Homework Assignment Week 8: Linear Programming and Network Flow

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

Nov. 25, 2024

1. (a) Implement Ford-Fulkerson algorithm (Fat, Short pipes) for maximum flow.

```python
def Max_Flow_Fat(input):
    source, sink, edges = input

    # Convert edges to a graph
    graph = {}
    for u, v, cap in edges:
        if u not in graph: graph[u] = []
        if v not in graph: graph[v] = []
        graph[u].append((v, cap))
        graph[v].append((u, 0))

    parent = {}
    maximum_flow = 0
    set_paths = defaultdict(lambda: defaultdict(int))
    while finding_maximum_capacity_path(graph, source, sink, parent):
        bottleneck = float('inf')
        path = {}
        # Max flow is the lowest capacity in the path
        s = sink
        while s != source:
            parent_node = parent[s]
            for u, capa in graph[parent_node]:
                if u == s:
                    bottleneck = min(bottleneck, capa)
                    path[parent_node] = s
                    break
            s = parent[s]
```

```python
    # Residual and Reverse edges
    v = sink
    while v != source:
        parent_node = parent[v]
        for u, capa in graph[parent_node]:
            if u == v:
                parent_node = parent[v]
                graph[parent_node].remove((v, capa))
                graph[parent_node].append((v, capa - bottleneck))
                graph[v].append((u, capa + bottleneck))
                break
        v = parent[v]

    for curr, ahead in path.items():
        set_paths[curr][ahead] += bottleneck

    maximum_flow += bottleneck
    parent = {}

assigned_path = []
for curr in sorted(set_paths.keys()):
    for ahead in sorted(set_paths[curr].keys()):
        assigned_path.append((curr, ahead, set_paths[curr][ahead]))

return maximum_flow, assigned_path
```

Fig 1. Fat pipes' function

```python
def Max_Flow_Short(input):
    source, sink, edges = input

    # Make a graph with edges
    graph = {}
    for u, v, cap in edges:
        if u not in graph: graph[u] = []
        if v not in graph: graph[v] = []
        graph[u].append((v, cap))
        graph[v].append((u, 0))

    parent = {}
    maximum_flow = 0
    set_paths = defaultdict(lambda: defaultdict(int))
    while bfs_finding_shortest_path(graph, source, sink, parent):
        flow = float('inf')
        path = {}

        # Find the shortest path
        s = sink
        while s != source:
            parent_node = parent[s]
            for u, capa in graph[parent_node]:
                if u == s:
                    flow = min(flow, capa)
                    path[parent_node] = s
                    break
            s = parent[s]

        for node, next_node in path.items():
            set_paths[node][next_node] += flow
        maximum_flow += flow
```

```python
    # Residual and Reverse graph
    v = sink
    while v != source:
        prev = parent[v]
        for i, (vert, cap) in enumerate(graph[prev]):
            if vert == v:
                graph[prev][i] = (vert, cap - flow)
                break
        for i, (vert, cap) in enumerate(graph[v]):
            if vert == prev:
                graph[v][i] = (vert, cap + flow)
        v = parent[v]

assigned_path = []
for node in sorted(set_paths.keys()):
    for next_node in sorted(set_paths[node].keys()):
        assigned_path.append((node, next_node, set_paths[node][next_node]))

return maximum_flow, assigned_path
```

Fig 2. Short pipes' function

```python
def finding_maximum_capacity_path(graph, source, sink, parents):
    visited = set()
    visited.add(source)
    heap = heapdict()
    heap[source] = graph[source]

    while heap:
        curr_node, adj_nodes = heap.popitem()
        for v, capacity in adj_nodes:
            if v not in visited and capacity > 0:
                heap[v] = graph[v]
                visited.add(v)
                parents[v] = curr_node
    return sink in visited
```

```python
def bfs_finding_shortest_path(graph, source, sink, parents):
    visited = set()
    visited.add(source)
    q = []
    q.append(source)

    while q:
        previous = q.pop(0)
        for node, capacity in graph[previous]:
            if node not in visited and capacity > 0:
                q.append(node)
                visited.add(node)
                parents[node] = previous
                if node == sink:
                    return True
    return False
```

Fig 3. Helper functions (Dijkstra, BFS)

(b) Generate a table and a graph showing the execution time vs. edges, and then Analyze the results.

The experiment is conducted under the condition that the fixed number of nodes, which is 1,000, is used. Based on the fixed number of nodes, the number of edges increases 1,000 at each step. The result is below.

Number of edges	Estimated Fat pipes:	Fat pipes:	Estimated Short pipes:	Short pipes:
1000	6907755.278982136	0.0019867420196533203	1000000000	0.001020193099975586
2000	30403609.83816833	0.0010025501251220703	4000000000	0.0011615753173828125
3000	165918083.49272206	0.00199378204345703	9000000000	0.0010004043579101562
4000	318212198.7896909	0.004041433334350586	16000000000	0.0009860992431640625
5000	510583432.28018665	0.0060007572174072266	25000000000	0.001984834671020508
6000	750978910.4534254	0.011001110076904297	36000000000	0.0029997825622558594
7000	1320803969.5360525	0.023508548736572266	49000000000	0.004004001617431641
8000	2135974596.5557804	0.0450022220611572 3	64000000000	0.008998394012451172
9000	2914055551.6721554	0.06320929527282715	81000000000	0.014407873153686523
10000	3858814621.512745	0.0992276668548584	100000000000	0.014000892639160156

Table. 1 Fat and Short pipes time table

Table 1 shows that the real time taken with two different approaches, which are Fat and Short pipes respectively. The time taken with Fat pipe approach takes $O(E^2 \log E log|F|)$, where E is the number of edges and F is the maximum flow since we use priority queue using heap dictionary per each edge, which takes $O(E^2 \log E)$ time, and the number of finding maximum flow, which takes $O(\log|f|)$. On the other hand, according to the Introduction to Algorithm CRLS, the time taken with Short pipe approach takes $O(VE^2)$, where V is the number of vertices. As can be seen in Table 1, Short pipes approach is relatively superior to Fat pipes based on the real time taken. For visualization, a graph of two different approaches is below.

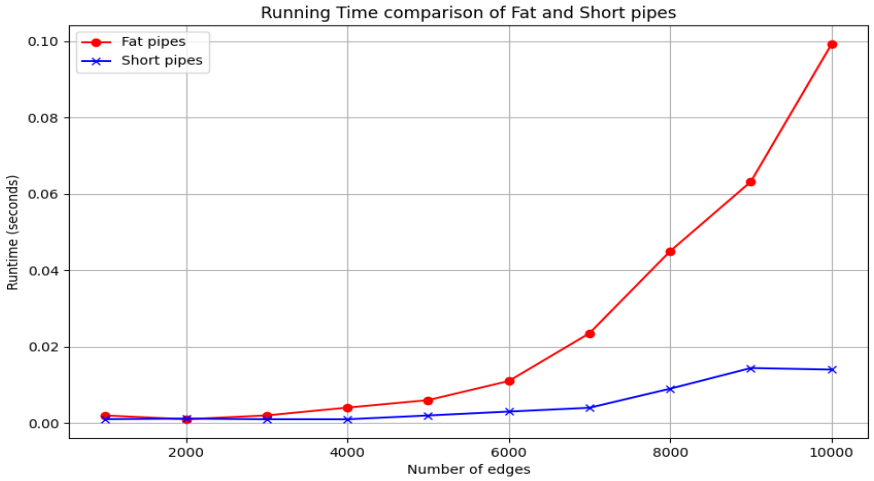

Fig. 4 A graph comparing execution time of Fat and Short pipes

As shown in Fig 4, we can guess that Short pipes takes less time than Fat pipes. In particular, the real time taken of Short pipes is 0.014 when the number of edges reach 10,000 whereas the time of Fat pipes is nearly 0.09. The trend of this graph indicates that as the number of edges increases, the time of running Fat pipes takes exponentially increase. Otherwise, it seems that the number of edges do not significantly affect the performance of Short pipes even though the time complexity is influenced by the number of edges.

Input Size:	Estimated G.R. Fat:	Real G.R. Fat:	Estimated G.R. Short:	Real G.R. Short:
2000	4.401373327551975	0.5046201848073923	4.0	1.1385837812573032
3000	5.457183682328092	1.9942925089179548	2.25	0.861247947454844
4000	1.9178873820806228	2.0213450989744812	1.7777777777777777	0.9857006673021925
5000	1.60453758285249	1.4848091558020176	1.5625	2.012814313346228
6000	1.4708250659440119	1.833286980015098	1.44	1.5113513513513515
7000	1.7587763799366598	2.1369251441203243	1.3611111111111112	1.3347639484978542
8000	1.61717760229481	1.9142917993549826	1.3061224489795917	2.24735024413481
9000	1.3642744423885078	1.4045816490333929	1.265625	1.6011605108367337
10000	1.324207638834635	1.569827134230289	1.2345679012345678	0.9717529082755539

Table. 2 Fat and Short pipes growth rate table

Table 2 shows the growth rate of each approach based on the input size, which is the number of edges. Despite the estimation growth rates of Fat and Short pipes, the real growth rate is different from the estimation of them. This may possibly be derived from the randomly generated graph where the edges might be arbitrarily distributed or skewed. The magnitude of the growth rate of Fat pipes is relatively larger than one of the growth rates of Short pipes, meaning that the changes of the numbers in Fat pipes are greater than the changes of numbers in Short pipes.

2. (a) Formulate the problem as a max-flow problem, i.e., describe the architecture of the network and the capacities of the edges.

To formulate the problem as a max-flow problem, we need to set up an architecture of the network and the capacities of the edges. The architecture is:

- Source node S: The starting point to cover all research paper.
- Papers nodes P_i $(1 \leq i \leq 40)$: connected to S with edge capacities 3.
- Reviewer nodes R_j $(1 \leq j \leq 12)$: connected to P_i with edge capacities 1
- Sink node T: The end point where all R_j are collected with edge capacities 11.

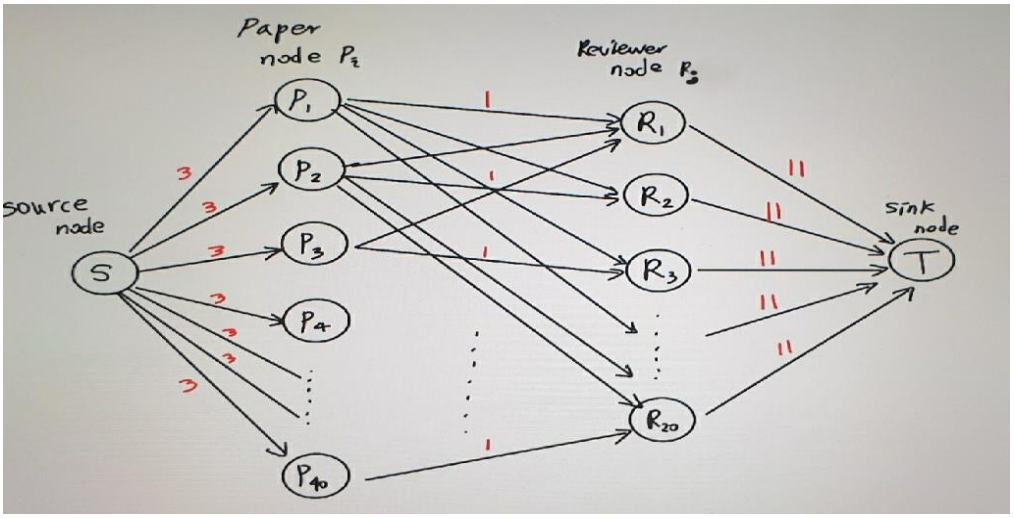

Fig. 5 Network flow architecture

(b) What is the maximum flow?

The maximum flow is 120.

We can find the maximum flow from S to T in this network. All edge capacities:

$c(S \rightarrow P_i)$: $3\ (capacity) \times 40(Paper\ nodes) = 120$

$c(P_i \rightarrow R_j)$: $1\ (capacity) \times 20(bids) \times 12(Reviewer\ \ nodes) = 240$

$c(R_j \rightarrow T)$: $11\ (capacity) \times 12(Reviewer\ \ nodes) = 132$

The minimum capacity across these layers determines the maximum flow. The bottleneck is the source node S to paper nodes P_i layer. Thus, 120 is the maximum flow.

3. (Hall's Theorem) Show that there is a perfect match if and only if every subset of S of boys is connected to at least $|S|$ girls.

Let's assume that B is a set of boys $\{b_1, b_2, \ldots, b_N\}$ and G is a set of girls $\{g_1, g_2, \ldots, g_N\}$.

Hall's Theorem:

A subset $S \subseteq B$ is connected to $N(S) \subseteq G$, where $N(S)$ is the set of neighbors of S in G. The condition $|N(S)| \geq |S|$ guarantees that S can be matched to $N(S)$ without conflict.

The maximum flow in the network is limited by the minimum cut, which partitions the network into two disjoint sets such that no additional flow can cross from source s to sink t. The cut capacity C is determined by three flows: flow from s to boys in B, flow from boys B to girls in G, and flow from girls in G to t. For any subset $S \subseteq B$, the cut capacity includes $|S|$ units from s to S, at most $|N(S)|$ units from S to G, and the sink edge capacities from $N(S)$ to t. Therefore, no cut can have a capacity less than N, meaning that the maximum flow is also N.

If a perfect matching exists, every boy $b_i \in B$ is matched to a unique girl $g_i \in G$. For any subset $S \subseteq B$, the corresponding matched girls can form a subset $N'(S)$. This subset $|N'(S)|$ is equal to $|S|$. Since every matched girl is also a neighbor of the subset S, it follows the condition $|N(S)| \geq |S|$.

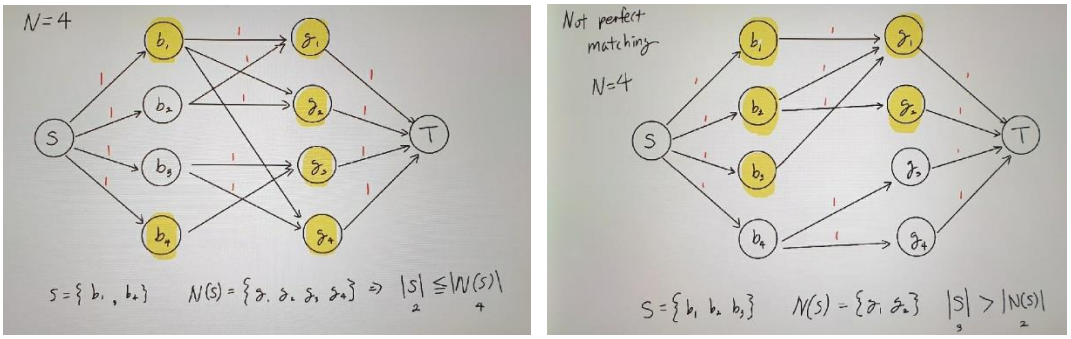

Fig. 6 Perfect match and Not perfect match cases

4. (a) Write the shortest path problem as a linear program

To find the shortest path from node s to t in a directed graph where edge (u, v) has length $l[u, v] > 0$, we need to define the problem as a linear programming. The variable $x[u, v]$ denotes existing edges from node u to v on the shortest path from node s to t.

According to the CLRS book, the shortest path means the minimum cost of the flow from s to t. So, we can express it as:

$$\text{Min} \sum_{(u,v) \in E} l[u, v] \cdot x[u, v]$$

For each node v, which is except node s and t on the path, the total flow into v equals the flow out of v:

$$\sum_{u:(u,v)\in E} x[u,v] = \sum_{w:(v,w)\in E} x[v,w]$$

From the source node s, it ensures that one unit of flow leaves s since the minimum value of the flow is 1 under the condition that $l[u,v] > 0$. This means:

$$\sum_{(s,u)\in E} x[s,u] = 1$$

From the sink node t, it ensures that one unit of flow enters t:

$$\sum_{(v,t)\in E} x[v,t] = 1$$

For variable $x[u,v]$, the value of it should be equal to or greater than 0 for all edges from node u to v, and be binary (0 or 1).

$$x[u,v] \geq 0, x[u,v] \in \{0,1\}$$

(b) Show that the dual of the problem can be written as $\boldsymbol{Max\ X[s] - X[t]}$, where $\boldsymbol{X[u] - X[v] \leq I[u,v]\ for\ all\ (u,v)}$ in E.

According to the CLRS book, we can express (a) as a generic primal linear programming form:

$$Min\ c^T x, \qquad Ax \geq b, x \geq 0,$$

$$where\ \ c = l[u,v]\ \ and\ \ \ x = x[u,v]$$

In dual problem, the form above can be converted to:

$$Max\ b^T X, \quad Ay \leq c$$

The variable x can be expressed as:

$$x_{ij} = \begin{cases} 1 & if\ the\ shortest\ path\ contains\ i \to j \\ 0 & otherwise \end{cases}$$

A formula for calculating the amount of flows at each single node i:

$b[i]$ =Amount of leaving flow from i − Amount of entering flow to i

$$= \sum_j x_{ij} - \sum_k x_{kj}$$

Based on the constraints, we can assume that:

$$b[i] = \begin{cases} 1 & if\ i\ is\ the\ starting\ node\ s \\ -1 & if\ i\ is\ the\ ending\ node\ t \\ 0 & otherwise \end{cases}$$

Therefore, the dual problem's objective function is:

$$Max\ X[s] - X[t]$$

From the constraints, we get:

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

$$\Rightarrow X[u] - X[v] \leq l[u, v]$$

We conclude:

$$Max\ X[s] - X[t],$$

$$where\ X[u] - X[v] \leq l[u, v]$$