. Pseudo-code is provided in Figure 5.

While this approach may seem promising, the algorithm is flawed. **Construct a family of examples, such that given $b \geq 2$, *NaiveGreedy* always fails arbitrarily badly on the constructed graph.** In your example, state what the optimal solution is, and what *NaiveGreedy* will pick. Note that the algorithm should *always* fail on your example.

[2]It has been shown that for $k \geq 3$, this problem becomes significantly harder.

"Arbitrarily badly" means that for any $c \geq 0$, we can construct an example where the ratio of the size of the remaining graph in the optimal solution, to that in the greedy solution, is greater than c.

4.5 *RemoveCore*(G) [4 points]

Seeing that a naive greedy approach doesn't work out terribly well, we will instead present an algorithm that gets us closer to an exact solution. To do this, we break up the graph G into different parts.

First, we compute the k-core of G, called C_k. We then remove all edges between all vertices in C_k to produce G'.

As a result, G' is a forest (a set of trees), which can be divided into two sets R and S. Let R be the set of trees where at least 1 node in each tree belongs to the k-core, C_k, and S be the set of trees where no node in each tree belongs to C_k.

We call this operation of removing edges from the k-core of the graph *RemoveCore*(G).

If we take care in specifying which nodes are initially anchored in the new graph G', solving the anchored k-core problem for G' is equivalent to solving it for G. **Explain why this is so.** You do not need to give a precise argument.

4.6 *TwoStepGreedy*(G, b)

With *RemoveCore*(G) in hand, we now provide an algorithm that approximately solves the anchored 2-core of a graph[3]. In other words, given a budget b, this algorithm finds b nodes that approximately maximizes the size of the remaining equilibrium graph.

At a high level, instead of looking only ahead a single step (as in *NaiveGreedy*), we look ahead two steps, and pick anchors in the graph to maximize the number of nodes that are "saved". Again, we start with a budget of b, i.e. we can anchor a maximum of b nodes.

1. Run *RemoveCore*(G) to remove edges amongst nodes in the k-core, as well as any anchored nodes and the nodes they save. We thus obtain our sets of trees R and S. (Hint: the roots of trees in R correspond to nodes in C_k.)
2. Find an anchor $v_1 \in R$, that maximizes the number of nodes saved across all possible placements of a single anchor in R. (Hint: this is equivalent to finding the node that is furthest from any root of a tree in R.)

[3]In fact, we obtain a solution whose value is at least 25% of the optimal solution. While out of the scope of this problem, if we wanted to solve the problem exactly, we could instead brute force all possible combinations of anchors when $b < 5$.

```
A ← ∅   // The set of anchored nodes
while b > 0 do
    G' ← RemoveCore(G, A)
    Partition G' into R and S
    v1 ← a node that when anchored, maximizes the number of nodes saved among all possible placements in R
    v2 ← a node that when anchored, maximizes the number of nodes saved among all possible placements in R, assuming that v1 is anchored
    (v3, v4) ← the nodes on the endpoints of a longest path across all trees in S
    if c(v1) + c(v2) > c(v3, v4) or b = 1 then
        A ← A ∪ {v1}   // If v1 is undefined, pick any unanchored node
        b ← b − 1
    else
        A ← A ∪ {v3, v4}
        b ← b − 2
    end if
end while
return A
```

Figure 6: Anchored 2-core algorithm $TwoStepGreedy(G, b)$.

3. After finding v_1, find another anchor $v_2 \in R$, that maximizes the number of vertices saved in R, assuming that v_1 has already been placed. (Hint: there are two cases to consider: when v_2 is in the same tree as v_1, and when v_2 is in a different tree.)

4. Find a pair of anchors $v_3, v_4 \in S$ that together maximize the number of vertices saved across all placements of two anchors in S.

5. Let $c(v_1)$, $c(v_2)$, $c(v_3, v_4)$ represent the number of nodes "saved" by v_1, v_2, and the pair (v_3, v_4) respectively (including themselves).

   (a) If $c(v_1) + c(v_2) > c(v_3, v_4)$, or $b = 1$, we place an anchor at v_1, and decrement our budget b by 1.

   (b) Otherwise, we place two anchors at v_3 and v_4, and decrement our budget b by 2.

6. We can now add these new anchors and the nodes they save to the k-core of the graph. We then re-run steps 1 to 5 above until our budget $b = 0$.

The pseudocode for this algorithm is provided in Figure 6.

4.7 Data exploration [15 points]

We have provided you with two undirected graphs, $G1$ (`graph1.txt`) and $G2$ (`graph2.txt`), which can be obtained at http://cs224w.stanford.edu/data/hw4q4graphs.zip. Read them in using a network analysis software package of your choice.

```
1: C_k ← the k-core of G
2: N ← C_k ∪ A ∪ {v | v was saved by the anchoring of nodes in A}.
3: for u ∈ N do
4:     for v ∈ N do
5:         if u ≠ v then
6:             Remove the edge (u, v) from G
7:         end if
8:     end for
9: end for
10: return G
```

Figure 7: *RemoveCore*(G, A).

If you are using snap.py, install version 0.8.3 before you attempt this part. You might need to use functions that were not implemented in older versions.

As above, given a budget b, we want to pick nodes in the network, such that the size of the remaining equilibrium graph (r) is maximized.

We are interested in comparing the performance of two different algorithms:

1. The algorithm that has been the main focus of the question, as in Figure 6 (*TwoStepGreedy*)
2. Simply picking the b nodes with the highest degree in the network (*HighestDegree*)

While *HighestDegree* may not always give a good solution, it runs several times faster than *TwoStepGreedy*, and there may be cases where the algorithm performs well. *HighestDegree* simply sorts all nodes that are not part of C_k in descending degree order, and picks the first b nodes, breaking ties randomly.

Implement and run the *TwoStepGreedy* and *HighestDegree* algorithms on $G1$ and $G2$, varying the budget b between 0 and 10 (inclusive). We have also provided pseudo-code for *RemoveCore*(G, A) (Figure 7).

To help in debugging, we have also provided three toy graphs (`toy1.txt`, `toy2.txt`, `toy3.txt`) that you can test your code on. The expected values of r you should be getting for each of these toy graphs are provided in the comments of each text file.

For each graph, plot each algorithm's performance (r) against the provided budget (b). Describe the plots in a few brief sentences.

Generate two plots, one for each graph. Each plot should contain two lines, one for each algorithm. The y-axis is the size of the equilibrium graph (r) if b anchors are picked, and the x-axis is the budget b.

4.8 Possible structures of $G1$ and $G2$ [5 points]

Briefly describe a possible structure of $G1$ that would explain your observations. Do the same for $G2$. This is an open-ended question; well-reasoned structures will receive full credit. You may use diagrams if you wish.

What to submit

4.1) The nodes that remain in the equilibrium graph.

4.2) An algorithm.

4.3) The two nodes. The number of additional nodes saved.

4.4) An explanation.

4.5) A family of examples. The optimal solution and the solution *NaiveGreedy* picks.

4.6) Nothing!

4.7) Print and submit your code online. Two plots. Your brief observations.

4.8) Two graph structure descriptions.

If you're interested, you can read more about anchored k-cores at http://theory.stanford.edu/~klewi/papers/kcore.pdf