Homework 4, Question 2

In [1]:
```
import networkx as nx
from collections import defaultdict,deque
from random import choice,shuffle
from ggplot import *
```

Create a small test graph based on lecture slides.

In [2]:
```
tg = nx.Graph()
tg.add_edges_from([(0,1),(0,2),(0,3),(0,4),(1,2),(1,5),(2,5),(3,6),(3,7),(4,7)
,(6,8),(5,8),(6,9),(7,9),(8,10),(9,10)])
```

Perform a BFS from a vertex s

In [3]:
```
def BFS(g,v):
    # Book-keeping stuff
    distance = {}
    sigma = defaultdict(float)
    parents = defaultdict(set)
    queue = deque([v])
    done = set()

    distance[v] = 0
    sigma[v] = 1.0

    induced = nx.DiGraph()

    while len(queue) > 0:
        n = queue.popleft()
        done.add(n)
        # Add neighbors of n that are not done to the queue
        for nbr in g.neighbors_iter(n):
            if nbr not in done:
                if nbr in distance and distance[nbr] <= distance[n]:
                    pass
                else:
                    distance[nbr] = distance[n]+1
                    if n not in parents[nbr]:
                        parents[nbr].add(n)
```

```
                        sigma[nbr] += sigma[n]
                        induced.add_edge(n,nbr)
                    queue.append(nbr)
    return sigma,induced
```

Now that we have the induced tree, calculate dependency (δ_s) of an edge $\{v, w\}$ recursively.

In [4]:
```
def dependency(v,w,induced,sigma):
    if induced.has_edge(v,w):
        d = sigma[v]/sigma[w]
        if len(induced.neighbors(w)) > 0:
            return d*(1+sum([dependency(w,c,induced,sigma) for c in induced.ne
ighbors(w)]))
        else:
            return d
    else:
        return 0.0
```

In [5]:
```
def dependencies(graph):
    dependencies = defaultdict(list)
    for src in graph.nodes_iter():
        sigma,induced = BFS(graph,src)
        for v,w in graph.edges_iter():
            dependencies[src].append((v,w,dependency(v,w,induced,sigma)))
    return dependencies
```

Algorithm 1: Exact Betweenness

In [6]:
```
def exact_betweenness(graph):
    betweenness = defaultdict(float)
    depends = dependencies(graph)
    for src in depends:
        for deps in depends[src]:
            v,w,d = deps
            betweenness[(v,w)] += d
    return dict(betweenness)
```

For verification, compare between above code and networkx

In [7]:
```
exact_betweenness(tg)
```

In [7]: `exact_betweenness(tg)`

Out[7]:
```
{(0, 1): 7.3857142857142852,
 (0, 2): 7.3857142857142852,
 (0, 3): 10.752380952380953,
 (0, 4): 8.019047619047619,
 (1, 2): 1.0,
 (1, 5): 6.1857142857142859,
 (2, 5): 6.1857142857142859,
 (3, 6): 8.8761904761904766,
 (3, 7): 6.4952380952380944,
 (4, 7): 7.352380952380952,
 (5, 8): 11.895238095238096,
 (6, 8): 8.519047619047619,
 (6, 9): 6.1095238095238091,
 (7, 9): 8.7523809523809533,
 (8, 10): 7.3761904761904766,
 (9, 10): 6.7095238095238088}
```

In [8]:
```
nx.edge_betweenness_centrality(tg,normalized=False,weight=1)
```

Out[8]:
```
{(0, 1): 7.385714285714284,
 (0, 2): 7.385714285714284,
 (0, 3): 10.752380952380953,
 (0, 4): 8.01904761904762,
 (1, 2): 1.0,
 (1, 5): 6.185714285714286,
 (2, 5): 6.185714285714286,
 (3, 6): 8.876190476190477,
 (3, 7): 6.495238095238095,
 (4, 7): 7.352380952380952,
 (5, 8): 11.895238095238096,
 (6, 8): 8.519047619047619,
 (6, 9): 6.109523809523809,
 (7, 9): 8.752380952380951,
 (8, 10): 7.376190476190477,
 (9, 10): 6.7095238095238106}
```

Algorithm 2: Approximate Betweennes

In [9]:
```
def approx_betweenness(graph,c=5,s=10):
```

```
    nodes = graph.nodes()
    shuffle(nodes)
    N = float(len(nodes))
    max_D = c*N
    max_samples = int(N/s)

    betweenness = defaultdict(float)
    k = defaultdict(int)

    n_samples = 0
    for src in nodes[:max_samples]:
        sigma,induced = BFS(graph,src)
        change = False
        for v,w in graph.edges_iter():
            if betweenness[(v,w)] <= max_D:
                betweenness[(v,w)] += dependency(v,w,induced,sigma)
                k[(v,w)] += 1
                change = True
        if not change:break

    return {edge:((N/k[edge])*betweenness[edge]) for edge in graph.edges_iter(
)}
```

