Homework 4, Question 4

In [1]:
```
import networkx as nx
from ggplot import *
from collections import *
from itertools import combinations
```

Helper functions

In [2]:
```
EMPTY_SET=set()
```

Helper function to find k-core, given an anchor set (default empty)

In [3]:
```
def FindKCore(graph,k=2,anchor=EMPTY_SET):
    G = graph.copy()
    Q = deque()

    # place nodes with degree < k on the queue
    for n in G.nodes_iter():
        if n not in anchor and nx.degree(G,n) < k:
            Q.append(n)

    # Now iterate until queue is empty
    while len(Q) > 0:
        n = Q.popleft()
        if G.has_node(n) and n not in anchor and nx.degree(G,n) < k: # for nod
es with deg < k
            for nbr in nx.all_neighbors(G,n):
                Q.append(nbr)  # place neighbors on queue
            G.remove_node(n)   # delete node
    return G
```

HighestDegree algorithm

In [4]:
```
def HighestDegree(graph,b,debug=False):
    G = graph.copy()
    degs = G.degree()
    h_nodes = sorted(degs, key=degs.get,reverse=True)  # sort nodes by descend
ing degree
    saved_size = OrderedDict()
    saved_size[0] = FindKCore(graph).number_of_nodes()
    A = set()
    cnt = 0
    for h in h_nodes[:b]:
        cnt += 1
        A.add(h)
        saved_size[cnt] = FindKCore(graph,anchor=A).number_of_nodes()
    return saved_size
```

TwoStepGreedy Algorithm

Partition graph into R and S

In order to partition the graph, we:

- Find the connected components of the graph
- A component containing a k-core (plus saved) node is added to R
- Remaining components are added to S

Also we find a set of candidate anchor nodes, i.e. nodes with degree less than k. For k=2, these are basically leaves of the trees in R and S.

In [5]:
```
def PartitionGraph(graph,kc_nodes,anchor=EMPTY_SET,k=2):
    G = graph.copy()
    R_cand = set()
    S_cand = set()

    G_ccs = nx.connected_component_subgraphs(G)

    for g_cc in G_ccs:
        cc_nodes = set(g_cc.nodes())
        kc_overlap = cc_nodes.intersection(kc_nodes)
        if len(kc_overlap) > 0:
            root = kc_overlap.pop()
            R_nodes = set()
            for n in cc_nodes:
                d = nx.degree(G,n)
                if n not in anchor and n not in kc_nodes and d > 0 and d < k:
                    R_nodes.add(n)
                R_cand = R_cand.union(set((u,root) for u in R_nodes))

        else:
            S_nodes = set()
            for n in cc_nodes:
                d = nx.degree(G,n)
                if n not in anchor and d > 0 and d < k:
                    S_nodes.add(n)
            S_cand = S_cand.union(set((u,v) for u,v in combinations(S_nodes,2)
))

    return R_cand,S_cand
```

Helper function $RemoveCore(G, A)$

```
In [6]: def RemoveCore(graph,k=2,anchor=EMPTY_SET):
            G = graph.copy()
            Gn = FindKCore(G,k,anchor)
            for u,v in Gn.edges_iter():
                if u != v:
                    G.remove_edge(u,v)

            KC_nodes = set(filter(lambda n: nx.degree(G,n) > 0,Gn.nodes()))

            G_small = G.copy()
            for n in G.nodes_iter():
                if nx.degree(G,n) == 0:
                    G_small.remove_node(n)

            return G, G_small,KC_nodes
```

Find the best new potential anchor from sets R or S. If $n = 1$, then use R, else use S. The function also takes the list of current saved nodes.

As suggested in the assignment, there are two cases in R, where v_2 belongs to the same tree as v_1 or otherwise. I have taken care of both cases as follows: Starting from a potential anchor, trace the path towards to root till we hit a k-core or saved node. That is the maximum number of nodes that can be saved by this new anchor. Use that number to find the best anchor. This works for both v_1 and v_2.

