CS260: Project Report
Algorithms for Solving Maximum Flow Problem

Lu Yu, Siqing Hou, Zihao Wang

December 4, 2016

1 Introduction

Network flow problem is a kind of classical network optimization problem. We could find its applications in network packets routing, transportation scheduling, bridges destroy with minimum cost etc. Usually, a maximum flow problem needs to be solved in order to optimize practical cost for certain task. Formally, maximum flow problem [4] can be formulated as finding a flow from source node s to sink node t with a given directed graph $G = (V, E)$, where each edge e is associated with its capacity $c(e) > 0$. Many pioneering research have provided efficient solutions to maximum flow problem. In this project, we are interested in different aspects of two popular solutions, Edmonds–Karp algorithm [1] and Dinic's Blocking flow algorithm [2]. More specially, we will conduct a series of experiments with different settings to examine their performances in completeness, optimality, time complexity and space complexity. We will further discuss their differences based on the experimental results.

2 Preliminaries

Let $G = (V, E)$ be a network (directed graph) with s and $t \in V$ being the source and the sink nodes respectively. Formally, maximum flow problem [4] can be formulated as finding a maximum flow from source node s to sink node t. Each edge e is associated with its capacity $c(e) > 0$. The capacity of an edge (u, v) is a mapping $c : E \to \mathbf{R}^+$, denoted by $c(u, v)$. It represents the maximum amount of flow that can pass through an edge. A flow is a mapping $f : E \to \mathbf{R}^+$, denoted as $f(u, v)$, with subject to the following two constraints:

1. Capacity Constraint:
$$\forall (u, v) \in E, 0 \leq f(u, v) \leq c(u, v)$$
i.e., a flow on each edge doesn't exceed the capacity.

2. Local Equilibrium:
$$\forall v \in V \setminus \{s, t\} : \sum_{\{u:(u,v)\in E\}} f(u, v) = \sum_{\{u:(v,u)\in E\}} f(v, u)$$
i.e., incoming flow equals to outgoing flow at every vertex (except s and t)

The value of a st-flow can be defined as follows:
$$|f| = \sum_{\{v:(v,t)\in E\}} f(v, t)$$

which equals to the sum of values of incoming flows to the sink node t. Maximum flow problem aims at finding a st-flow with the maximum $|f|$, i.e. to route as much flow as possible from s to t. Before diving into specific solutions, let‘s first review some important concepts that will be frequently used:

1. **Residual Capacity**: a mapping function $c_f : V \times V \to \mathbf{R}^+$. For all $(u, v) \in V \times V$,

   If only one of (u, v) and (v, u) is in E, without loss of generality, let $(u, v) \in E$.

   $$c_f(u, v) = c(u, v) - f(u, v), c_f(v, u) = f(u, v)$$

   If $(u, v) \in E$ and $(v, u) \in E$

   $$c_f(u, v) = c(u, v) - f(u, v) + f(v, u), c_f(v, u) = c(u, v) + f(u, v) - f(v, u)$$

   If $(u, v) \notin E$ and $(v, u) \notin E$

   $$c_f(u, v) = c_f(v, u) = 0$$

2. **Residual Graph**: the graph $G_f = (V, E_f, c_f|_{E_f}, s, t)$, where

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

3. **Augmenting Path**: a path p from source node s to sink node t on the residual graph. The *residual capacity* of path p could be defined as

   $$c_f(p) = min\{c_f(u, v) : (u, v) \in p\}$$

4. **Augmentation**: the action of sending a flow of $c_f(p)$ along the augmenting path p. After the augmentation, the flow f is updated. Here we don't formulate the updating of the flow f because the flow f can be reconstructed from c_f, instead we formulate the updating of the residual capacity as

   $$\forall (u, v) \in p, c_f(u, v) \leftarrow c_f(u, v) - c_f(p), c_f(v, u) \leftarrow c_f(v, u) + c_f(p)$$

5. **Level Graph**: a graph $G_L = (V, E_L, c_f|_{E_L}, s, t)$ defined on the residual graph,

   $$E_L = \{(u, v) \in E_f : d(v) = d(u) + 1\}$$

   where $d(v)$ denotes the length of the shortest path from s to v in the G_f.

6. **Blocking Flow**: a st-flow such that the graph $G' = (V, E'_L, s, t)$ with $E'_L = \{(u, v) : f(u, v) < c_f|_{E_L}(u, v)\}$ containing no path from s to t.

