An Experimental Study of *k*-vertex Connectivity Algorithms

AZZEDDINE RIGAT

Hunan University
CISE - College of Information Science and Engineering,
DCS - Departement of Computer Science
41000 - Changsha- China
rigat.azzeddine@yahoo.com

Abstract. We present an algorithm for the *k*-vertex connectivity problem, which runs in $O(km)$ time. The time complexity of the algorithm depends on the connectivity, *k*, and the edges number, m. The algorithm mainly performs Breadth first search (BFS) to find all the disjoint paths. The goal of this paper is to provide an experimental comparison of the efficiency of *k*-vertex algorithms. We compare the running times of several standard algorithms, as well as a new algorithm that we have recently developed. We study the state-of-art Max-Flow/Min-Cut algorithms; 'Dinic', 'Push-relabel', and 'Pseudoflow'. Experiments show that our algorithm performs well on *k*-vertex connectivity problem.

Keywords: graph algorithms, *k*-vertex connectivity, maximum flow, minimum cut, network design problem, reliability.

(Received April 24th, 2012 / Accepted June 28th, 2012)

1 Introduction

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m.$ The vertex connectivity of two vertices s,t $\in$ V , denoted by $k(s,t)$, is defined as the least number of vertices chosen from $V \setminus$ *{s, t}*, whose deletion from G would destroy every path between s and t, and if $(st) \in E$ then let $k(s,t) = n - 1$. A graph is called *k*-vertex-connected if its vertex connectivity is *k* or greater. This means a graph G is said to be *k*-connected if there does not exist a set of $k - 1$ vertices whose removal disconnects the graph.

Menger's theorem [17] was the first which characterizes the connectivity of a graph in terms of the number of independent paths between vertices. The vertex-connectivity statement of Menger's theorem is as follows: Let G be a finite undirected graph and s and t two nonadjacent vertices. Then the theorem states that the size of the smallest vertex cut for s and t is equal to the maximum number of pairwise vertex-independent paths from s to t.

Vertex-independent path: If s and t are vertices of a graph G, then a collection of paths between s and t is called vertex-independent paths if no two of them share a vertex (other than s and t themselves).

Even and Tarjan [7] were among the first to present Max-Flow based connectivity algorithms. Subsequent results include the work of Kleitman [15], Galil [9, 10], Esfahanian and Hakimi [5], and Henzinger and Rao [12]. The problem of determining whether the connectivity is larger than a prescribed value, without computing the actual value of *k* has been studied by Tarjan [20].

Many algorithms were proposed, among which there are some well-know *k*-vertex connectivity algorithms and Max-Flow algorithms that show optimal performances, such as:

1.1 For the *k*-vertex connectivity algorithms

- Gabow algorithm [8] using expander graph in *O((n+min{*$k^{5/2}$*,* $kn^{3/4}$*})m)*
- Piotr algorithm [18] using matrix techniques in *O(*$n^{1.575}$*+n*k^2*)*

- Yuichi and Hiro algorithm [21] using property testing with a time complexity depends exponentially on *k*.

1.2 The Max-Flow/Min-Cut algorithms show better experimental performance to find the graph connectivity or the *k*-vertex connectivity, these are the best well known and standard algorithms for this kind of problem:

- Dinic [4] using augmenting paths technique with time complexity of $n\sqrt{m}$ [7]
- Goldbereg-style [3] using push-relabel methods in $O(nmlog(n))$
- Pseudoflow [13] using normalized tree [16]in $O(nmlog(n))$
- Boykov and Kolmogorov [1] using augmenting paths technique in *O(kn*^{2}*m)*, this algorithm outperforms all the Max-Flow/Min-Cut algorithms in some of the Computer Vision and Pattern Recognition problems.

The goal of this paper is to compare experimentally the speed of several Max-Flow/Min-Cut algorithms on graphs typical for *k*-vertex connectivity, as well as presenting a new *k*-vertex connectivity algorithm. In Section 2, we describe the interested Max-flow/Min-cut algorithms. In Section 3, we provide the relation between Max-Flow/Min-Cut algorithms and *k*-vertex connectivity problem. Section 4, introduces a new *k*-vertex connectivity algorithm that we developed. In Section 5, we tested our new algorithm and the three standard Max-Flow/Min-Cut algorithms: pseudoflow algorithm [13, 2], Hi_pr version of Goldberg-style 'push-relabel' method [11, 3], and Dinic algorithm [4]. More detailed conclusions are presented in Section 6.

2 Description of Max-Flow/Min-Cut Algorithms

2.1 Dinic algorithm

In order to describe this algorithm we need this definition.
Layered Network: Given a flow, f, and the corresponding residual graph, G_f = *(V, E*$_f$*)*, a layered network $AN(f)$ (AN stands for Auxiliary Network) is constructed as follows:
Using breadth-first-search (BFS) in G_f, we construct a tree rooted at s. Each node in V gets a distance label which is its distance in the BFS tree from s. Thus, the nodes that have an incoming arc from s get assigned a label of 1 and nodes at the next level of the tree get a label of 2, etc. Nodes with the same label are said to be at the same layer. Let the distance label of t be $k = d(t)$ then all nodes in layer k are removed from the layered network and all arcs are removed except those which go from a node at one layer to a node at the next consecutive layer. This layered network comprises all of the shortest paths from s to t in G_f.

The basic process of the Dinic algorithm is outlined below.

1. The Dinic algorithm constructs a layered network between s and t.
2. If a layered network of edges between s and t could not be constructed then the total flow is maximum thus stop.
3. Edge flows in the layered network are updated using augmenting paths to give a maximal flow in the layered network by finding the shortest $s \rightarrow t$ path along non-saturated edges of the residual graph. If a path is found then the algorithm augments it by pushing the maximum possible flow that saturates at least one of the edges. Repeat this step until there is no $s \rightarrow t$ path.
4. The flows in the layered network are then used to update the flows in the main network.
5. Repeat from step 1.

2.2 Goldbereg-style algorithm

Push-relabel algorithm uses quite a different approach. It maintains some pre-flow f, during the algorithm, there are 'active' nodes that have a positive 'flow excess' other than s and t. The algorithm repeatedly looks for a vertex $u \neq t$ with excess flow, and tries to reduce its excess by complementing $f(e)$ on some arc of G_f out of u, preferably one that may lead to the sink. The algorithm stops when there are no vertices other than s with positive excess. While, the algorithm maintains a labeling of nodes giving a low bound estimate on the distance to the sink along non-saturated edges. The algorithm attempts to 'push' excess flows towards nodes with smaller estimated distance to the sink. Typically, the 'push' operation is applied to active nodes with the largest distance (label) or based on FIFO selection strategy.

2.3 Pseudoflow algorithm

The first step in the Pseudoflow algorithm is to start by a Simple-Init, during this phase all the Source-adjacent and sink-adjacent arcs are saturated, thus the source and sink have no role to play in the next Stage. The second step is the Min-Cut phase where the algorithm is looking for (pushing flow) an admissible arc for merging; looking for a path from the source to the sink. A relabeling of a node is the increase of a node's label by one unit. A node of label h is relabeled to $h + 1$, if there is no merger arc in the residual graph to a neighbor of label $h - 1$, and if all its children have label at least $h + 1$. The algorithm stops when there are no vertices with positive excess and of label $\prec n$.

3 Adaptation of Max-Flow Algorithms to Solve *k*-vertex Connectivity Problem

In this section, we will cover some of the basic ideas in computing the vertex connectivity of a graph G, using Max-Flow/Min-Cut algorithms; similar ideas are applicable to digraphs. All edges in the graph are assigned some capacity. It has been shown that $k(s, t)$ for a pair of non adjacent vertices s and t can be determined by solving a Max-Flow problem in a particular network, as described below [6]:

- Input: Graph $G = (V, E)$, and a pair of non-adjacent vertices s and t.
- Output: Value for $k(s, t)$.
  1. Replace each edge $pq \in E$ with arcs (p, q) and (q, p), and call the resulting digraph D.
  2. For each vertex u other than s and t in G, replace u with two new vertices u_1 and u_2, and then add the new arc *(u_1, u_2)*. Connect all the arcs that were coming to u in G to u_1, and similarly, connect all the arcs that were going out of u in G to u_2 in D.
  3. Assign s as the source vertex and t as the sink vertex.
  4. Assign the capacity of each arc to 1, and call the resulting network H.
  5. Find a Max-Flow function f in H.
  6. Set $k(s, t)$ equal to the total flow of f. Stop.

4 New *k*-vertex Connectivity Algorithm

In this section, we present our new algorithm, which improved empirical performance of augmenting path techniques for *k*-vertex connectivity. Normally, augmenting path based algorithms (for example Dinic [4]), start a new BFS (Breadth-First-Search) for source $\rightarrow$ sink paths as soon as all paths of a given length are exhausted. The new algorithm presented here is mainly based on BFS; worldly speaking our algorithm builds BFS of each vertex-independent path.

4.1 Algorithm's Overview

Our algorithm composes of three phases: initialization, building, and augmentation. In the initialization phase it inserts all the Source-adjacent in the list SA, and then it maintains a search tree S with root at the source s. The vertices that are not in S are called 'free'. We have : $S \subset V$, $s \in S$.

The free vertices can be either 'deactivated' or 'visited'. The visited vertices represent the vertex participate in a previous vertex-independent path while the deactivated vertices are the vertices which can add to the tree. The S tree composes of 'active' and 'passive' vertices. The active vertices represent the outer border in the tree while the passive vertices are internal. The tree can not grow when it finds a Sink-adjacent deactivated vertex, or when there is no active vertex. An augmenting path is found as soon as the algorithm finds a Sink-adjacent deactivated vertex.

The algorithm executes the initialization phase, and then iteratively repeats the following two stages:

- building phase: search tree S grows until it finds an $s \rightarrow t$ path
- augmentation phase: the found path is augmented.

At the initialization phase all the Source-adjacent vertices add to SA and assign the Source as their parent, and then assign the Sink as parent for all the Sink-adjacent vertices.

At the building phase the active vertices explore adjacent vertices and acquire new children from a set of deactivated vertices. The newly acquired vertices become active members of the search tree. As soon as all neighbors of a given active vertex are explored, we pass to the next active vertex. The building phase terminates if an active vertex encounters a neighboring vertex that has sink as its parent. In this case, we detect a path from the source to the sink. The augmentation phase assigns visited value to all the vertices within the path found.

After the augmentation phase is completed the algorithm returns to the building phase. The algorithm terminates when the search tree S has no active vertices left. This implies that all vertex-independent paths are listed.

4.2 Details of our Implementation

Assume that we have a directed graph *G = (V, E)*. We will keep a list of all the active vertices, A, and all the Source-adjacent, SA. The general structure of the algorithm is:

```
Initialization: $S \leftarrow$ {Source};
$SA \leftarrow$ {Source-adjacent vertices};
$A \leftarrow \oslash$;
$PARENT$ {Source-adjacent vertices} $\leftarrow$ Source;
$PARENT$ {Sink-adjacent vertices} $\leftarrow$ Sink;
$GlobalLabel \leftarrow 1$;
initialize all the vertices' label with $Label \leftarrow 0$;

while $TRUE$ do
    Build $S$ to find an augmenting path $P$ from $s$ to $t$;
    if $P = \oslash$ then
        terminate
    end
    $GlobalLabel \leftarrow GlobalLabel + 1$;
    Augment on $P$;
end
```

The details of the building and augmentation stages are described below. It is convenient to store content of search tree S via flag $TREE(p)$ indicating affiliation of each vertex p so that:

$TREE(p) = s$ if $p \in S$ or $TREE(p) = \oslash$ if p is 'free'

If vertex p belongs to the search tree S then the information about its parent will be stored as $PARENT(p)$.

```
while $SA$ do
    Pick a vertex $p \in SA$;
    if $Visited(p) = FALSE$ then
        if $PARENT(p) = Sink$ then
            $PARENT(p) \leftarrow Source$;
            return $p$;
        end
        for every neighbor $q$ do
            if $Visited(q) = FALSE$ then
                if $Label(q) \prec GlobalLabel$ then
                    if $PARENT(q) = Sink$ then
                        $PARENT(q) \leftarrow p$ ;
                        Return $q$;
                    end
                    $Label(q) \leftarrow GlobalLabel$;
                    $PARENT(q) \leftarrow p$;
                    $A \leftarrow A\{q\}$;
                end
                remove the edge from $p$ to $q$;
            end
        end
        remove the edge from the source to $p$;
        remove $p$ from $SA$;
    end
end
while $A \neq \oslash$ do
    Pick an active vertex $p \in A$;
    for every neighbor $q$ do
        if $Visited(q) = FALSE$ then
            if $Label(q) \prec GlobalLabel$ then
                if $PARENT(q) = Sink$ then
                    $PARENT(q) \leftarrow p$;
                    return $q$;
                end
                $Label(q) \leftarrow GlobalLabel$;
                $PARENT(q) \leftarrow p$;
                $A \leftarrow Aq$;
            end
            remove the edge from $p$ to $q$;
        end
    end
    Remove $p$ from $A$;
end
$P \leftarrow \oslash$;
```

4.2.1 Building phase

At this stage the search tree S builds.

4.2.2 Augmentation phase

The input of this stage is sink-adjacent vertex p.

```
while $p \neq source$ do
    $Visited(p) \leftarrow TRUE$;
    $p \leftarrow PARENT(p)$;
end
```

The building phase is a simple BFS and it stops when the tree can not grow, in the case when the tree encounters a sink-adjacent vertex; that mean there is at least one path from the source to the sink, by backtracking the parents from the sink-adjacent to the source-adjacent we have the shortest path between the source and the sink. Each vertex that participates in the current shortest path will be removed from the graph by assigning it as visited. At the End, the algorithm has *k*-vertex independent paths, which lead to the *k*-vertex connectivity according to Menger's theorem [17].

4.3 Time Complexity

The building stage is repeated at most $k + 1$ times executing BFS which leads to $O(km)$, while in the augmentation stage the algorithm backtracks the parents at most n times for all the vertex-independent paths which leads to $O(n)$, because every time when the algorithm trackbacks a parent it assigns the current vertex as visited and will not use it further. Then as a result the time complexity of this algorithm is $O(km)$.

5 Experimental Tests on *k*-vertex Connectivity

5.1 Implementation

In this section, we test our algorithm in addition to the following standard Max-Flow/Min-Cut algorithms:

- DINIC [4]
- Push_Relabel [3]
- Pseudoflow [13]

For DINIC, we used the implementations written by Setubal [19], for Push_Relabel, we used the implementation of the Two-Level Push-Relabel Algorithm (hi_pr, version 3.6), and for Pseudoflow, we used the highest label pseudoflow implementation (Pseudo_fifo, version 3.2)[2].

5.2 Computing Environment

The experiments were run on a CentOS-5 workstation with a dual core of 2.7 GHz CPU and 4 GB of RAM. All codes were written in C and compiled with the gcc compiler using '-O4' optimization option.

6 Problem Classes

We test codes performance on DIMACS problem families [14] and on a random graph. We use RMF-Long, RMF-Wide, Wash-Line-Moderate, AK, Acyclic-Dense problem families, and the random graph family. The instance generated depends on a random seed. These problem classes are:

- AC: The acyclic dense network family with parameter *k* has n=2^k nodes and $n(n + 1)/2$ arcs.
- AK: The AK generator was designed by Goldberg and Cherkassky (1997) as a hard set of instances for push-relabel and Dinic's algorithms. Given a parameter *k*, the program generates a unique network with $4k + 6$ nodes and $6k + 7$ arcs. The instance does not depend on a random seed in that the graph, given the number of nodes, is unique.
- GENRMF-Long: This family is created by the RMFGEN generator of Goldfarb and Grigo-riadis (1988). A network with n=2^x nodes is generated using parameters a=$2^{x/4}$ and b=$2^{x/2}$.
- GENRMF-Wide: This family is created by the RMFGEN generator. A network with n=2^x nodes is generated using parameters a=$2^{2x/5}$ and b=$2^{x/5}$.
- Washington Line-Moderate: A network with n = 2^x nodes in this family is generated by the Washington generator using function = 6, arg1 = 2^{x-2}, arg2 = 4, and arg3 = $2^{(x/2)-2}$.
- Random graph: its random function depends on the seed, and the vertex degree (vertex neighbors) is randomly selected for each vertex; from zero to the half participated vertices of the current problem size.

6.1 Testing Methodology

For each problem type of a particular size, we generated 10 instances each using a different seed. The sequence of seeds was itself generated randomly. For each instance, we average over 5 times runs. Thus, for instances that depend on a random seed, each data point for a given problem size is the average of 50 runs.

6.2 Results and Analysis

In this section, we provide the results of our experiments for each problem family.

6.2.1 Acyclic-Dense problems

Figure 1 shows that 'Our' and Pseudo_fifo codes are the fastest; it's clear here that Dinic implementation is too slow and can not solve AC problems bigger than 8.4 millions arcs.

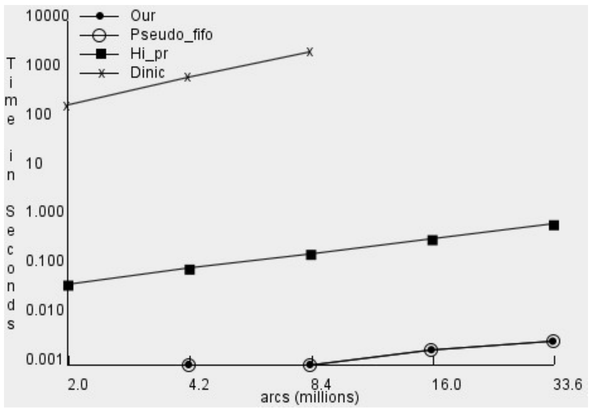

Figure 1: Running times for AC instances

6.2.2 AK problems

For these family problems, Figure 2 shows that 'Our' code is the fastest. On the other hand Hi_pr does not support the adaptation phase (3) for the AK problems.

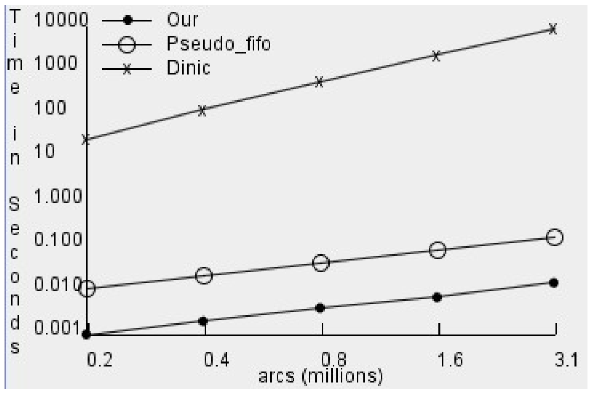

Figure 2: Running times for Ak instances

6.2.3 RMF_Long problems

Figure 3 shows that 'Our' implementation has a linear growth rate and is the fastest for most of the instances, but it loses against Hi_pr at the last instance. On the other hand Pseudo_fifo shows abnormally in some instances which cost it a big time consumption.