In [7]:
```
def FindBestAnchor(graph,C,paths,kc_nodes,n=1,anchor=EMPTY_SET):
    G = graph.copy()
    max_saved = 0
    best_anchor = anchor
    if n == 1:
        best_root = None
        for a,root in C:
            saved = 0
            for n in paths[a][root]:
                if n in kc_nodes or n in anchor: break
                saved += 1

            if saved > max_saved:
                max_saved = saved
                best_anchor = set((a,))
                kc_nodes = kc_nodes.union(set(paths[a][root]))
    if n == 2:
        for c1,c2 in C:
            saved = len(paths[c1][c2]) # Find two leaves furthest from each ot
her in S
            if saved > max_saved:
                max_saved = saved
                best_anchor = set([c1,c2])
                kc_nodes = kc_nodes.union(set(paths[c1][c2]))

    return max_saved,best_anchor,kc_nodes
```

In order to find optimal anchors quickly, we pre-calculate all pairs of shortest paths between nodes in the graph with K-Core removed. This automatically gives us all paths between nodes in the sets R and S.

In [8]:
```
def TwoStepGreedy(graph,b,debug=True):
    G = graph.copy()
    A = set()
    Gp,_,_ = RemoveCore(G)
    paths = nx.all_pairs_shortest_path(Gp)
    cnt = 0
    saved_size = OrderedDict()
    saved_size[0] = FindKCore(G).number_of_nodes()

    while b > 0:
        Gp,Gs,kc_nodes = RemoveCore(G,anchor=A)
        R_cand,S_cand = PartitionGraph(Gs,kc_nodes,anchor=A)
        if debug: print list(R_cand),list(S_cand),
        if len(R_cand) == 0 and len(S_cand) ==0:
            break

        # Find optimal 1st and 2nd anchors from R i.e. v1 and v2
        N1,A1,kc_nodes_R = FindBestAnchor(G,R_cand,paths,kc_nodes)
        R_cand = R_cand.difference(A1)
        N2,A2,kc_nodes_R = FindBestAnchor(G,R_cand,paths,kc_nodes_R,anchor=A1)

        N2 = max(N1,N2)

        # Find optimal pair from S i.e. (v3,v4)
        N3,A3,kc_nodes_S = FindBestAnchor(G,S_cand,paths,kc_nodes,n=2)
        S_cand = S_cand.difference(A3)

        if b==1 or N2 > N3:
            A = A.union(A1)
            b -= 1
            cnt += 1
        else:
            saved_size[cnt+1] = FindKCore(graph,anchor=A.union(A1)).number_of_
nodes()
            A = A.union(A3)
            b -= 2
            cnt += 2
        saved_size[cnt] = FindKCore(graph,anchor=A).number_of_nodes()

    if debug: print "Final anchors:",list(A)
    return saved_size
```

