Homework Assignment Week 8: Linear Programming and Network Flow

Name : Woonki Kim

Email : Kimwoon@oregonstate.edu

1. Implement and evaluate Ford-Fulkerson algorithm for maximum flow. The input is a directed graph with edge capacities, a source node and a sink node. The output is the maximum flow and an assignment of flows to edges. Then, you will compare the two approaches "*Fat Pipes*" and "*Short Pipes*" to find augmented paths on Erdos-Renyi graphs.

- Code

```
def build_residual(edges):
    residual_network = defaultdict(list)
    capacities = {}
    for u, v, capacity in edges:
        residual_network[u].append(v)
        residual_network[v].append(u)
        capacities[(u, v)] = capacity
        capacities[(v, u)] = 0

    return residual_network, capacities


def Max_Flow_Fat(graph):
    source = graph[0]
    sink = graph[1]
    edges = graph[2]
```

```
    residual_network, capacities = build_residual(edges)

    max_flow = 0
    max_path = defaultdict(lambda: defaultdict(int))

    #process Dijkstra max to find augmented path
    while True:

        visited = set()
        hd = heapdict.heapdict()
        hd[source] = float("inf")
        prev_map = {}
        cur_flow = 0
        path = []

        while hd:
            cur, capacity = hd.popitem()

            if cur in visited:
                continue
            visited.add(cur)

            if cur == sink:
                cur_flow = capacity
                back_idx = sink

                #back track to find path to sink
                while back_idx != source:
                    prev = prev_map[back_idx]
                    path.append((prev, back_idx))
                    back_idx = prev
                break

            for adj in residual_network[cur]:
```

```
                bottle_neck = capacities[(cur, adj)]
                if adj not in visited and bottle_neck > 0:

                    bottle_neck = min(capacity, bottle_neck)
                    if adj not in hd or bottle_neck > hd[adj]:
                        hd[adj] = bottle_neck
                        prev_map[adj] = cur

        if cur_flow == 0:
            break

        for u, v in path:
            capacities[(u, v)] -= cur_flow
            capacities[(v, u)] += cur_flow

        for u, v in path:
            max_path[u][v] += cur_flow

        max_flow += cur_flow

    max_flow_path = []
    for u in sorted(max_path.keys()):
        for v in sorted(max_path[u].keys()):
            max_flow_path.append((u, v, max_path[u][v]))

    return max_flow, max_flow_path



def Max_Flow_Short(graph):
    source = graph[0]
    sink = graph[1]
    edges = graph[2]

    residual_network,capacities = build_residual(edges)
```

```
    max_flow = 0
    max_path = defaultdict(lambda: defaultdict(int))

    while True:
        visited = set()
        queue = deque([source])
        visited.add(source)
        prev_map = {}
        cur_flow = 0
        path = []

        #BFS to find augmented path
        while queue:
            cur = queue.popleft()
            for adj in residual_network[cur]:
                bottle_neck = capacities[(cur, adj)]

                if adj not in visited and bottle_neck> 0:
                    prev_map[adj] =cur

                    if adj == sink:
                        cur_flow = float('inf')
                        back_idx =sink

                        while back_idx!= source:
                            prev = prev_map[back_idx]
                            cur_flow =min(cur_flow, capacities[(pre
v, back_idx)])
                            path.append((prev, back_idx))
                            back_idx = prev
                        break

                    queue.append(adj)
                    visited.add(adj)
```

```
                if cur_flow > 0:
                    break

            if cur_flow == 0:
                break

            for u, v in path:
                capacities[(u, v)] -= cur_flow
                capacities[(v, u)] += cur_flow

            for u, v in path:
                max_path[u][v] += cur_flow

            max_flow += cur_flow

        max_flow_path = []
        for u in sorted(max_path.keys()):
            for v in sorted(max_path[u].keys()):
                max_flow_path.append((u, v, max_path[u][v]))

        return max_flow, max_flow_path
```

- Time Complexity

  Time complexity analysis introduced in our textbook *"Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein, 3rd Edition".*

  - Fat-Pipe:

    The time complexity is $O(E \cdot |f|)$, where $|f|$ is maximum flow achievable.

  - Short-Pipe(Edmonds-Karp algorithm):
    - The time complexity is $O(VE^2)$.

- Analysis

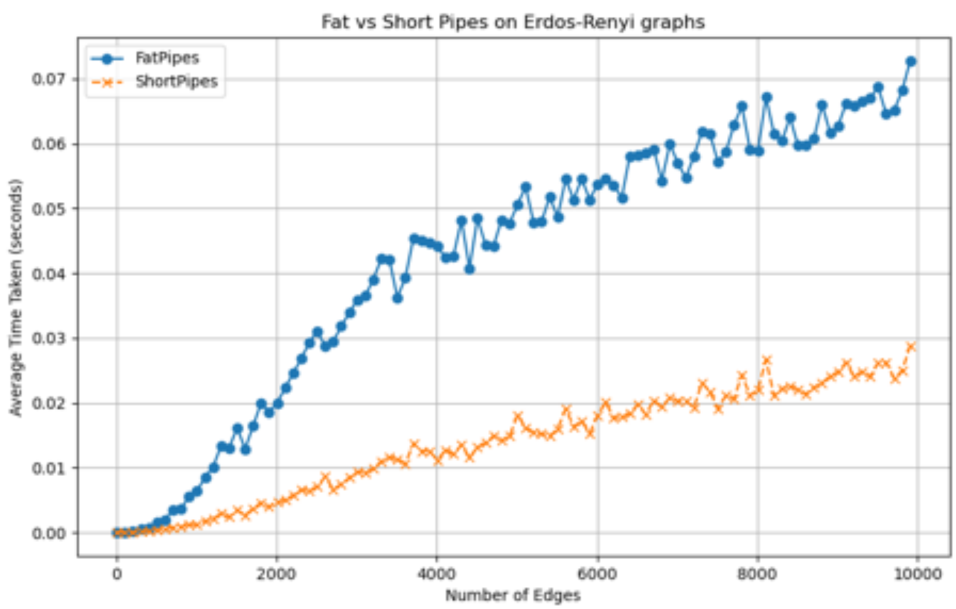
I have measured average time taken for both Max_Flow_Fat and Max_Flow_Short by iterating 10 times.

Number of nodes were fixed to 100, edges increment from 100 to 10000 by 100.

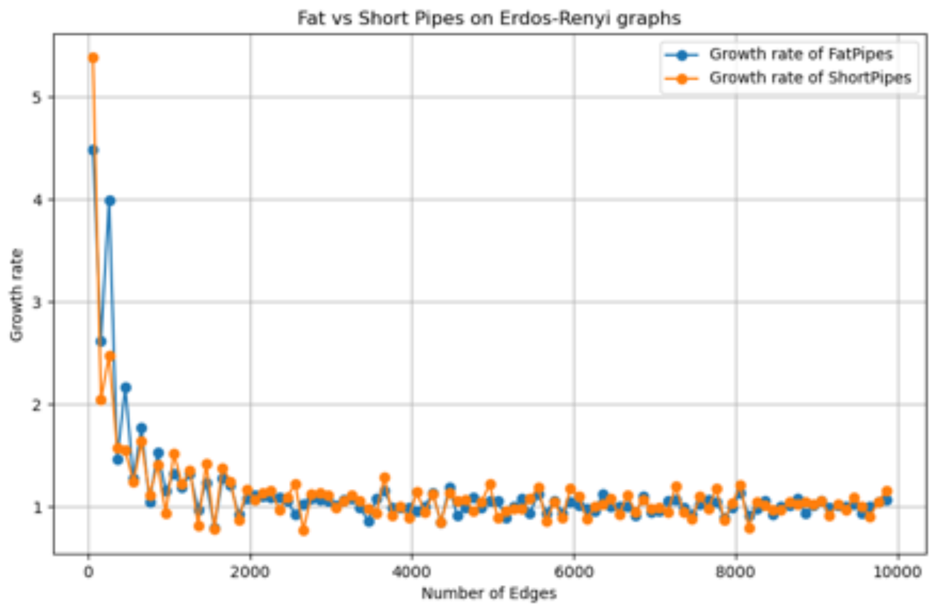
Provided generate_seq function generates graph with capacity between 5 to 10.

Below is the graph of average time taken and the growth rate of it.

- Average Time Measured



- Growth rate.



- Why Short pipes consistently outperforms Fat pipes.

  The main reason why Fat-pipes taking more time than Short-Pipes is due to usage of priority queue. Fat-Pipe iterates to search for the maximum capacity path using additional computation, while Short-Pipe uses bfs to search shortest path without comparison. This characteristic makes Fat-Pipe to take more time compared to Short-Pipes.