3 Algorithms

3.1 Ford-Fulkerson Method

3.1.1 Algorithm Description

The two algorithms we are interested in are both variations of Ford-Fulkerson method [3]. The Ford-Fulkerson method takes any legal flow f as the input, typically the empty flow, in which $f(u, v) = 0$ holds for each edge $(u, v) \in E$. In each iteration, we find a augmenting path p on the residual graph G_f, calculate the residual capacity $c_f(p)$ and perform an augmentation along p by sending a flow of $c_f(p)$. When we cannot find an augmenting path any more, the resulting flow is maximal. The correctness of Ford-Fulkerson method is guaranteed by the max-flow min-cut theorem, which shows that a flow f is maximal if and only if there is no augmenting path in the residual graph G_f [3]. Ford-Fulkerson method can be written in the following pseudo-code:

Algorithm 1: Ford-Fulkerson method

```
1 Input:Network $G = (V, E)$, capacities $c$
2 Onput:A maximum flow $f$;
3 $c_f \leftarrow c$;
4 while There is an augmenting path $p$ in $G_f$ do
5     $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$;
6     for $(u, v) \in p$ do
7         $c_f(v, u) \leftarrow c_f(v, u) - c_f(p)$;
8         $c_f(u, v) \leftarrow c_f(u, v) + c_f(p)$;
9 $f \leftarrow c - c_f$;
```

3.1.2 Time Complexity

Although Ford-Fulkerson method is guaranteed to return a maximal flow after terminating, the basic Ford-Fulkerson algorithm [5] is only guaranteed to terminate if all capacities are rational, thus it is impossible to give an upper bound of the time complexity of the basic Ford-Fulkerson algorithm in general cases. A special case is that if the initial data—capacities and the initial flow—is integer, the current flow remains integer and eventually becomes maximum. We denote the maximum flow as f^*. The time complexity in this integer case is $O(|E| \cdot |f^*|)$ [5]. This complexity depends on $|f^*|$, which is pseudo-polynomial but not polynomial. In basic Ford-Fulkerson algorithm, we don't specify the strategy of finding an augmenting path in the residual graph. The following two algorithms, Edmonds-Karp algorithm and Dinic's algorithm, as two variations of Ford-Fulkerson method, use two different strategies to find an augmenting path or a blocking flow in the residual graph and thus ensure the completeness in non-rational case and improve the time complexity to be polynomial and independent to $|f^*|$.

3.2 Edmonds–Karp Algorithm

3.2.1 Algorithm Description

Compared to the basic Ford-Fulkerson algorithm, Edmonds-Karp algorithm find the shortest augmenting path each iteration. Since the path cost is unit 1 for all the edges, finding the shortest augmenting path can be done by a bread-first search on the residual graph starting from the source s. Edmonds-Karp Algorithm can be written in the following pseudo-code. In this pseudo-code, a BFS call will return a path from the given starting node to terminal node.

Algorithm 2: Edmonds-Karp algorithm

```
1 Input:Network $G = (V, E)$, capacities $c$
2 Onput:A maximum flow $f$;
3 $c_f \leftarrow c$;
4 while An augmenting path $p \leftarrow$ BFS(s,t) in $G_f$ exists do
5     $\Delta c \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$;
6     for $(u, v) \in p$ do
7         $c_f(v, u) \leftarrow c_f(v, u) - \Delta c$;
8         $c_f(u, v) \leftarrow c_f(u, v) + \Delta c$;
9 $f \leftarrow c - c_f$;
```

3.2.2 Time Complexity

Edmonds-Karp algorithm can compute the maximum flow in a flow network in $O(|V| \cdot |E|^2)$ time [5]. To prove this, we will first introduce two lemmas. Let $d_f(u, v)$ be the shortest-path distance from u to v in the residual graph G_f, where each edge has unit distance.

Lemma 1. *For all vertices $v \in V \backslash \{s, t\}$, $d_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation [10].*

Proof. Assume that for some vertex $v \in V \backslash \{s, t\}$ there is a flow augmentation that causes the distance $d_f(s, v)$ to decrease, we will drive a contradiction in this case.

Let f be the flow before the first augmentation that decreases some shortest-path distance, f' be the flow just after that augmentation. Let v be the vertex with the minimum $d_{f'}(s, v)$ whose distance was decreased by this augmentation. Thus $d_{f'}(s, v) < d_f(s, v)$. Let p be the shortest path from s to v in $G_{f'}$, and u is the predecessor of v in the path p. So that $(u, v) \in E_{f'}$ and $d_{f'}(s, u) = d_{f'}(s, v) - 1$. Because v is the vertex with the minimum $d_{f'}(s, v)$ whose distance was decreased by this augmentation, $d_{f'}(s, u) \geq d_f(s, u)$. We claim that $(u, v) \notin E_f$. If we had $(u, v) \in E_f$, then we would also have

$$d_f(s, v) \leq d_f(s, u) + 1 \leq d_{f'}(s, u) + 1 = d_{f'}(s, v)$$

which contradicts with the assumption $d_{f'}(s, v) < d_f(s, v)$.

Since $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$, we must have sent flow through the edge from v to u by this augmentation. Thus (v, u) is in a shortest path from s to t in G_f. Thus

$$d_f(s, v) = d_f(s, u) - 1 \leq d_{f'}(s, u) - 1 = d_{f'}(s, v) - 2$$

which contradict with the assumption $d_{f'}(s, v) < d_f(s, v)$. We conclude that our assumption that such a vertex v doesn't exist. □

Lemma 2. *The total number of flow augmentations performed by Edmonds-Karp algorithms is $O(|V||E|)$.*

Proof. We say that an edge (u,v) in G_f is critical on an augmenting path p if $c_f(u, v) = c_f(p)$. After we have augmented flow along p, any critical edge on the path disappears form the residual network. By the definition of $c_f(p)$, there exists at least one critical edge on any p. We will show that each edge in E can become critical at most $|V|/2$ times.

When an arbitrary edge (u, v) is critical when the current flow if f, we have

$$d_f(s, v) = d_f(s, u) + 1$$

After this augmentation, (u, v) disappears from the residual network. It won't appear later on another augmenting path until we augment (v, u). If f' is the flow before this augmentation occurs, we have

$$d_{f'}(s, u) = d_{f'}(s, v) + 1$$

Since $d_f(s, v) \leq d_{f'}(s, v)$ by Lemma 1, we have

$$d_{f'}(s, u) = d_{f'}(s, v) + 1 \geq d_{f'}(s, v) + 1 = d_{f'}(s, u) + 2$$

Thus from the time (u, v) becomes critical to the next time it becomes critical, $d_f(s, u)$ increases by at least 2. Since $(u, v) \in p$ thus $u \neq t$,

$$0 \leq d_f(s, u) \leq |V| - 2$$

Therefore, (u, v) can become critical at most $|V|/2$ times. There are at most $O(|E|)$ edges in a residual network, the total number of critical edges during the entire execution is $O(|V||E|)$. Each augmenting path has at least one critical edge, thus we have at most $O(|V||E|)$ augmenting path being augmented in the entire execution. □

Theorem 1. *Edmonds-Karp algorithm can be implemented in $O(|V||E|^2)$ time.*

Proof. In each iteration, we find an augmenting path in the residual graph by BFS and send flow along the augmenting path. This can be done in $O(|E|)$ which is identical to BFS. The number of iterations is $O(|V||E|)$ by Lemma 2. Thus the total time complexity of Edmonds-Karp algorithm is $O(|V||E|^2)$. □

3.3 Dinic's Algorithm

The introduction of the concepts of the level graph and blocking flow enable Dinic's algorithm to achieve its performance. Given a directed graph H, let $d(v, u)$ denote the distance to a vertex u from vertex v; given a source node s, we denote $d(v) = d(s, v)$. Let the ith vertex layer V_i be the set of vertices with $d(v) = i$ (where $V_0 = \{s\}$), and the ith edge layer E_i be the set of all the edges of H going from V_{i-1} to V_i. We define the *level graph* $L(s)$ as the directed graph $(\bigcup V_i, \bigcup E_i)$. Notice that $L(s)$ can be built by an extension of BFS, with the same running time $O(|E|)$ as that of the regular BFS.

In network flow problem, since we only care about the path from a source node s to a sink node t, let us prune $L(s)$ to $\hat{L}(s, t)$. Let $l = d(s, t)$, $\hat{L}(s, t)$ contains $l + 1$ layers. The ith vertex layer $\hat{V}_i$ is a subset of V_i where t is at distance $l - i$ from each node. The ith layer of its edges, $\hat{E}_i$, consists of all the edges of H going from $\hat{V}_{i-1}$ to $\hat{V}_i$. The pruning of $L(s)$ to $\hat{L}(s, t)$ can be done by running the same extension of BFS using opposite edge direction starting from t. It is easy to claim that the layered sub-graph $\hat{L}(s, t)$ is the union of all the vertices and edges residing on all the shortest paths from s to t.

Given a network G and the current flow f, we can find the level graph $G_L = (V_L, E_L)$, of the residual graph G_f from s to t. Since G_L is a subgraph of G_f, the capacity $c_f|_{E_L}(u, v)$ of each edge (u, v) in G_L is equal to its capacity $c_f(u, v)$ in G_f. A *blocking flow* in G_L is defined as a flow f such that the graph $G'_L = (V_L, E'_L)$ where $E'_L = \{(u, v)|f(u, v) < c_f|_{E_L}(u, v)\}$ contains no path from s to t.

As a sub-procedure of Dinic's algorithm called by each iteration, finding the blocking flow of a level graph G_L can be solved by the following backtracking algorithm. We run DFS from node s until we find a path from s to t, send a flow to saturate this path. Then we backtrack to tail of the edge closest to s and continue DFS until t is not reachable from s. In the following code, a DFS call will return a path from the given starting node to terminal node, ignoring the edges with capacity of 0.

Algorithm 3: Find and augment the blocking flow in level graph

```
1  Input:Level graph $G_L$, capacity $c_f$, source $s$, sink $t$
2  Onput:capacity $c_f$ after augmenting;
3  $x = s$;
4  $Path \leftarrow$ DFS$(x, t)$ ;
5  while $Path$ is not nil do
6      $\Delta c \leftarrow \min\{c_f(u, v) : (u, v) \in Path\}$;
7      for $(u, v) \in Path$ from $t$ to $s$ do
8          $c_f(v, u) \leftarrow c_f(v, u) - \Delta c$;
9          $c_f(u, v) \leftarrow c_f(u, v) + \Delta c$;
10         if $c_f(u, v) = 0$ then
11             $x \leftarrow u$
12     Backtrack to $x$ and continue DFS;
```

In each iteration, Dinic's algorithm will build a level graph G_L from s to t of the residual graph G_f, find the blocking flow f' with backtracking algorithm and augment the current flow with the blocking flow f' as $f = f + f'$. The Dinic's algorithm can be written in the following pseudo-code.

Algorithm 4: Dinic's algorithm

```
1  Input:Network $G = (V, E)$, capacities $c$
2  Onput:A maximum flow $f$;
3  $c_f \leftarrow c$;
4  while $d(t) \neq \infty$ in the level graph $G_L$ constructed from $G_f$ do
5      Find and augment the blocking flow in level graph $G_L$;
6  $f \leftarrow c - c_f$;
```

3.3.1 Time Complexity

The original implementation of this algorithm [2] has time complexity $O(|V|^2 \cdot |E|)$.

Lemma 3. *The blocking flow in the level graph $G_L = (V_L, E_L)$ can be found in $O(|V||E|)$ time [2].*

Proof. The layered sub-graph G_L is the union of all the vertices and edges residing on all the shortest paths from s to t. The length of an arbitrary path from s to t is less than $|V|$. Thus, finding a path and sending a flow along the path can be done in $O(|V|)$. After sending a flow through the path, at least one edge in the level graph is saturated, thus we need to send at most $|E|$ flows to form a blocking flow. The total time complexity of finding and augmenting the blocking flow in level graph is $O(|V||E|)$. □

Lemma 4. *The distance $d(s, t)$ strictly increases in each iteration of the algorithm.*

Proof. Let $d(i, t)$ be the distance from i to t in one iteration, $d'(i, t)$ be the distance from i to t in the next iteration. Let f be the flow in one iteration, f' is the flow in the next iteration. Let $\tilde{E}$ be the subset of E_f in which edges are selected to be augmented in one iteration, $\tilde{E}'$ be the subset of $E_{f'}$ in which edges are selected to be augmented in one iteration.

First we claim that

$$d(i,t) \leq d(j,t) + 1$$

for all $(i,j) \in E_{f'}$. There is two case for (i,j) to come to exist in $E_{f'}$.