Plots for G1 and G2

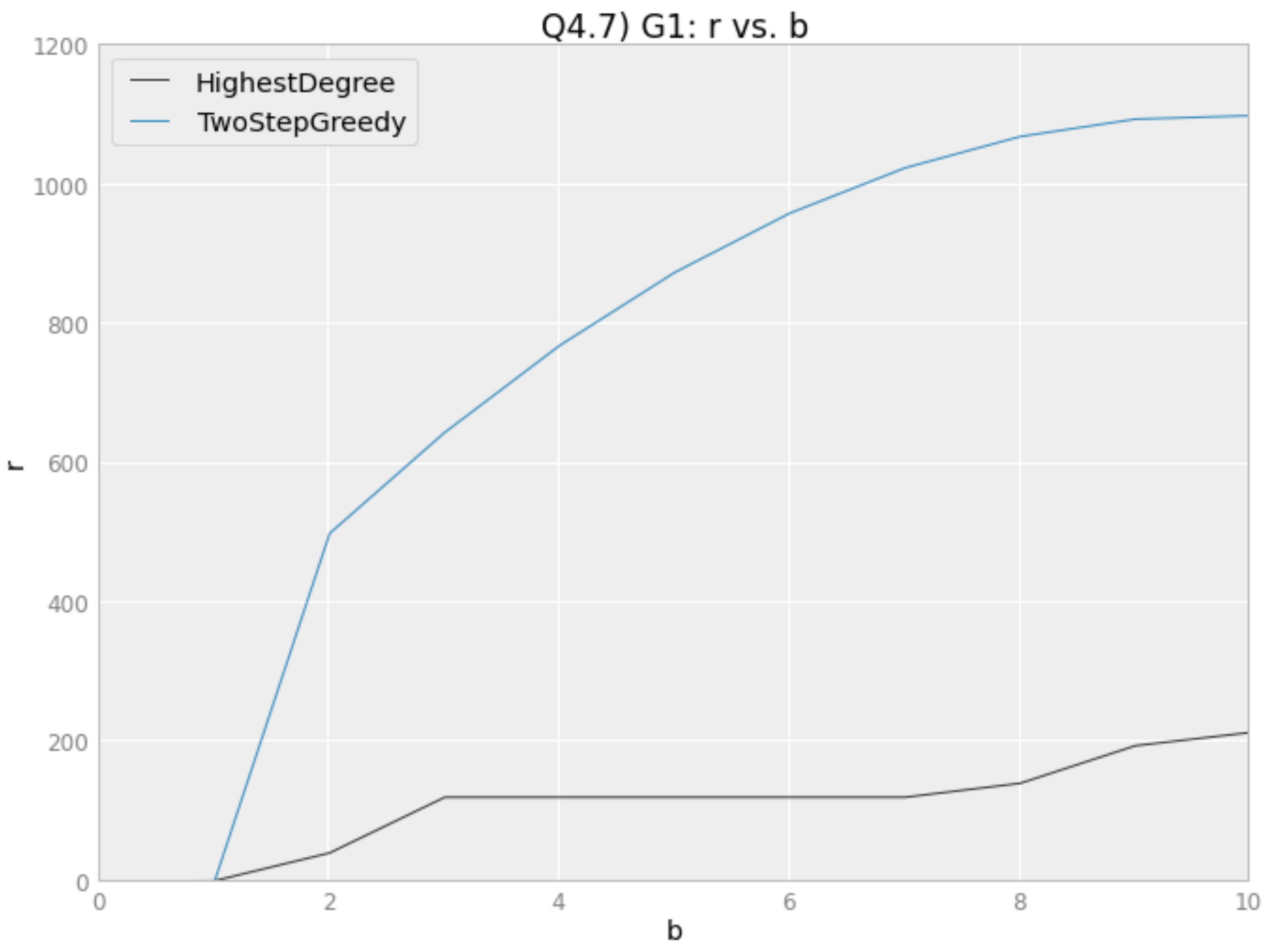
In [9]:
```
G1 = nx.read_edgelist("g1.txt")
G2 = nx.read_edgelist("g2.txt")
```

In [10]:
```
def KCoreAlgorithm(graph,algorithm):
    N=11
    X = arange(N)
    Y=ones(N)
    saved_size = algorithm(graph,N-1,debug=False)
    xt,yt = zip(*saved_size.items())
    Y[list(xt)]=yt
    return X,Y
```