- Why both methods show same linear growth rate as edges increase.

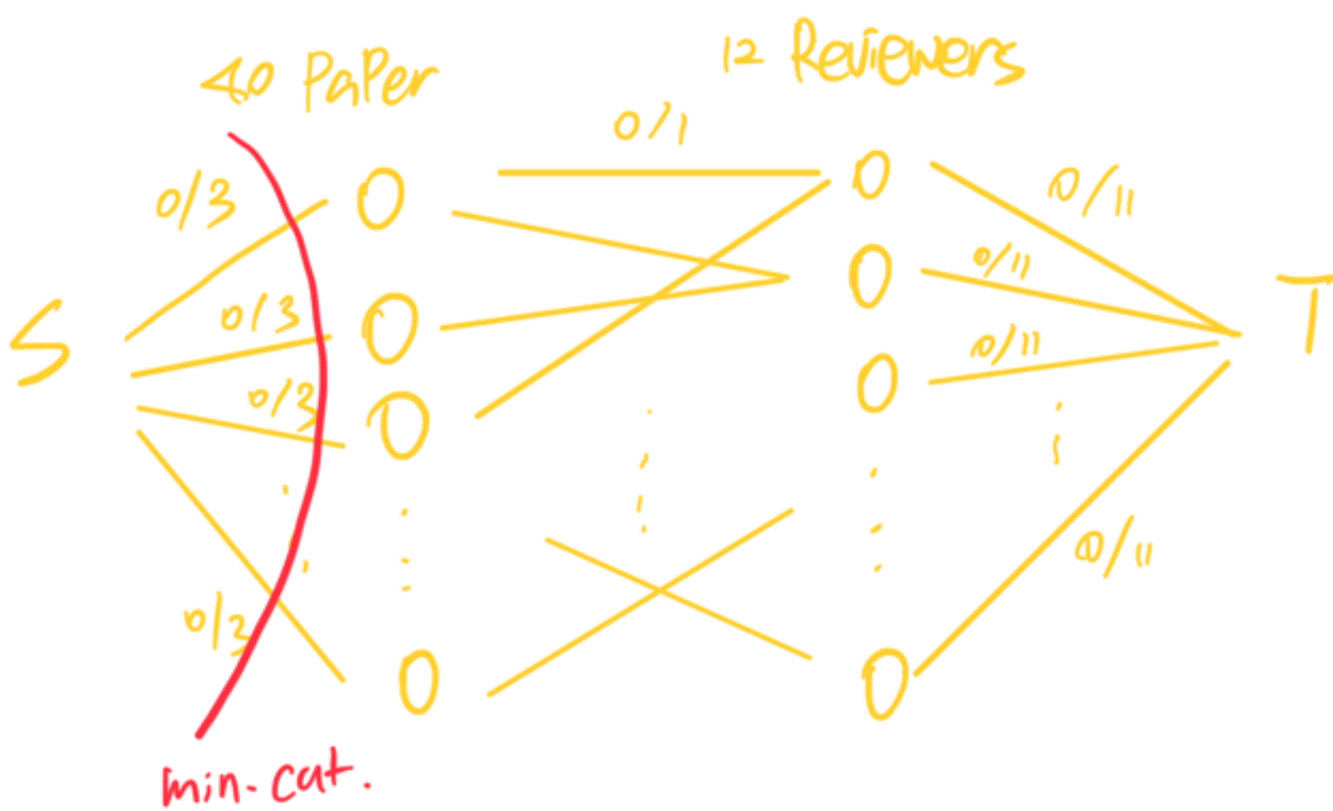
  Fat-Pipe has $O(E \cdot |f|)$ time complexity, but as edges grow up to 10000, the maximum flow which is between 5 to 10, looses its power eventually dominated by number of edges. Thus the growth rate shows linear when edges is big enough.

  Short-Pipe has $O(VE^2)$ time complexity, but as the graph becomes sparse, the edge to node ratio becomes higher, making more paths to be found between nodes. This makes Short-pipe to find augmenting path without traversing $O(E^2)$ times rather only traverses for $O(E)$. Since number of nodes are fixed to 100 dominated by number of edges, the time complexity becomes $O(E)$. Thus as edges grow with fixed number of nodes, graph becomes dense, making short-pipe to have $O(E)$ time complexity.

  For this reasons, both shows same linear growth rate that is dependent on edge numbers.

