Networks and Markets - Homework Assignment 1

Moises Baly

September 28, 2016

*Collaborated with Abhishikth Nandam an536@cornell.edu

Game Theory

Kleinberg and Eisley, exercise 6.11.1

The claim is true. If player A has a dominant strategy, it will always play that strategy. That means that player B will have no choice but to play the strategy that maximizes its benefit - best response. Neither player will want to deviate, since it would reduce their payoff (PNE).

Kleinberg and Eisley, exercise 6.11.3

The PNE can be found in node (D, L), with utilities $(2, 4)$. Best response dynamics ensure that if both players are rational, they will try to maximize their utility. Player B will try to maximize its profit playing L, for which Player A will best response is to play D.

Kleinberg and Eisley, exercise 6.11.4

(a)

Player B has a dominant strategy. It is always more profitable to play L, for which the player will obtain an utility of 3, beating every other possible outcome.

(b)

Since Player B has a dominant strategy and assuming rationality, the play will always be L. Since Player A will want to maximize its utility, it will have no choice but to play b, making (b, L) the only PNE in the game.

Kleinberg and Eisley, exercise 6.11.12

(a)

The PNE are $\{(U, L), (D, R)\}$. For player B, L is a weakly dominated strategy, given that there is another strategy at least as large no matter what Player A does (R). Also, there is a strategy for player A - (D) - that produces a greater utility for player B.

(b)

For Player B there are two choices, assuming it has no knowledge of A. It can either choose R with a potential payout of 2 or 1 if things go wrong; or, it can choose to play L with a potential payout of 1 or 0 if things go wrong. Assuming Player B is rational, the only play that makes sense is to go for R.

Graph Theory

2.5.1

(a)

In the graph image below, every node is pivotal:

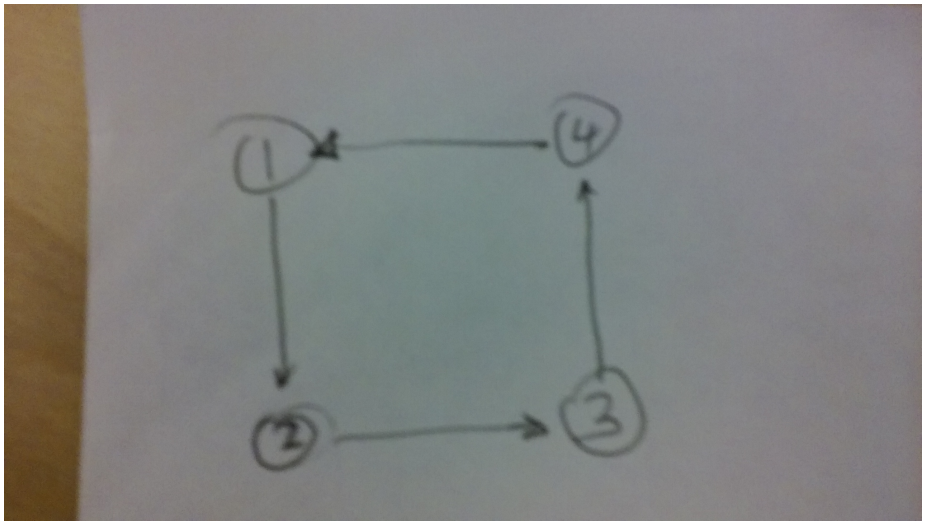

Figure 1: Pivotal nodes graph

Every shortest path between an arbitrary pair of nodes is unique, therefore making nodes in that path pivotal. For example the only shortest path between nodes 1 and 4 goes through nodes 2 and 3, making them pivotal. The same applies to every possible pair.

(b)

Same graph as previous section. For example, 1 is pivotal to shortest paths (4,2) and (3,2).

(c)

The graph below meets the restrictions of the problem:

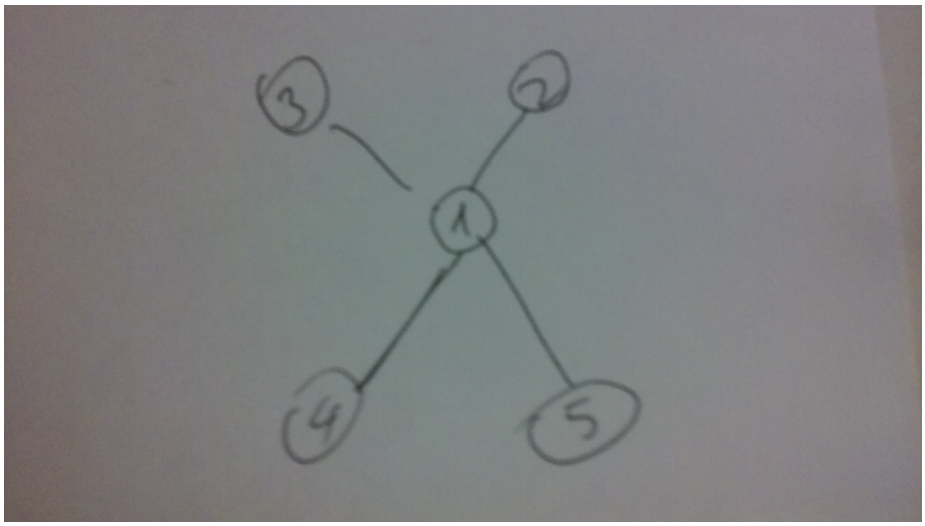

Figure 2: Unique pivotal node

Node 1 is in the shortest path of every other pair of nodes, and each one of those paths is unique.

2.5.3

In the example graph below, the diameter is 4. The average shortest path of graphs with this structure is given by the formula (generalized for n nodes in the complete graph component):

$$\frac{\frac{n*(n-1)}{2}+4*(n-1)+3*(n-1)+2*(n-1)+(1+2+3)+(1+2)+(1)}{\frac{(n+3)*(n+2)}{2}}$$

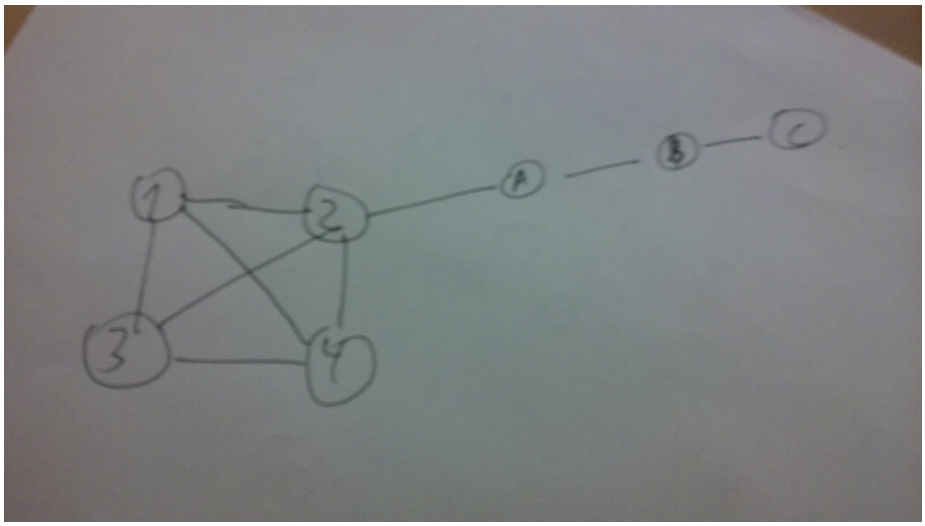

Figure 3: Complete graph with 3-node tail

Let the formula be represented by F. Then, we are looking for:

$\frac{4}{F} \geq 3$

After solving the above for n, we obtain that one of the roots is $n \geq 31$. So, for any complete component of the above graph, if the number of nodes is greater than 31, the diameter/average restriction will hold.

4.6.1

A phenomenon observed in social networks is that if two people many friends in common, they are more likely to meet each other and form a friendship. In the graph, a, c and e have friends in common (b, d). It likely then that a, c and c, e will connect. Same principle applies to b, d.

13.6.1

The largest SCC is composed of nodes $\{1, 3, 4, 8, 9, 13, 14, 15, 18\}$
The IN nodes are $\{6, 7, 11, 12\}$
The OUT nodes are $\{5, 10, 16\}$
The tendril nodes are $\{2, 17\}$

13.6.2

(a)

If we add edge $(12, 13)$, making that edge undirected, we would add node 12 to the SCC.

(b)

We would need to use one of the tendril nodes. If we add edge $(2, 6)$, making that edge undirected, we would add node 2 to the IN.

(c)

We would need to use one of the tendril nodes. If we add edge $(16, 17)$, making that edge undirected, we would add node 17 to the OUT.

Shortest Paths

We need to start with some infrastructure code.

We define a graph node.

```
public class Node {

        // Defines an ID for this node
        int ID;

        //      Defines the weight of the incoming edge
```

```
        double w;

        public Node(int ID, double w) {
                this.ID = ID;
                this.w = w;
        }
}
```

We then define a Graph, using an adjacency list implementation.

```
public class Graph {

        //Defines a graph as an adjacency list.
        private ArrayList<Node>[] nodes;
        private int N;

        public Graph(int capacity) {
                nodes = new ArrayList[capacity];
                for (int i = 0; i < capacity; i++) {
                        nodes[i] = new ArrayList<>();
                }
                N = capacity;
        }

        /**
         * Return the nodes adjacent to Node i
         *
         * @param i
         * @return
         */
        public ArrayList<Node> adjacents(int i) {
                return nodes[i];
        }

        /**
         * Connects node i  > j
         *
         * @param i
         * @param j
         * @param w
         */
        public void connect(int i, int j, double w) {
                ArrayList<Node> adj = adjacents(i);
                adj.add(new Node(j, w));
        }

        /**
         * Connects node i  > j and j > i
         *
         * @param i
         * @param j
```

```
     * @param w
     */
    public void connectUnd(int i, int j, double w) {
            ArrayList<Node> adj = adjacents(i);
            adj.add(new Node(j, w));
            nodes[i] = adj;

            adj = adjacents(j);
            adj.add(new Node(i, w));
            nodes[j] = adj;
    }

    public int size() {
            return N;
    }


    private String printAdj(int i) {
            ArrayList<Node> adj = adjacents(i);
            StringBuffer str = new StringBuffer();
            for (Node n : adj) {
                    str.append("(" + n.ID + ", " + n.w + ")   >");
            }
            return str.toString();
    }

    public void print() {
            for (int i = 0; i < N; i++) {
                    System.out.println(i + " : " + printAdj(i));
            }
    }
```

We can then jump into the exercise. The *BFS* function will look like this:

```
public double bfs(int i, int j) {
        boolean[] visited = new boolean[N];
        for (int k = 0; k < N; k++) {
                    double dist = _bfs(i, j, visited);
                    if (dist != -1) return dist;
            }
            return -1;
    }

private double _bfs(int i, int j, boolean[] visited) {
        if (i == j) return 0;
        Queue<Integer> q = new LinkedList<>();
        double[] distances = new double[N];
        q.add(i);
        while (!q.isEmpty()) {
            int x = q.poll();
            visited[x] = true;
```

```
                for (Node n : adjacents(x)) {
                        if (!visited[n.ID]) {
                                distances[n.ID] += (distances[x] + n.w);
                                visited[n.ID] = true;
                                if (n.ID == j) return distances[n.ID];
                                q.add(n.ID);
                        }
                }
        }
        return  1;
    }
```

Note we have a wrapper function around the actual BFS implementation.
The function to build a Graph, connecting edges based on a probability p:

```
public Graph connectWithP(int size, double p) {
        Random r = new Random(System.currentTimeMillis());
    Graph g = new Graph(size);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
                //Lets avoid self loops
            if (i != j) {
                //If p is very high, we want to connect most of the times.
                if (r.nextDouble() <= p)
                        g.connect(i, j, 1);
                                }
                        }
                }
        return g;
        }
```

The average shortest path function uses our Graph infrastructure, BFS and the previous function.

```
public double avgShortestPath(Graph g, int times, boolean debug) {
        int N = g.size();
    double acc = 0;
    Random r = new Random(System.currentTimeMillis());
    for (int t = 0; t < times; t++) {
        int i = r.nextInt(N);
        int j = r.nextInt(N);
        double d = g.bfs(i, j);
        if (debug) {
                appendToFile(i, j, d);
        }
        // In case that nodes are not connected, ignore them.
        if (d !=  1)
                acc += d;
                }
                if (debug) appendToFile("Average shortest path: " +
        String.format("%.6f", acc / times));
                return acc / times;
```

```
        }
```

Note there is some extra code for debugging purposes in that function.

We now dive into section (C), running the avg. shortest path in $p = 0.1$ connected graph a large number of times.

```
public double runC() {
        Graph g = connectWithP(1000, 0.1);
    return avgShortestPath(g, 30000, true);
}
```

You can see the output of this function in file *outC.txt*, which is a debug of the shortest path of every pair of nodes ran, plus the distance between them. We have an avg. distance of Average shortest path: 1.897800. It means that with $p = 0.1$ our avg. distance will converge to approximately 2.

We now dive into section (D), running the avg. shortest path while trying different values of p.

```
public void runD() {
        double p = 0;
    while (p <= 0.5) {
        Graph g = connectWithP(1000, p);
        double avg = avgShortestPath(g, 10000, false);
        appendToFile(p, avg);
        p += 0.01;
    }
}
```

You can see the output of this function in file *outD.txt*.

If we plot the avg distance in function of p, we get a graph like:

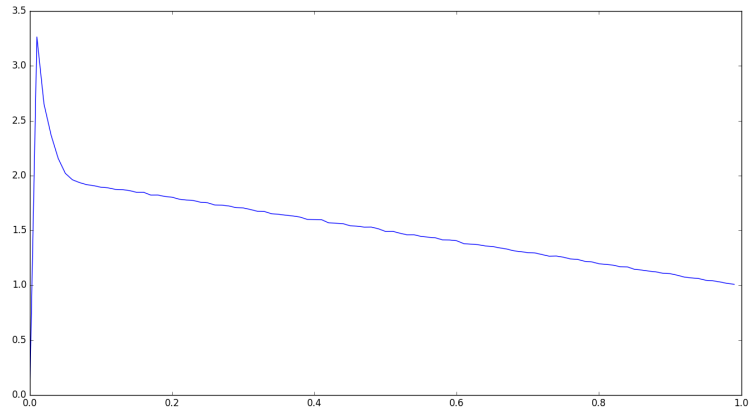

Figure 4: Avg. distance in function of p

We can see a spike in avg distance when p is very small, which can probably be explained by very few connections contributing the global average. We can see a steady convergence towards 1 as the graph becomes complete.