1. $(i,j) \in E_f$. In this case, $d(i,t) \leq d(j,t) + 1$ by the definition of shortest path.
2. $(j,i) \in \tilde{E_f}$, which means we augmented (j,i) in the previous iteration. In this case, $d(j,t) = d(i,t) + 1$, $d(i,t) = d(j,t) - 1 \leq d(j,t) + 1$

Thus the claim holds. Considering an arbitrary s-t path P in $E_{f'}$. There exists at least an arc $(i,j) \notin \tilde{E}$, otherwise P should be augmented in the previous iteration. Thus either $(i,j) \notin E_f$, or $d(i,t) \neq d(j,t) + 1$. In the former case, we must have augment (j,i) in the previous iteration, which as above implies

$$d(i,t) = d(j,t) - 1 < d(j,t)$$

The latter case, since we claimed that $d(i,t) \leq d(j,t) + 1$, implies

$$d(i,t) \leq d(j,t)$$

For other $(i,j) \in P$ and $(i,j) \in \tilde{E}$,

$$d(i,t) = d(j,t) + 1$$

By now we claim that $|P| \geq d(s,t)$. By definition, $d'(s,t) = \min\{|P|\}$, thus $d'(s,t) \geq d(s,t)$. □

Theorem 2. *Dinic's algorithm can be implemented in $O(|V|^2|E|)$ time.*

Proof. The number of layers in the level graph equals to $d(s,t) + 1$. By Lemma 4, the number of layers is strictly increasing by at least one in each iteration. $d(s,t) < |V|$ since each edge has unit length. Thus the number of iterations is at most $|V|$.

In each iteration, the construction of the level graph can be done by an extension of BFS in $O(|E|)$. According to 4, finding and augmenting the blocking flow can be done in $O(|V||E|)$. Thus each iteration takes $O(|V||E|)$ time.

The total time complexity is the time complexity of each iteration multiplied by the number of iterations, which is $O(|V|^2|E|)$ □

By using a data structure called dynamic trees in the procedure of finding a blocking flow, the running time of Dinic's algorithm can be improved to $O(|V| \cdot |E| log|V|)$ [6].

4 Implementation

We have implemented both Edmonds-Karp algorithm and Dinic's algorithm (the original implementation without dynamic trees) in C language. We will describe some implementation issues that are not discussed in the algorithm description.

4.1 Networks with multiple sources and sinks

In our dataset, some test instances may have several sources denoted as $s_1, s_2 ... s_n$ and several sinks $t_1, t_2 ... t_m$, rather than just one of each. We can reformulate these networks to one-source one-sink networks [5]. We add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2 ... m$. We also add a ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2 ... n$. Any flow in the original network G corresponds to a flow in the network after adding supersource and supersink G', and vice versa [5].

4.2 Maintenance of the residual graph

Since the flow can be reconstructed from the residual graph, we only maintain an adjacent list of the residual graph. For each $(u, v) \in E$, there exists two entries in the adjacent list as edge (u, v) and (v, u). Either of the two entries has a pointer pointing to the other. In each entry we maintain two values $c(u, v)$ and $c_f(u, v)$. We denote $c_i(u, v)$ as the capacity of edge (u, v) read from the input. After initialization, we assign a zero flow to the graph as following.

$$c(u, v) = c_f(u, v) = c_i(u, v)$$

If $(v, u) \in E$,

$$c(v, u) = c_f(v, u) = c_i(v, u)$$

else

$$c(v, u) = 0, c_f(v, u) = 0$$

An edge (u, v) is considered not in the residual graph if and only if $c_f(u, v) = 0$.

4.3 Construction of the level graph

Instead of performing the BFS twice, we only perform the BFS of reverse edges from t, because we don't care the dead-ends from t in the reverse graph [2].We don't construct the level graph in a new data structure, instead we just label the distance to sink $d(v, t)$ of every node. Each node v that is not in G_L will get a distance label $d(v, t) = -1$, which will be skipped. We also skip all edges (u, v) that don't satisfy $d(u, t) = d(v, t) + 1$, which means these edges are not in the level graph.

5 Experimental Setting

In this section, we will present the experimental results and settings to compare the differences between EK and Dinic's algorithms. We mainly conduct experiments on three datasets, each of which includes several graphs with an increasing scale.

5.1 Dataset

We will use three types of data from Péter Kovács (2015) [7]. One is a simulated dataset generated by NETGEN which is a classic and popular generator developed by Klingman et al. [8] [9], and we selected datasets with different scales and sparsity. In sparse networks, we have $|E| = 8|V|$, while in dense networks, $|E| = \lfloor |V| \times \sqrt{|V|} \rfloor$, specific information is described in Table 1.

We also use Road instances data based on real-world road networks in some states in US. According to the description from reference [7], this piece of dataset includes states which have real road networks with increasing size (namely, DC, DE, NH, NV, WI, FL, and TX) and generated MCF problem instances as follows. The original undirected graphs are converted to directed graphs by replacing each edge with two oppositely directed arcs. The cost of an arc is set to the travel time on the corresponding road section. We use the path dataset from Road instances, it contains nodes and paths with information about the maximum flow capacity. The summary about this dataset could be found in the following table:

Our test suite also includes MCF instances based on large-scale maximum flow problems arising in computer vision applications. The corresponding data files were made available by the Computer Vision Research Group at the University of Western Ontario [7] for benchmarking maximum flow algorithms. The statistical information about the datasets that we have used could be found in Table 3.

Data Category	Dense	Sparse	Deg
Graph Information	$\|V\| = n$	$\|V\| = n$	$\|V\| = n$ (fixed)
	$\|E\| = m = n\sqrt{n}$	$\|E\| = m = 8n$	$\|E\| = m$ (increasing)
Ek (time complexity)	$O(n^4)$	$O(n^3)$	$O(nm^2)$
Dinic (time complexity)	$O(n^3\sqrt{n})$	$O(n^3)$	$O(n^2m)$

Table 1: Graph scale of three types of NETGEN datasets. The number of vertexes of Dense and Sparse NETGEN exponentially increase. More specifically, we have 8 dense NETGEN datasets, of which the scale ranges from 256 (2^8) to 65536 (2^{16}) vertexes. We use 15 sparse NETGEN datasets, of which the scale ranges from 256 (2^8) to 4194304 (2^{22}) vertexes. For Deg NETGEN, the number of vertexes is fixed (4096), while the number of edges increases from 8192 to 4194304. We conduct experiments on 10 Deg NETGEN datasets.

STATE NAME	DC	DE	NH	NV	WI	FL	TX
$\|V\|$	9559	49109	116920	261155	519157	1048506	2073870
$\|E\|$	29768	120576	265402	620924	1266876	2653624	5156088

Table 2: Statistical information about Road dataset.

Vision-Inv	01	02	03	04	05
$\|V\|$	245762	491522	983042	1949698	3899394
$\|E\|$	1431453	2877382	5776516	11520153	23091149

Table 3: Statistical information about Vision dataset.

5.2 Experimental Analysis

In order to compare the differences between EK and Dinic's algorithms, we conduct a series of experiments, mainly covering the time complexity, iterations and augmentations spent to calculate the maximum flow.

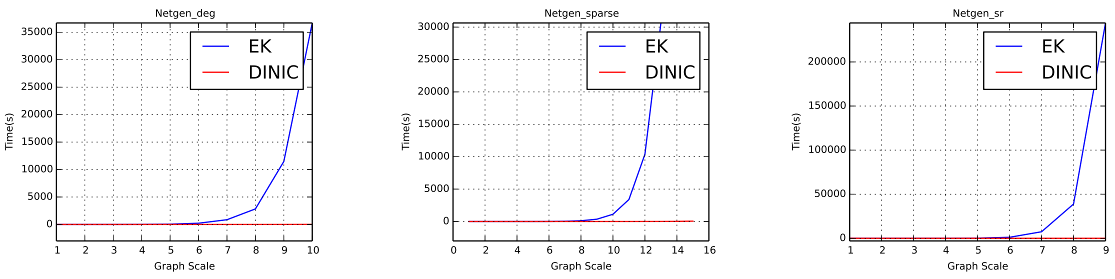

Figure 1: Running time costed to solve the maximum flow problem on three NETGEN dataset with different scale. The x axis takes log operations on the number of vertexes of different datasets because the number of vertexes always lies in a big range. Thus, as x grows, the number of vertexes of each dataset exponentially increases. Hereafter, x axis in other figures also has the same meaning if without any special descriptions.

Time Analysis: We do some experiments to compare EK and Dinic's algorithms in terms of the time spent on solving maximum flow problem. From Figure 1 we can see that EK algorithms always takes more time to solve the maximum flow problems on three datasets. One interesting finding is that the exact running time might depend on the characteristics of datasets according to the different performances between Road data and the other two datasets. Results shown in Figure 2 on Road datasets show that the running time of both EK and Dinic algorithms increases as the graph scale grows, but not so big difference as it shows in NETGEN and Vision datasets. According to the theoretical upper bound of EK and Dinic's algorithms, the time complexity is polynomial as the growth of number of vertexes. However, in our experimental results, Dinic's algorithm could much more efficient than EK to solve maximum flow problem in practice.