2. (10 points) Suppose you are running a conference and want to assign 40 papers to 12 reviewers. Each reviewer bids for 20 papers. You want each paper to be reviewed by 3 reviewers. Formulate this problem as a max-flow problem, i.e., describe the architecture of the network and the capacities of the edges. Assume a reviewer cannot review more than 11 papers. What is the maximum flow you can expect in your network?

- Graph:



- By max-flow min cut theorem:
    - The maximum flow available from s to paper : 3*40 = 120
    - The maximum flow available from paper to reviewers = 12*20*1 = 240(since 20 reviewers can bid for 20 papers, and the capacity is 1)
    - The maximum flow available from reviewers to t = 12 * 11 = 132

- min cut = 120, thus max flow = 120.

3. (10 points: Hall's theorem) Consider a bipartite matching problem of matching *N* boys to *N* girls. Show that there is a perfect match if and only if every subset of S of boys is connected to at least |S| girls. Hint: Consider applying the Max-flow Min-cut theorem.

Perfect match

Perfect matching is when every vertex on each side is one-to-one matched in a graph.

Max-Flow Min Cut Theorem:

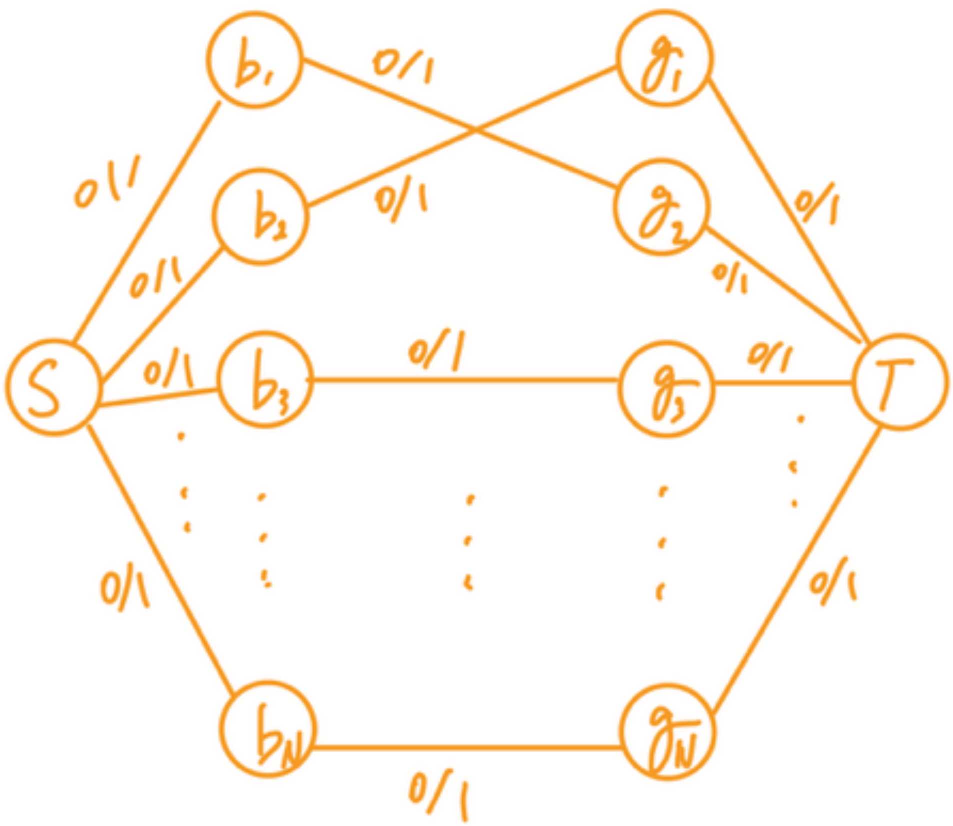
1. The s-t flow (f) is a maximum flow.
2. The residual graph has no augmenting path
3. |f| = c(S,T) for some cut (S,T)

Meaning that “max flow is min cut”

Graph

We can construct graph by making each boys and girls as a node, where boys are adjacent with source node and girls and girls are adjacent with sink node and boys. Moreover, all edges’ capacity is 1,

Below is a example of the perfect match:



Proving that if every subset of S of boys is connected to at least |S| girls, there is a perfect match

- Reformulating statement
  - If there is a perfect match between boys and girls, we can send at least N flows from S to T.
  - So we can reformulate statement as:

    If every subset of S of boys is connected to at least |S|, it is possible to send at least N flows from S to T.

- Proof by contradiction:

  Lets assume that when every subset of S of boys is connected to at least |S| girls, there is no way we can possibly send N flows from S to T.

Source to boys and girls to sink has obviously cut cost of N, and if subset of S of boys is connected to at least |S| girls the cut between boys and girls would be at least N, making max flow at least N according to max-flow min cut theorem.

Thus, we can send at least N flows from S to T, contradicting our assumption.

Proving that if there is a perfect match between boys and girls, then every subset of S of boys is connected to at least |s| girls.

- proof by contradiction:

  Lets assume that when perfect match exists, not every subset of S of boys is connected to at least |s| girls.

  If boys are connected less than |s| girls, the graph cannot form one-to-one mapping between boys and girls, contradicting the definition of perfect match.

  Thus, it contradicts our assumption, making our original statement true.

Thus for both direction of statement holds.

4. (10 points) Suppose you want to find the shortest path from node *s* to *t* in a directed graph where edge *(u,v)* has length *l[u,v] > 0*. Write the shortest path problem as a linear program. Show that the dual of the program can be written as *Max X[s]-X[t]*, where *X[u]-X[v] <= l[u,v]* for all *(u,v) in E*.

shortest path from node *s* to *t* in a directed graph where edge *(u,v)* has length *l[u,v] > 0*

$x[u, v]$ indicates edges u to v exists on the shortest path from s to t

Linear programming

$$\min \sum_{u,v} l[u,v] \cdot x[u,v]$$
$$\text{subject to}$$
$$\sum_u x[u,t] - \sum_w x[t,w] = 1$$
$$\sum_w x[s,w] - \sum_u x[u,s] = 1$$
$$\sum_u x[u,v] - \sum_w x[v,w] = 0 (\text{for every vertex} v \neq s,t)$$
$$x[u,v] > 0$$

- Making sure that s is the starting node and at least one flow is out from s: $\sum_w x[s,w] - \sum_u x[u,s] = 1$
- Making sure that t is the terminating node and at least on flow is into t: $\sum_u x[u,t] - \sum_w x[t,w] = 1$
- Making sure that every flow entering v to leave v when v is not s or t.: $\sum_u x[u,v] - \sum_w x[v,w] = 0 (\text{for every vertex} v \neq s,t)$
- Making sure that there is no negative edges: $x[u,v] > 0$

Building a primal standard format.

$$\text{Min } \mathbf{c}^T\mathbf{x}, \quad \text{subject to } \mathbf{Ax} \geq \mathbf{b},\, \mathbf{x} \geq 0.$$
$$\text{where, } c = l[u,v],\ x = x[u,v],\, A = I$$

Dual problem.

- First, build objective function: $\text{Max } b^T X$

  There is three equality constraints.

$$\sum_u x[u,t] - \sum_w x[t,w] = 1$$
$$\sum_w x[s,w] - \sum_u x[u,s] = 1$$
$$\sum_u x[u,v] - \sum_w x[v,w] = 0 (\text{for every vertex} v \neq s,t)$$

We can rewrite this to :

$$\sum_u x[u,t] - \sum_w x[t,w] = 1$$
$$\sum_u x[u,s] - \sum_w x[s,w] = -1$$
$$\sum_u x[u,v] - \sum_w x[v,w] = 0 (\text{for every vertex} v \neq s,t)$$

now it all has same format.

Thus we can define $b[\alpha] = \sum_u x[u,\alpha] - \sum_w x[\alpha,w]$.

Which corresponds with: $b[\alpha] = \begin{cases} 1 & \text{if } \alpha = s \\ -1 & \text{if } \alpha = t \\ 0 & \text{if } \alpha \neq s, \alpha \neq t \end{cases}$

Thus in objective function $b^T X$, we only care about the case when $\alpha = s \text{ and } t$.

When summing up the case when $\alpha = s \text{ and } t$, we can write our dual problem's objective function as: $\text{Max } X[s] - X[t]$

- Now consider building constraints.

  The constraint is $A^T y \leq c$, where $A = I, y = X[u] - X[v]$ and $c = l[u,v]$

  Substituting in we get, $X[u] - X[v] \leq l[u,v]$

Final expression

$$\text{Max } X[s] - X[t]$$
$$\text{subject to } X[u] - X[v] \leq l[u, v]$$