In [11]:
```
xh1,yh1 = KCoreAlgorithm(G1,HighestDegree)
```

In [12]:
```
xg1,yg1 = KCoreAlgorithm(G1,TwoStepGreedy)
```

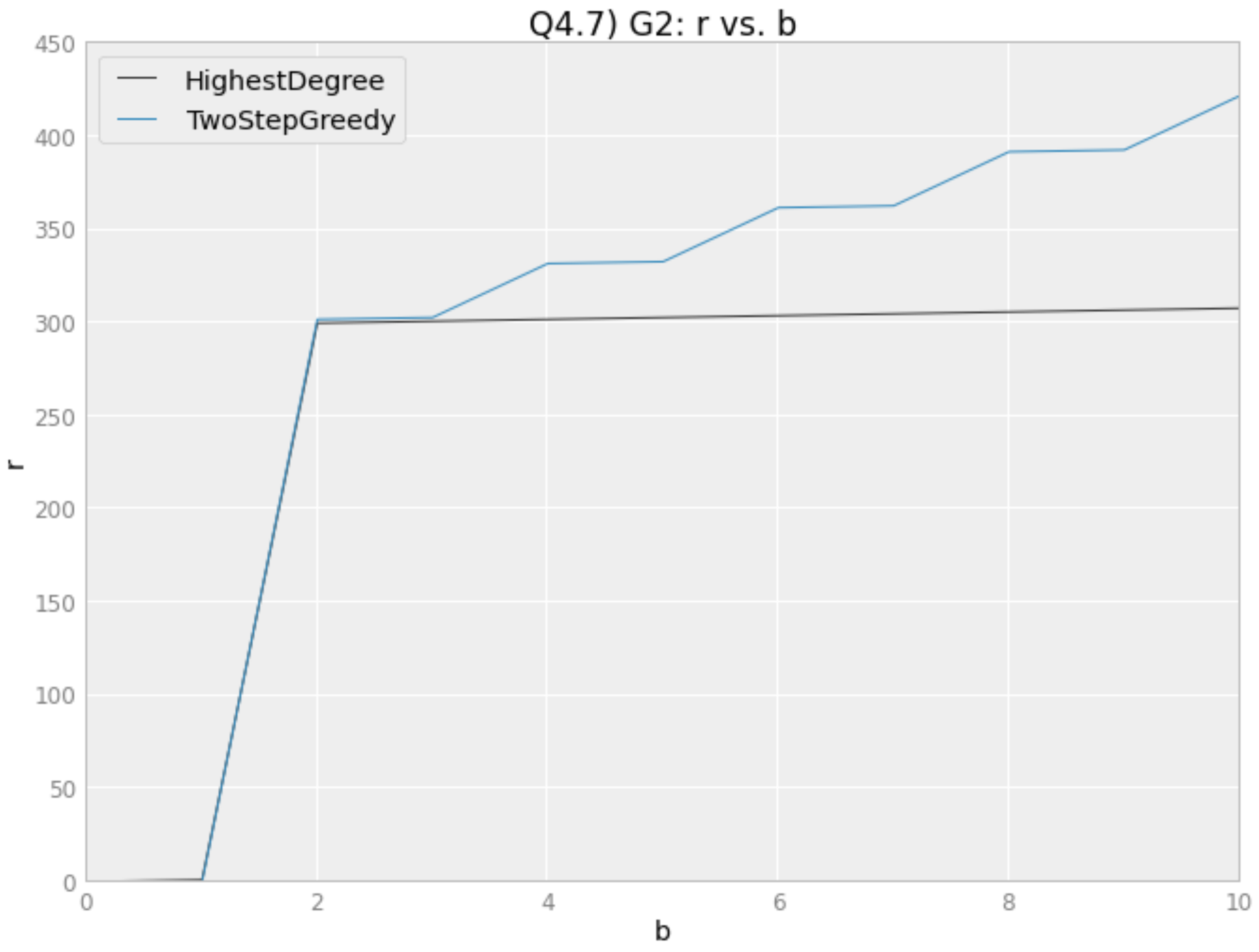
In [17]:
```
plt.plot(xh1,yh1,label="HighestDegree")
plt.plot(xg1,yg1,label="TwoStepGreedy")
plt.xlabel("b")
plt.ylabel("r")
plt.legend(loc="best")
p=plt.title("Q4.7) G1: r vs. b")
```



In [14]:
```
xh2,yh2 = KCoreAlgorithm(G2,HighestDegree)
```

In [15]:
```
xg2,yg2 = KCoreAlgorithm(G2,TwoStepGreedy)
```

In [16]:
```
plt.plot(xh2,yh2,label="HighestDegree")
plt.plot(xg2,yg2,label="TwoStepGreedy")
plt.xlabel("b")
plt.ylabel("r")
plt.legend(loc="best")
p=plt.title("Q4.7) G2: r vs. b")
```



In [16]: