simulations

June 24, 2017

1 Selfish Routing with Positive Externalities

1.1 Introduction

We consider random networks in which multiple agents with heterogeneous sources and destinations determine their own routes as a best reply to others best replies, the nash equilibrium, that is obtained from an initial state where all agents choose their route on an empty network.

There are two primary questions we are interested in learning in the simulations. The firt being, how bad can a graph in terms of the social cost of the Waldrop equilibrium of the full graph versus that of its optimal subgraph. The second is whether or not a greedy algorithm can approximate the results of the exhaustive search, which would indicate that this problem might be submodular in the edges removed.

These simulations are computationally expensive, as an exhaustive search is needed to prove the best possible subgraph, however even these crude initial explorations serve as extremely informative regarding these two primary questions in better understanding the nature of this problem.

1.2 Setup & Background

We define some core concepts related to the questions we are exploring in the simulations:

1.2.1 Wardrop Equilibrium

We first define a Wardrop equilibrium, which simply says that given a certain flow f, and cost function along a path $c_{\mathcal{P}} : \mathcal{P} \to \mathcal{R}^{+}$, the cost of every equilibrium path is a equal, and less than the cost along any non-equilibrium path:

$$c_{\mathcal{P}}(f) \leq c_{\widetilde{\mathcal{P}}}(f)$$

1.2.2 Social Cost

Given multiple sources S, multiple agents A, a shared destination t, and convex and non-decreasing cost function $c(e, f)$, shared up to the paramaterized constant e by each edge (which we can consider to be the distance between two edges in some sort of measure external to the traffic on the network, such as the physical distance of a road), the social cost, Γ, for a given graph G, at Wardrop equilibrium is given by:

$$C(G) = \sum_{a \in A} d(S_a, t)$$

Where distance function d(s,t) is defined as the distance along the shortest path, $p(s,t)$, between s and t:

$$d(s,t) = \sum_{e \in p(s,t)} c(e,f)$$

1.2.3 Selfish Ratio

Now we can define Selfish Ratio, namely the ratio between the Wardrop equilibrium of the full network, to that of its optimal subnetwork:

$$\beta(G,r,c) = \max_{H \subset G} \frac{C(f)}{C(f^H)}$$

1.3 Code for Simulation

All code can be run via a Jupyter Notebook via the official Jupyter Docker stacks, scipy-notebook. The code can be found in the /lib folder of the same repository as this notebook. Note that it is parallelized and best run on a machine with many cores, the current simulations take ~2 hours on a machine with 32 cores.

```
In [ ]: !conda uninstall --yes networkx
        !pip install --upgrade git+git://github.com/networkx/networkx.git#egg=networkx
        !pip install ggplot
```

```
In [29]: import networkx as nx
         import seaborn as sns
         import matplotlib.pyplot as plt
         import pandas as pd
         from ggplot import *

         # Our code
         from lib.equilibrium import *
         from lib.generation import *
```

```
In [4]: def random_distances(graph, bottom = 0.5, top = 2, mult = 1):
            n = graph.shape[0]
            rands = (np.random.uniform(bottom,top,n**2).reshape((n,n))*mult)
            return graph * rands

        def base_setup(edges, n, A, cost_fn):
            agents = random_placements(n, A)
            graph = random_distances(path_to_mat(edges, n))
            edges = list(nx.DiGraph(graph).edges(data=True))
            state = initial_state(agents, graph)
            new_state = find_equilibrium(state, graph, cost_fn)
            graphs = [path_to_mat(c, n) for c in combinations(edges)]
            return new_state, graph, graphs, cost_fn
```

```
def barabasi(n, attch):
    edges = nx.barabasi_albert_graph(n, attch).edges()
    return list(edges) + map(lambda t: (t[1],t[0]), edges)

def erdos(n, p):
    return list(nx.erdos_renyi_graph(n, p, directed = True).edges())

def full(n):
    adj = np.ones((n,n))
    return list(nx.DiGraph(adj).edges())

def run_one(state, full_graph, graphs, cost_fn, iters = 100):
    return [
        score_graph(state, full_graph, cost_fn),
        find_optimum_subgraph(state, full_graph, cost_fn, iters)[0],
        combinatorial_optimum(state, graphs, cost_fn)[0],
        float((full_graph != 0).sum())/(full_graph.shape[0]**2) # sparsity
    ]

def run(m, n, A, cost_fn, setup, args):
    def run_model():
        try:
            return run_one(*base_setup(setup(n, *args), n, A, cost_fn))
        except nx.exception.NetworkXNoPath:
            return None

    res = [run_model() for _ in range(m)]
    res = [r for r in res if r is not None]
    df = pd.DataFrame(np.array(res),
                      columns = ["original", "greedy", "exhaustive", "sparsity"])
    return df.assign(generator = setup.__name__,
                     generator_param = args[0],
                     cost = cost_fn.__name__)
```

1.3.1 Cost Functions

We examine two cost functions, one linear, one exponential, both convex and decreasing in their parametarized flow.

```
In [5]: def linear_cost(f,c):
            return c * 1/f if f > 0 else float('inf')

        def exp_cost(f,c):
            return c * (.5)**(f - 1)
```

2 Simulations

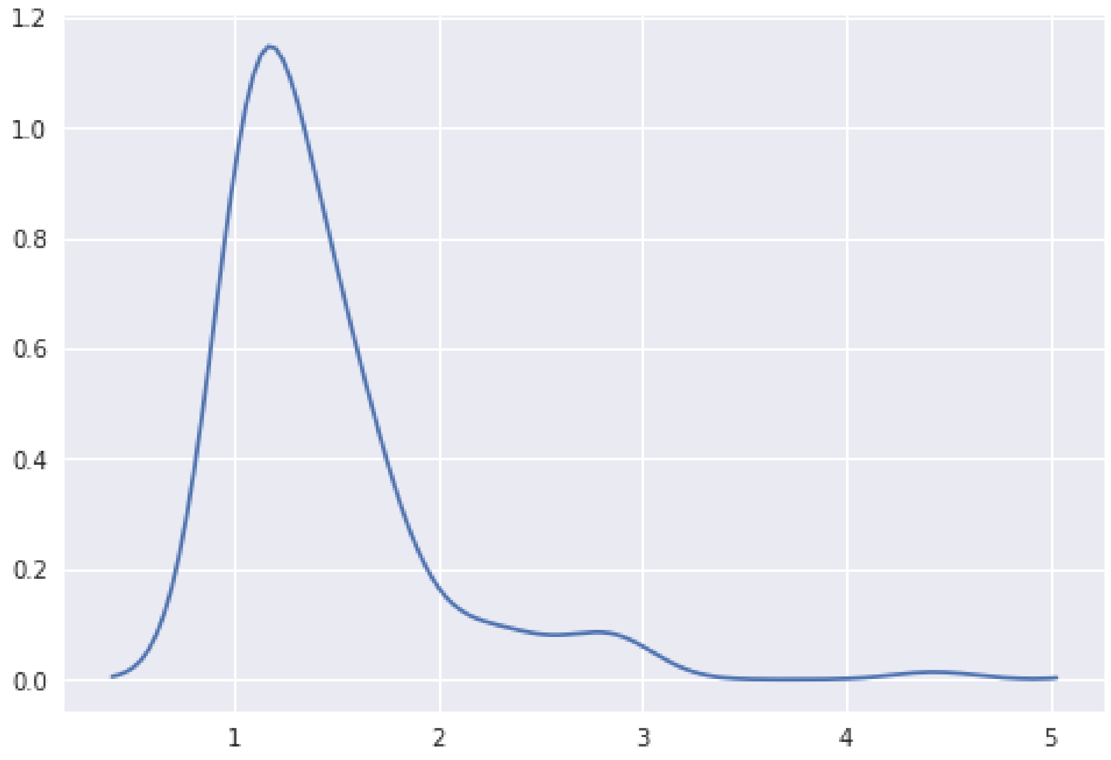
5 nodes, 8 agents, random networks generated via Barabasi-Albert and Erdos Renyi with various parameters. We run 20 simulations with each set of parameters.

```
In [ ]: df = pd.concat([
            run(20, 5, 8, linear_cost, erdos, args = [.8]),
            run(20, 5, 8, exp_cost, erdos, args = [.8]),
            run(20, 5, 8, linear_cost, barabasi, args = [3]),
            run(20, 5, 8, exp_cost, barabasi, args = [3]),
            run(20, 5, 8, linear_cost, erdos, args = [.5]),
            run(20, 5, 8, exp_cost, erdos, args = [.5]),
            run(20, 5, 8, linear_cost, barabasi, args = [2]),
            run(20, 5, 8, exp_cost, barabasi, args = [2])
            ], ignore_index = True)
```

2.1 Selfish Ratio

These initial simulations don't allow us to bound the Selfish Ratio, but they do show that there is very often some inefficiency in the network, but that inefficiency is rarely very large.

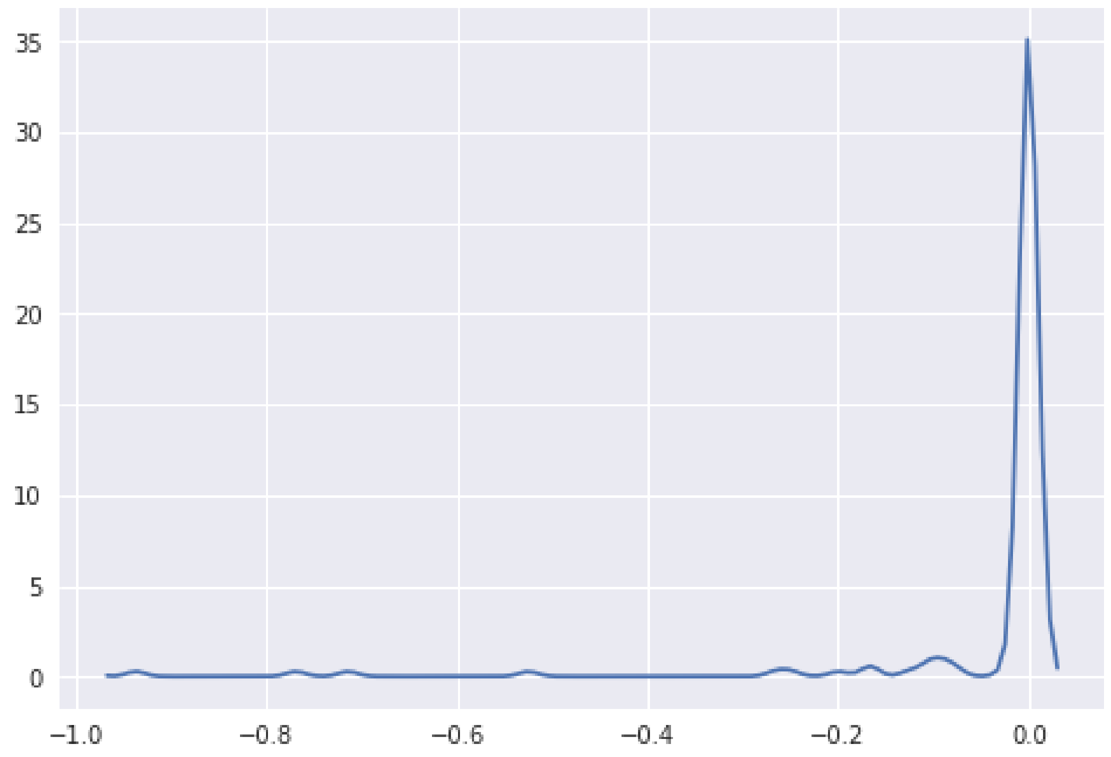
```
In [38]: ratio = df.original/df.exhaustive
         p = sns.kdeplot(ratio, bw = .2)
```



2.2 ϵ-Approximation Error of Greedy Algorithm

From the results of this initial simulation, it is clear that we can't necessarily bound the lower end of the Greedy algorithm, it is not clear that it cannot be arbitrarily bad given a particular network structure. We can, however, see clearly that in these particular networks tested, it finds a perfect solution a large percentage of the time, which is also very interesting.

```
In [39]: epsilon = (df.exhaustive - df.greedy)/df.exhaustive
         p = sns.kdeplot(epsilon, bw = .01)
```



```
In [40]: np.min(epsilon)
```

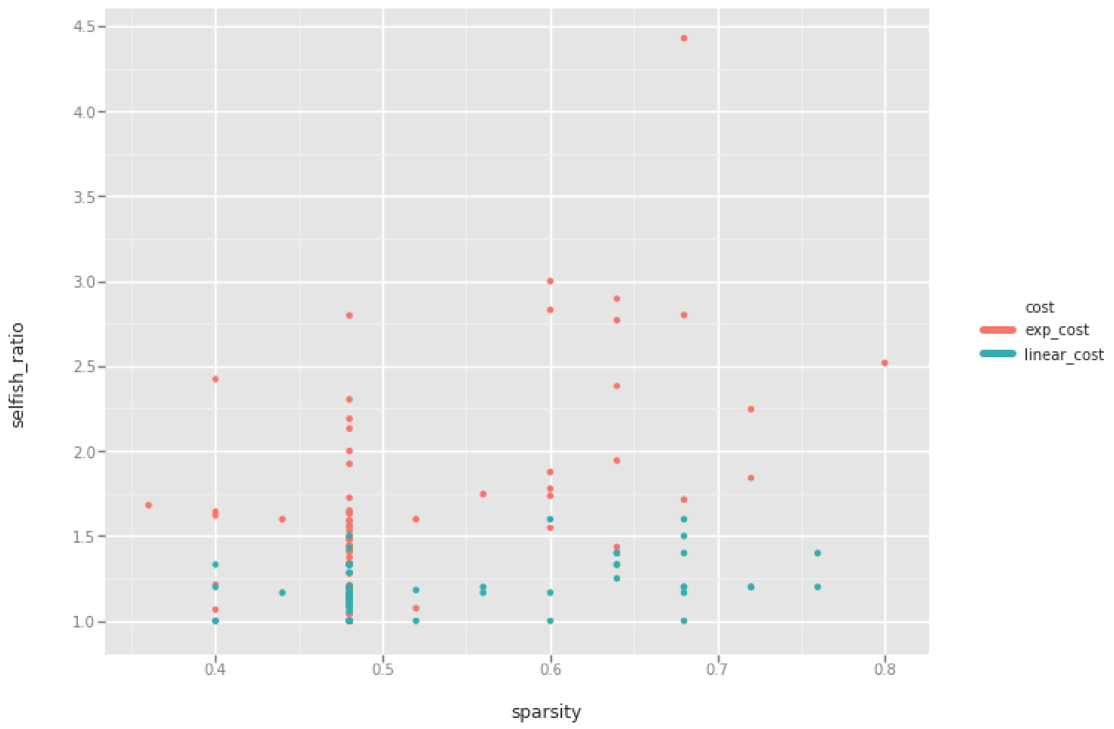
```
Out[40]: -0.9375
```

2.3 Selfish Ratio as a Function of Sparsity

We will need more simulations to conclude anything with regards to this relationship. Theory points us to the selfish ratio being worse the more full the graph is, however we do not see evidence of that in these limited simulations.

```
In [41]: df = df.assign(selfish_ratio = df.original/df.exhaustive)

         p = ggplot(df, aes(x = "sparsity", y = "selfish_ratio", color = "cost")) + geom_point()
```



```
Out[41]: <ggplot: (8742600837113)>
```

2.4 Raw Results

```
In [42]: df
```

```
Out[42]:      original     greedy  exhaustive  sparsity         cost generator  \
         0      6.0000   5.000000    5.000000      0.72  linear_cost     erdos
         1      8.0000   6.000000    6.000000      0.64  linear_cost     erdos
         2      5.0000   5.000000    5.000000      0.68  linear_cost     erdos
         3      9.0000   6.000000    6.000000      0.68  linear_cost     erdos
         4      8.0000   6.000000    5.000000      0.68  linear_cost     erdos
         5      6.0000   5.000000    5.000000      0.68  linear_cost     erdos
         6      7.0000   5.000000    5.000000      0.64  linear_cost     erdos
         7      6.0000   5.000000    5.000000      0.76  linear_cost     erdos
         8      7.0000   5.000000    5.000000      0.64  linear_cost     erdos
         9      6.0000   5.000000    5.000000      0.68  linear_cost     erdos
         10     7.0000   6.000000    6.000000      0.56  linear_cost     erdos
         11     8.0000   5.000000    5.000000      0.60  linear_cost     erdos
         12     5.0000   5.000000    5.000000      0.60  linear_cost     erdos
         13     6.0000   6.000000    6.000000      0.52  linear_cost     erdos
         14     7.0000   6.000000    6.000000      0.60  linear_cost     erdos
         15     7.0000   6.000000    6.000000      0.68  linear_cost     erdos
         16     8.0000   6.000000    6.000000      0.64  linear_cost     erdos
```

```
17     7.0000   5.000000   5.000000        0.68  linear_cost     erdos
18     6.0000   5.000000   5.000000        0.72  linear_cost     erdos
19     7.0000   5.000000   5.000000        0.76  linear_cost     erdos
20     7.3125   4.187500   4.187500        0.56     exp_cost     erdos
21    18.0000  11.625000   6.000000        0.60     exp_cost     erdos
22     4.0000   2.500000   2.500000        0.52     exp_cost     erdos
23     6.0000   3.937500   3.500000        0.68     exp_cost     erdos
24     5.5000   3.812500   3.812500        0.48     exp_cost     erdos
25     7.7500   3.000000   1.750000        0.68     exp_cost     erdos
26     4.0000   3.312500   2.171875        0.72     exp_cost     erdos
27     6.5000   2.296875   2.296875        0.60     exp_cost     erdos
28     4.5625   2.031250   2.031250        0.72     exp_cost     erdos
29     5.3125   2.109375   2.109375        0.80     exp_cost     erdos
..        ...        ...        ...         ...          ...       ...
114    8.0000   8.000000   8.000000        0.48  linear_cost  barabasi
115    5.0000   5.000000   5.000000        0.48  linear_cost  barabasi
116    6.0000   5.000000   5.000000        0.48  linear_cost  barabasi
117   11.0000  11.000000  10.000000        0.48  linear_cost  barabasi
118    4.0000   4.000000   4.000000        0.48  linear_cost  barabasi
119   12.0000  10.000000  10.000000        0.48  linear_cost  barabasi
120    6.0000   5.000000   5.000000        0.48  linear_cost  barabasi
121    9.0000   8.000000   8.000000        0.48  linear_cost  barabasi
122    5.0000   5.000000   5.000000        0.48  linear_cost  barabasi
123   12.0000  12.000000  11.000000        0.48  linear_cost  barabasi
124    7.7500   5.000000   5.000000        0.48     exp_cost  barabasi
125    6.2500   2.234375   2.234375        0.48     exp_cost  barabasi
126   12.5000  10.312500  10.312500        0.48     exp_cost  barabasi
127    4.3125   2.500000   2.500000        0.48     exp_cost  barabasi
128    6.5000   6.500000   6.500000        0.48     exp_cost  barabasi
129    9.2500   8.250000   8.250000        0.48     exp_cost  barabasi
130    6.8125   4.562500   4.562500        0.48     exp_cost  barabasi
131    3.7500   2.296875   2.296875        0.48     exp_cost  barabasi
132    5.5000   3.812500   3.812500        0.48     exp_cost  barabasi
133    6.2500   6.000000   6.000000        0.48     exp_cost  barabasi
134    4.8125   4.812500   4.812500        0.48     exp_cost  barabasi
135    6.7500   4.562500   4.562500        0.48     exp_cost  barabasi
136    4.1875   4.000000   3.609375        0.48     exp_cost  barabasi
137   16.0000  14.312500  14.312500        0.48     exp_cost  barabasi
138   13.0000   8.312500   8.312500        0.48     exp_cost  barabasi
139    8.7500   4.375000   4.375000        0.48     exp_cost  barabasi
140    5.7500   2.625000   2.625000        0.48     exp_cost  barabasi
141    8.2500   6.000000   6.000000        0.48     exp_cost  barabasi
142    8.5000   5.562500   5.562500        0.48     exp_cost  barabasi
143   10.7500   6.750000   6.750000        0.48     exp_cost  barabasi

     generator_param  selfish_ratio
0                0.8       1.200000
1                0.8       1.333333
```

```
2                 0.8       1.000000
3                 0.8       1.500000
4                 0.8       1.600000
5                 0.8       1.200000
6                 0.8       1.400000
7                 0.8       1.200000
8                 0.8       1.400000
9                 0.8       1.200000
10                0.8       1.166667
11                0.8       1.600000
12                0.8       1.000000
13                0.8       1.000000
14                0.8       1.166667
15                0.8       1.166667
16                0.8       1.333333
17                0.8       1.400000
18                0.8       1.200000
19                0.8       1.400000
20                0.8       1.746269
21                0.8       3.000000
22                0.8       1.600000
23                0.8       1.714286
24                0.8       1.442623
25                0.8       4.428571
26                0.8       1.841727
27                0.8       2.829932
28                0.8       2.246154
29                0.8       2.518519
..                ...            ...
114               2.0       1.000000
115               2.0       1.000000
116               2.0       1.200000
117               2.0       1.100000
118               2.0       1.000000
119               2.0       1.200000
120               2.0       1.200000
121               2.0       1.125000
122               2.0       1.000000
123               2.0       1.090909
124               2.0       1.550000
125               2.0       2.797203
126               2.0       1.212121
127               2.0       1.725000
128               2.0       1.000000
129               2.0       1.121212
130               2.0       1.493151
131               2.0       1.632653
132               2.0       1.442623
```

```
133               2.0       1.041667
134               2.0       1.000000
135               2.0       1.479452
136               2.0       1.160173
137               2.0       1.117904
138               2.0       1.563910
139               2.0       2.000000
140               2.0       2.190476
141               2.0       1.375000
142               2.0       1.528090
143               2.0       1.592593

[144 rows x 8 columns]
```