Run on preferential attachment graph

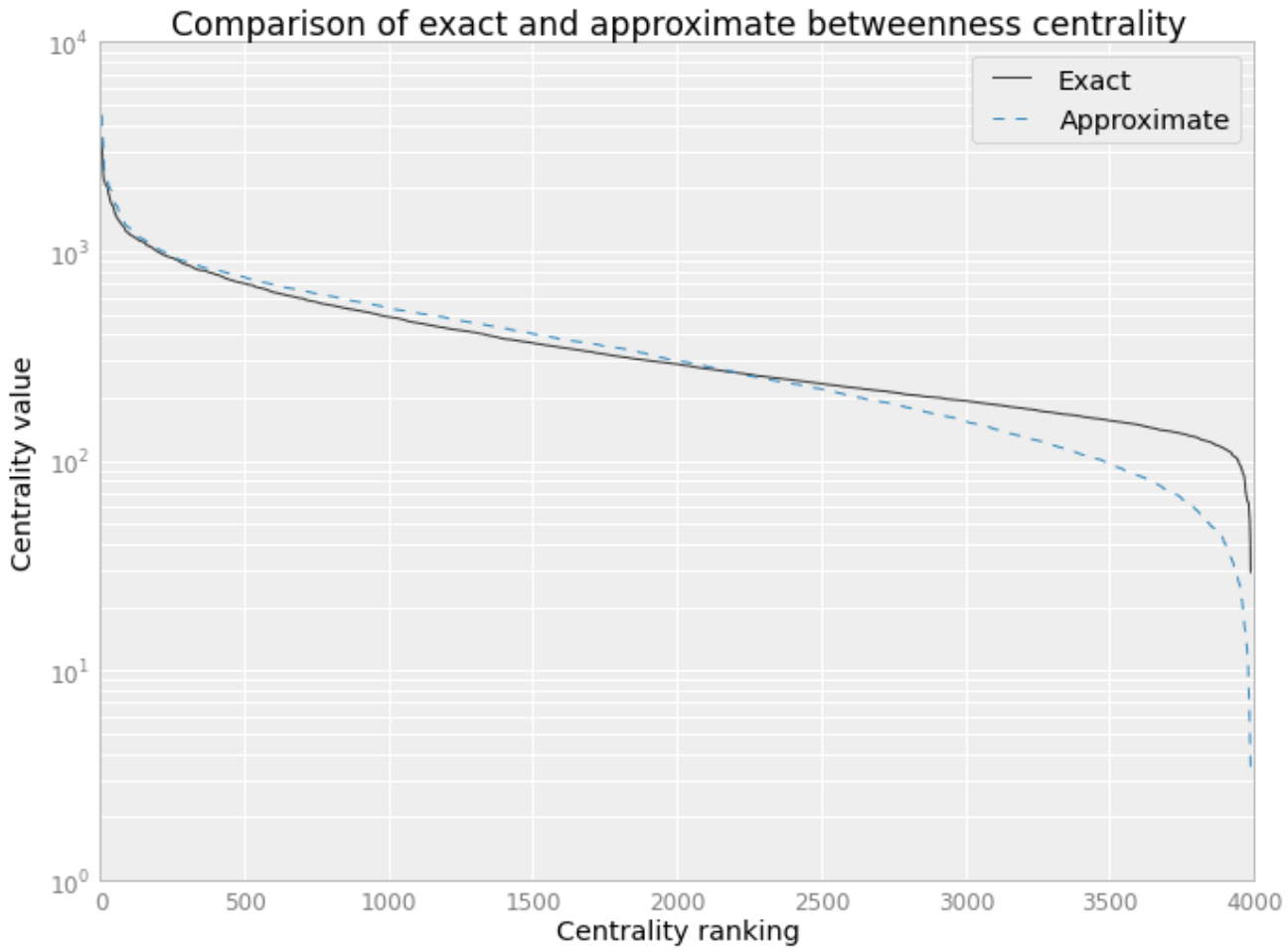
In [10]:
```
g = nx.barabasi_albert_graph(1000,4)
```

In [11]:
```
exact_b = exact_betweenness(g)
approx_b = approx_betweenness(g)
```

In [12]:
```
x = range(len(exact_b))
y1 = sorted(exact_b.values(), reverse=True)
y2 = sorted(approx_b.values(), reverse=True)
```

In [14]:
```
plt.plot(x,y1,label="Exact")
plt.plot(x,y2,'--',label="Approximate")
plt.yscale('log')
plt.legend()
plt.title("Comparison of exact and approximate betweenness centrality")
```

```
plt.xlabel("Centrality ranking")
p=plt.ylabel("Centrality value")
```



In [ ]: