3460:635 Advanced Algorithms

Project 1: Network Flow and Circulation with Demands Problem

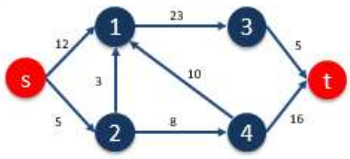
Submitted by: Sejuti Banik (UANet ID:4724011)

10/3/2019

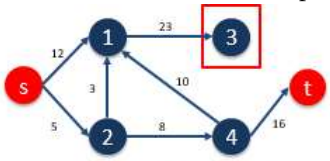
Introduction and Methodology: The goal of the project is to implement the Ford Fulkerson algorithm to solve the circulation with demands problem. To reach the goal, definitions of flow network, flow, augmenting path and residual network are required.

A **flow network** $G=(V, E)$ is a directed graph in which each edge $(u,v)\in E$ has a nonnegative capacity $c(u,v)\geq 0$ such that-- ***i)*** if $(u,v)\in E$ or $c(u,v)\geq 0$ then $(v,u)\notin E$ or $c(v,u)=0$

ii) let s be the source and t be the sink, then for each vertex $v\in V$, there exists a path $s\rightsquigarrow v\rightsquigarrow t$ in G.



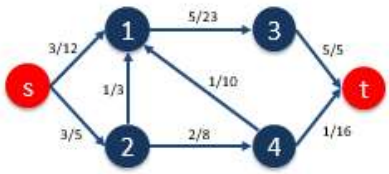
is a flow network



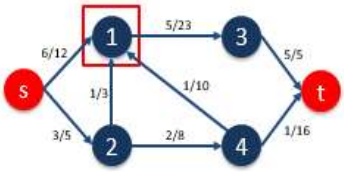
is NOT a flow network

A **flow** in G is a real-valued function $f: V\times V\rightarrow \mathbb{R}$ that satisfies the following two properties:

- Capacity constraint: $\forall$ u,$v \in V$, ***i)*** $0\leq f(u,v)\leq c(u,v)$ ***ii)*** $f(u,v)=0\Leftrightarrow (u,v)\notin E$
- Flow conservation: $\forall$ u$\in V-\setminus \{s,t\}$ $\sum_{v\in V} f(u,v)=\sum_{v\in V} f(v,u)$



is a valid flow



is NOT a valid flow

$$\max_{f\in F}|f| = \sum_{v\in V} f(s,v)$$

❑ Given a flow network $G=(V,E)$ with ***source*** s and ***sink*** t, let f be a flow in G, the ***residual network*** of G induced by f is $G_f=(V,E_f)$

where

$$E_f=\{(u,v)\in V\times V: c_f(u,v)>0\}$$

$$c_f(u,v)=\begin{cases} c(u,v)-f(u,v), & (u,v)\in E \\ f(v,u), & (u,v)\notin E \\ 0 & otherwise \end{cases}$$

Properties of Residual Network: ***i)*** G_f may also contain edges that are not in G.

ii) $c_f(u,v)=0$ when: ***a)*** (u,v) is saturated. ***b)*** $f(v,u)=0$ (no flow from v to u).

Augmenting path is a path in the residual network.

With each iteration, the Ford Fulkerson method increases the flow value starting from 0 by finding an augmenting path along the residual network. Knowing the augmenting edges in the residual graph G_f, we can find the edges along which we can change the flow to maximize the net flow. The maximum flow will ensure the minimum cut where minimum cut of a network is a cut whose capacity is minimum over all cuts of the network. The algorithm is given below.

Ford-Fulkerson method

```
1.  initialize flow f to 0
2.  while there exists an augmenting path p in Gf
3.      augment flow f along p
4.  return f
```

Moreover, the bounds of Ford Fulkerson Algorithm can be improved if the augmenting path in the residual network is the shortest path between the source and the sink. Hence, Edmond's Karp Algorithm implements the Ford Fulkerson algorithm alongwith defining BreadthFirst Search as the method to identify the shortest

path with available capacity in each iteration.

Edmonds-Karp Algorithm

➢ Implementation

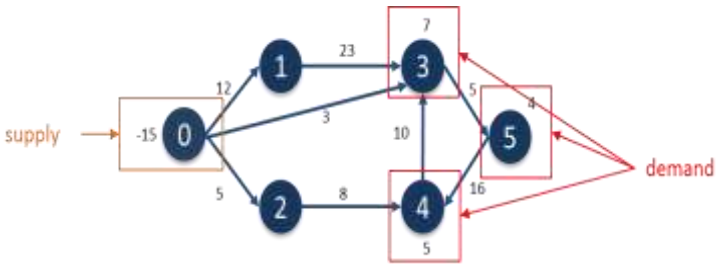
```
1.  for each edge (u, v) ∈ G.E                          ← initialization
2.      f(u, v) = 0
3.  p = SHORTEST_PATH(G, s, t)
4.  while p exists                                      ← termination condition
5.      c_f(p) = min{c_f(u, v): (u, v) ∈ p}  /* residual capacity */
6.      for each edge (u, v) ∈ p                        ← update the edges along
7.          if (u, v) ∈ G.E                               the augmenting path
8.              f(u, v) = f(u, v) + c_f(p)
9.          else
10.             f(v, u) = f(v, u) − c_f(p)
11.     G_f = RESIDUAL_GRAPH(G)
12.     p = SHORTEST_PATH(G_f, s, t)
13. return f
```

In Breadth First Search, starting from the source, each children of the source is visited and then the children of the source's children are visited and so on through the adjacency list until the destination is obtained. Each unvisited node is exactly visited once and the distance to a node from source is 1 greater than the distance to its parent from the source. Hence, Breadth First Search will provide the shortest distance as it obtains the destination by traversing along the adjacency list of the nodes and will return when it first finds the destination. If there does not exist any path between source and destination, it will return zero.

Breadth First Search Shortest Path

```
front = 0
q = ∅
for each vertex v in G (V, E)                           ← Initialization
    v.distance= ∞
    v.parent= NIL
    v.HastheNodeBeenDiscovered=White
q.add(start)
start.parent= NIL
start.distance=0
start.HastheNodeBeenDiscovered=Gray
while q ≠ ∅                                             ← Termination Condition
    front = q.poll()
    for each neighbor jEadjList(front)                  ← Traverses the graph along the
        if j.HastheNodeBeenDiscovered == White            children of the source and
            j.HastheNodeBeenDiscovered= Gray              continues until destination is
            j.distance=front.distance+1                   found if path exists
            j.parent=front
            q.add(j)
            if graph[j]==dest
                return 1
    front.HastheNodeBeenDiscovered= Black
return 0
```

Finally the circulation problem with demands is solved using the Edmond-Karp algorithm.
Given $G=(V,E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u,v) \geq 0$ and each vertex v$\in V$ has a weight d(v) – ***i)*** v demands if $d(v) > 0$ ***ii)*** v supplies if $d(v) < 0$ ***iii)*** v is transshipment if $d(v) = 0$. Below is a circulation network.



A ***circulation*** in G is a real-valued function $f: V \times V \rightarrow \mathbb{R}$ that satisfies the following two properties:
i) Capacity constraint: $\forall$ u, $v \in V$, **a)** $0 \leq f(u,v) \leq c(u,v)$ and **b)** $f(u,v) = 0 \Leftrightarrow (u,v) \notin E$
ii) Conservation: $\forall$ u $\in V - \{s,t\}$, $\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) = d(v)$
Necessary Condition for Circulation:
If $G(V,E)$ has a circulation, then $\sum_{v \in V: d(v)>0} d(v) = \sum_{v \in V: d(v)<0} d(v) = D$
Maximum flow formulation: Form G^* out of G by:

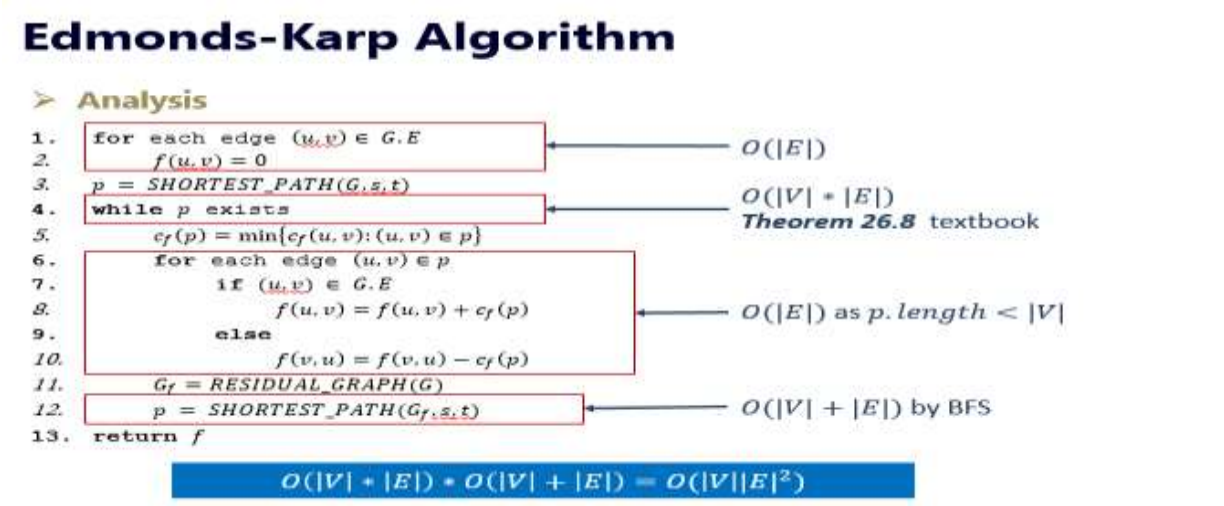
- Add new source s and sink t.
- For each v with $d(v) < 0$, add edge (s, v) with capacity -d(v).
- For each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$.

Sufficient Condition for Circulation: $G(V,E)$ has a circulation if and only if G^* has maximum flow of value D

Data Structures Used: To solve the problem statement, the network is built with graph and a two dimensional Linked List data structure is used to build the graph. In the linked list, the source is stored as the index or first dimension and the tuple of source, destination and weight are stored as the data in the second dimension. If we go through the index, then the length of the index gives the number of source-destination-weight tuples or the children of the index i.e. the source.
The Linked List has been used to implement the **Adjacency List** representation of the graph. As we are considering sparse graphs for our problem statement, the number of total edges $|E|$ is much less than square of vertex number $|V|^2$. If dense graphs were required where $|E|$ is closer to $|V|^2$then adjacency matrix is a better representation to find neighbors in $O(1)$ time. Hence, in saving space to a lot less than $O(V^2)$, adjacency list is used.

Difficulties and Solution: While generation of test cases, maximum difficulty was faced. It was tough to implement paths between source and destination randomly and solving them in random time. The solution is provided by generating random walks between the random sources and random destinations. The simulation of different graphs is demonstrated in the presentation.

Theoretical Analyses: We show the theoretical analyses through the pseudocode of the algorithms used. Below the analysis of Edmonds-Karp algorithm is given.



For Shortest Path:

Breadth First Search Shortest Path Analysis

```
front = 0
q = ∅
for each vertex v in G (V, E)
    v.distance= ∞
    v.parent= NIL
    v.HastheNodeBeenDiscovered=White
q.add(start)
start.parent= NIL
start.distance=0
start.HastheNodeBeenDiscovered=Gray
while q ≠ ∅
    front = q.poll()
    for each neighbor j∈adjList(front)
        if j.HastheNodeBeenDiscovered == White
           j.HastheNodeBeenDiscovered= Gray
           j.distance=front.distance+1
           j.parent=front
           q.add(j)
           if graph[j]==dest
              return 1
    front.HastheNodeBeenDiscovered= Black
return 0
```

Initialization for every Vertex, O(V)

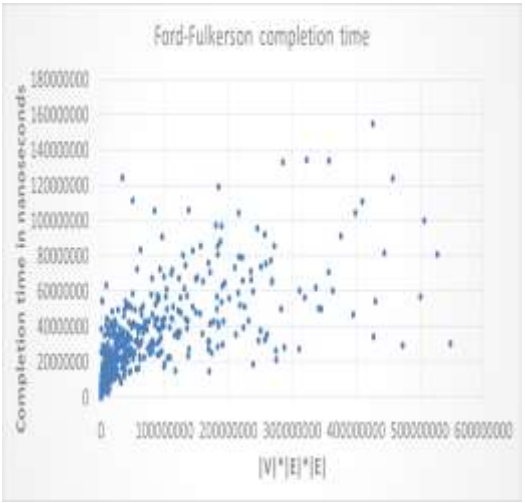
In worst case, will enqueue every vertex, O(V)

In worst case, will visit all the children of the graph. Total size of Adjacency List= sum total of edge, O(E)

The Time Complexity in Worst Case = O(V+E)

Experimental Run Results with Timing: The Ford Fulkerson and Breadth First Search Algorithm are tested with 1000 files where $5 \leq |V| \leq 100$ and $4 \leq |E| \leq 2475$. In the file, the 1st line represents children and capacity of vertex 0, 2nd line vertex 1 and so on. In a line, elements are considered in pair where the 1st one represents end of an edge with the vertex (vertex 0 for line 1 etc.) and 2nd is capacity of the edge.

```
8 11 15 18 5 43 6 12
11 7
14 8
19 9
18 22
19 31
19 34
1 43
10 4
13 30
12 9
4 27
7 8
3 7
16 41
19 49
17 2
9 6
2 8
```



SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.749318093
R Square	0.552075887
Adjusted R Square	0.551627065
Standard Error	15561386.64
Observation s	1000

ANOVA

	df	SS	MS	F	Significance F
Regression	1	2.98E+17	2.98E+17	1230.056	3E-176
Residual	998	2.42E+17	2.42E+14		
Total	999	5.4E+17			

	Coefficients	Standard Error	t Stat	P-value	Lower 95%	Upper 95%	Lower 95.0%	Upper 95.0%
Intercept	9813348.558	543864.1	18.04375	3.3E-63	8746100	10880597	8746100	10880597
X Variable 1	0.2119017	0.006042	35.07215	3E-176	0.200045	0.223758	0.200045	0.223758

(a) (b) (c)

Fig. a) Sample test case with 20 nodes and 22 edges (last node has no children), b) Time vs. |V|*|E|*|E| graph for 1000 test cases, and c) Regression analysis shows closeness to accuracy (greater than 0.7)

For Shortest Path by Breadth First Search we attach two sample test case, one where there exists a path between source and destination and another with no path between source and destination.

```
8 11 15 18 5 43 6 12
11 7
14 8
19 9
18 22
19 31
19 34
1 43
10 4
13 30
12 9
4 27
7 8
3 7
16 41
19 49
17 2
9 6
2 8
```

a)

```
The number of vertices is: 20
Source is 0 -> dest is 8 capacity is 11
Source is 0 -> dest is 15 capacity is 18
Source is 0 -> dest is 5 capacity is 43
Source is 0 -> dest is 6 capacity is 12
Source is 1 -> dest is 11 capacity is 7
Source is 2 -> dest is 14 capacity is 8
Source is 3 -> dest is 19 capacity is 9
Source is 4 -> dest is 18 capacity is 22
Source is 5 -> dest is 19 capacity is 31
Source is 6 -> dest is 19 capacity is 34
Source is 7 -> dest is 1 capacity is 43
Source is 8 -> dest is 10 capacity is 4
Source is 9 -> dest is 13 capacity is 30
Source is 10 -> dest is 12 capacity is 9
Source is 11 -> dest is 4 capacity is 27
Source is 12 -> dest is 7 capacity is 8
Source is 13 -> dest is 3 capacity is 7
Source is 14 -> dest is 16 capacity is 41
Source is 15 -> dest is 19 capacity is 49
Source is 16 -> dest is 17 capacity is 2
Source is 17 -> dest is 9 capacity is 6
Source is 18 -> dest is 2 capacity is 8
Shortest distance is 11
Shortest Path is:
1, 11, 4, 18, 2, 14, 16, 17, 9, 13, 3, 19
Execution time 0.1223353 ms
```

b)

```
The number of vertices is: 20
Source is 0 -> dest is 8 capacity is 11
Source is 0 -> dest is 15 capacity is 18
Source is 0 -> dest is 5 capacity is 43
Source is 0 -> dest is 6 capacity is 12
Source is 1 -> dest is 11 capacity is 7
Source is 2 -> dest is 14 capacity is 8
Source is 3 -> dest is 19 capacity is 9
Source is 4 -> dest is 18 capacity is 22
Source is 5 -> dest is 19 capacity is 31
Source is 6 -> dest is 19 capacity is 34
Source is 7 -> dest is 1 capacity is 43
Source is 8 -> dest is 10 capacity is 4
Source is 9 -> dest is 13 capacity is 30
Source is 10 -> dest is 12 capacity is 9
Source is 11 -> dest is 4 capacity is 27
Source is 12 -> dest is 7 capacity is 8
Source is 13 -> dest is 3 capacity is 7
Source is 14 -> dest is 16 capacity is 41
Source is 15 -> dest is 19 capacity is 49
Source is 16 -> dest is 17 capacity is 2
Source is 17 -> dest is 9 capacity is 6
Source is 18 -> dest is 2 capacity is 8
There is no path among 2 and 8
Execution time 0.448050 ms
```

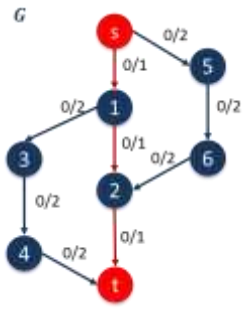
c)

Fig. a) Sample test case with 20 nodes and 22 edges (last node has no children), b) Run time when there is a path between random source 1 and destination 19, and c) Run time when there is no path between random source 2 and destination 8

<u>Insights on the Algorithm:</u> In the Ford Fulkerson Algorithm, through each iteration the flow through an augmenting path can increase or decrease. While formulating the solution of the given problem, the intuition was to decrease the flow in augmenting paths to increase the total flow along the network

```
1 1 5 2
2 1 3 2
7 1
4 2
7 2
6 2
2 2
```

(a)



(b)

Test case generation

❖ Random walk from s to t

```
1.  input V,E
2.  count = 0
3.  S = MAKE_VERTICES(V) /*generate vertex v_i, i = 1,...V-2) and save to
    set S*/
4.  while S ≠ ∅
5.      Q = RAND_SUBSET(S) /*take a random subset from S*/
6.      p = RAND_PATH(Q)    /*make a simple path from Q*/
7.      ADD_SOURSE(p,s)     /*make s the start vertex of p*/
8.      ADD_SINK(p,t)       /*make t the end vertex of p*/
9.      S = S - Q
10.     count = count + LENGTH(p)
11. while count < E
12.     RAND_EDGE(1,V-2) /*make edge between two random vertices*/
13.     count = count + 1
```

(c)

Fig. (a) File and (b) Graph representation of an extreme test case has been shown. There is an iteration where the flow of an edge decreases (iteration 2, edge (1, 2)) (c) Test case generating pseudocode

<u>**Test Case:**</u> While generating test cases three things were considered- 1) Requirement: **$\forall v \in V$, $\exists$ a path $s \rightsquigarrow v \rightsquigarrow t$ in G** 2) To satisfy: **Random walk from s to t** 3) Analyze correctness: through MATLAB
The pseudocode of test case generation is given in the above figure. While designing the test cases, special care was taken to generate random walks between different vertices. The random walk ensures that there is a shortest path among the source and sink to implement and solve the circulation with demands. In the pseudocode above, we explain the process of graph generation.