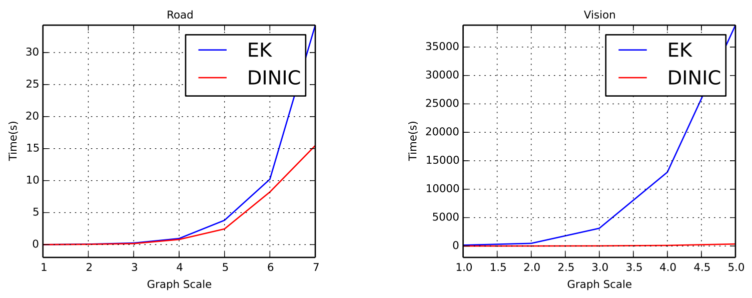

Figure 2: Running time curve of EK and Dinic's algorithms on Road and Vision datasets.

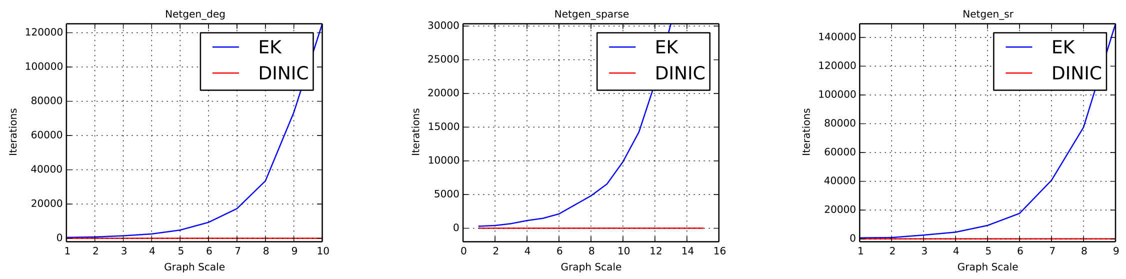

Figure 3: Illustrations to show the number of iterations costed to solve the maximum flow by EK and Dinic's algorithms as the graph scale grows on NETGEN datasets.

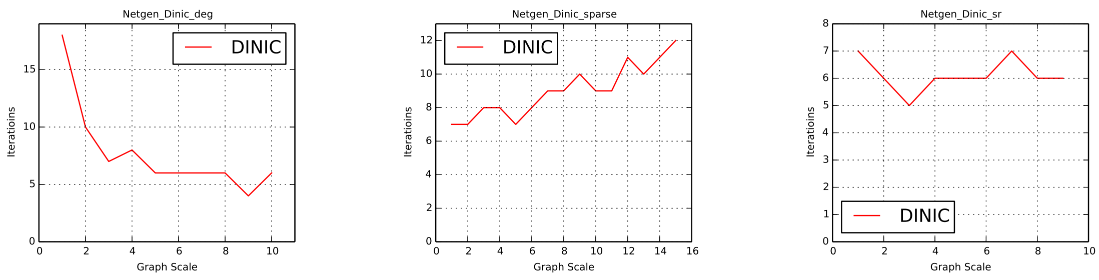

Figure 4: Illustrations to highlight the number of iterations evolutions of Dinic's algorithm as the graph scale grows.

Iterations Analysis: At this part, we do focus the number of iterations used to finish the maximum flow task. First, let's see Figure 3, which shows the results on NETGEN datasets. It's clear that Dinic's algorithm takes far less time than EK algorithm to calculate the maximum flow of NETGEN datasets. From Figure 4 we can see more details about the iteration curves of Dinic's algorithm. It fluctuates in a very small range. Also, we do experiments in Road and Vision datasets. Results are shown at Figure 5, from which we can also see the same trends.

Augmentations Analysis: Both algorithms will leverage augmenting path to solve the maximum flow problem. We capture the number of augmentations costed by both algorithms to solve maximum flow problem. The results are shown at Figure 6 and 7, from which we can see an interesting finding that both algorithms have the same number of augmentations. This is because both of them always select a shortest s-t path in the residual graph to augment. since they are using exactly the same adjacency list of the residual graph in this implementation, it is guaranteed they always select the same shortest s-t path in each augmentation.

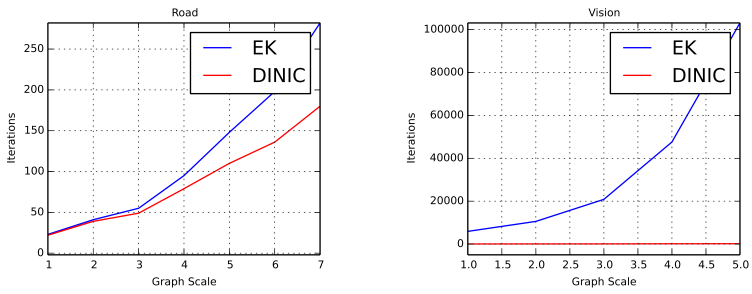

Figure 5: Figures to show the number of iterations spent to solve maximum on Road and Vision datasets.

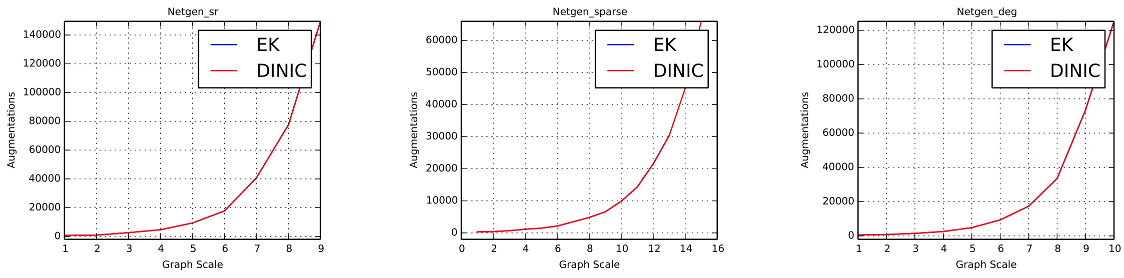

Figure 6: Figures to show the number of augmenting paths to solve maximum flow problem on different NETGEN datasets.

6 Conclusions

In this report, we broadly review the definition of maximum flow problem, and introduce the detailed theoretical analysis of two algorithms, *i.e.* EK and Dinic's algorithms. Subsequently, we design and conduct a series of experiments to explore their performances from a practical viewpoint. From the experimental results, we find that EK and Dinic's algorithms have the same number of augmentation paths. Dinic's algorithms could usually finish in a few iterations and cost much more less time, while EK will spend more time and iterations to find the maximum flow. Overall, Dinic's algorithm has better performance than EK algorithm from the experimental results. From this project, we gain a much more concrete knowledge about not only the

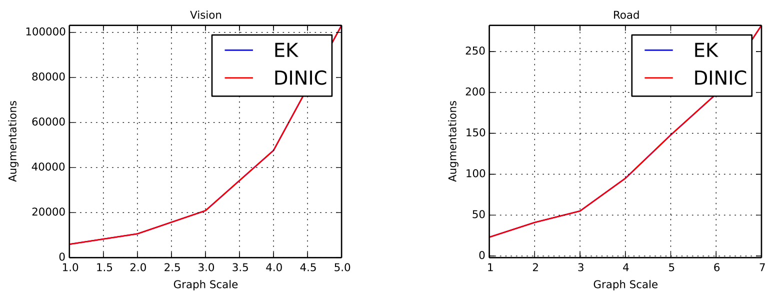

Figure 7: Figures to show the number of augmenting paths to solve maximum flow problem on Road and Vision datasets.

theoretical proof about the time complexity, but also the practical performances about these two algorithms.

References

[1] Jack Edmonds, Richard M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (JACM)19(2) (1972): 248-264.

[2] Dinitz Y. Dinitz'algorithm: The original version and even's version, Theoretical computer science. Springer Berlin Heidelberg, Volume 3895 of the series Lecture Notes in Computer Science, (2006) pp. 218-240.

[3] Ford L R, Fulkerson D R. Maximal flow through a network. Canadian Journal of Mathematics, 1956, 8(3): 399-404.

[4] Harris, T. E.; Ross, F. S. (1955). "Fundamentals of a Method for Evaluating Rail Net Capacities" (PDF).Research Memorandum. Rand Corporation.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition. The MIT Press.

[6] Sleator D D, Tarjan R E. A data structure for dynamic trees. Journal of computer and system sciences, 1983, 26(3): 362-391.

[7] Péter Kovács (2015) Minimum-cost flow algorithms: an experimental evaluation, Optimization Methods and Software, 30:1, 94-127.

[8] DIMACS, Network Flows and Matching: First DIMACS Implementation Challenge, 1990–1991. Available at ftp://dimacs.rutgers.edu/pub/netflow/

[9] D. Klingman, A. Napier, and J. Stutz, NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems, Manage. Sci. 20 (1974), pp. 814-821.

[10] Cormen, Thomas H. and Stein, Clifford and Rivest, Ronald L. and Leiserson, Charles E., Introduction to Algorithms (2nd ed.), 2001, McGraw-Hill Higher Education.