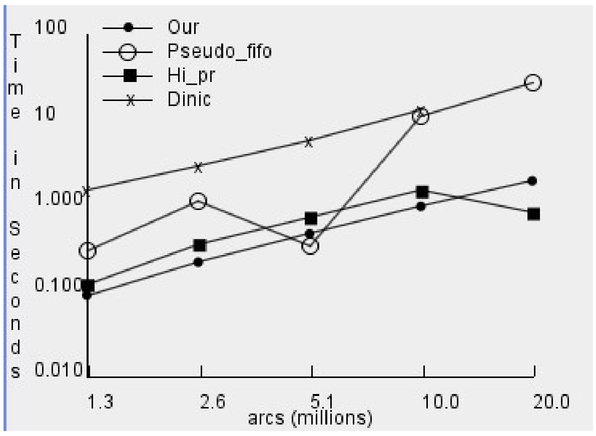

Figure 3: Running times for RMF_LONG instances

6.2.4 RMF_Wide problems

Figure 4 shows that 'Our' and Hi_pr codes are the fastest; but Hi_pr beat 'Our' algorithm at the last instance. It's clear here also that Pseudo_fifo code shows no linear growth rate.

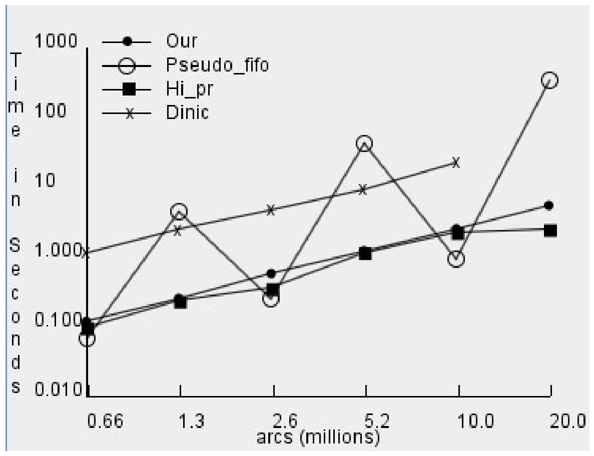

Figure 4: Running times for RMF_Wide instances

6.2.5 Wash_Line_Moderate problems

Figure 5 shows that while the running time of 'Our' implementation keeps converging to that of Hi_pr algorithm. Pseudo_fifo code is the fastest.

6.2.6 Random problems

Figure 6 shows that 'Our' code is the fastest for all the instances except the last one, where it falls in the middle

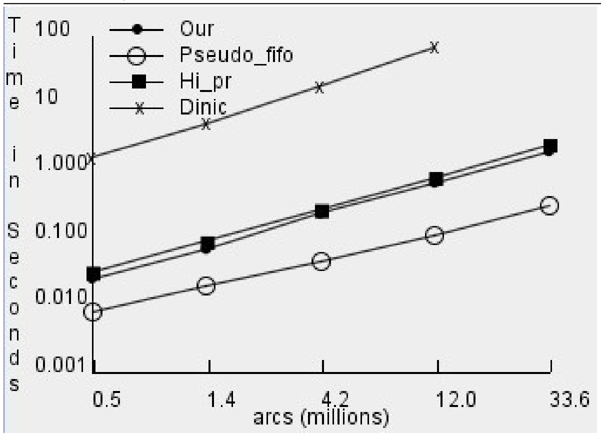

Figure 5: Running times for Wash_Line_Moderate instances

of the Pseudo_fifo and Hi_pr codes.

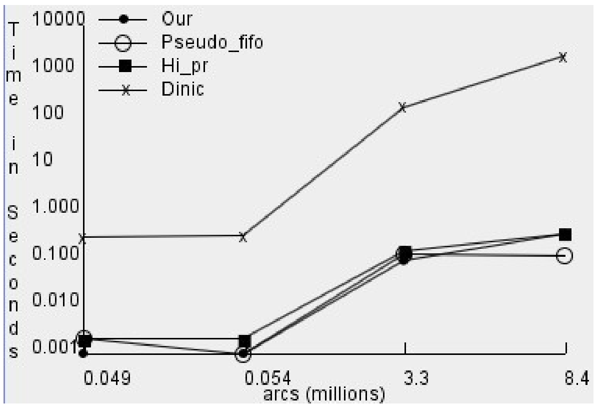

Figure 6: Running times for Random instances

6.2.7 Details on the Algorithm and the Reading phases

Finally, in the tables from 1 to 12 we have these notations:

- N: number of vertices
- M: number of edges
- PF: Pseudo_fifo
- HI: Hi_pr
- xxxx : the code crashes on the target instances.
- — : we do not include that running time.
- m : million
- k : thousand

These tables show the details of the Reading phase and Algorithm phase of all families, we did not include the Reading phase of Dinic algorithm because its Algorithm phase already shows slower convergence to solve *k*-vertex connectivity problems. The Reading phase of ‘Our’ algorithm outperforms all the other algorithms, because our implementation does not need the adaption of Max-Flow algorithms (section 3). Note : all the running times are in Seconds

Table 1: Running times of Algorithm phase for AC instances

N/k	2.0	2.9	4.1	5.8	8.2
M/m	2.0	4.2	8.4	16	33.6
$Dinic$	192.940	729.900	2361.390	xxxx	xxxx
HI	0.043	0.091	0.182	0.362	0.729
PF	0.000	0.001	0.001	0.002	0.003
Our	0.000	0.001	0.001	0.002	0.003

Table 2: Running times of Reading phase for AC instances

N/k	2.0	2.9	4.1	5.8	8.2
M/m	2.0	4.2	8.4	16	33.6
$Dinic$	—	—	—	—	—
HI	2.297	5.007	11.083	24.614	57.779
PF	1.799	3.792	7.610	15.448	30.821
Our	1.012	2.136	4.276	8.579	17.228

Table 3: Running times of Algorithm phase for Ak instances

N/m	0.13	0.3	0.5	1.0	2.0
M/m	0.2	0.4	0.8	1.6	3.1
$Dinic$	27.70	128.51	558.04	2252.99	9020.38
HI	xxxx	xxxx	xxxx	xxxx	xxxx
PF	0.011	0.021	0.042	0.083	0.166
Our	0.001	0.002	0.004	0.007	0.015

Table 4: Running times of Reading phase for Ak instances

N/m	0.13	0.3	0.5	1.0	2.0
M/m	0.2	0.4	0.8	1.6	3.1
$Dinic$	—	—	—	—	—
HI	xxxx	xxxx	xxxx	xxxx	xxxx
PF	0.284	0.553	1.107	2.234	4.516
Our	0.113	0.227	0.461	0.907	1.852

Table 5: Running times of Algorithm phase for RMF_LONG instances

N/m M/m	0.3 1.3	0.5 2.6	1.0 5.1	2.1 10	4.1 20
$Dinic$	1.493	2.857	5.704	12.731	xxxx
HI	0.120	0.354	0.729	1.487	0.823
PF	0.295	1.125	0.332	10.970	26.601
Our	0.089	0.216	0.464	0.966	1.929

Table 6: Running times of Reading phase for RMF_LONG instances

N/m M/m	0.3 1.3	0.5 2.6	1.0 5.1	2.1 10	4.1 20
$Dinic$	—	—	—	—	—
HI	2.793	5.662	11.825	25.350	54.612
PF	1.489	2.924	6.011	12.138	24.049
Our	0.789	1.547	3.139	6.387	12.853

Table 7: Running times of Algorithm phase for RMF_WIDE instances

N/m M/m	0.13 0.66	0.3 1.3	0.5 2.6	1.0 5.2	2.1 10	4.2 20
$Dinic$	1.163	2.45	4.771	9.415	23.015	xxxx
HI	0.097	0.234	0.355	1.166	2.268	2.49
PF	0.066	4.386	0.248	42.06	0.905	346.3
Our	0.118	0.252	0.573	1.207	2.523	5.456

Table 8: Running times of Reading phase for RMF_WIDE instances

N/m M/m	0.13 0.66	0.3 1.3	0.5 2.6	1.0 5.2	2.1 10	4.2 20
$Dinic$	—	—	—	—	—	—
HI	1.351	2.711	5.756	12.00	25.196	52.87
PF	0.766	1.476	3.082	6.278	12.551	25.85
Our	0.417	0.805	1.682	3.391	6.779	13.96

Table 9: Running times of Algorithm phase for WASH_LINE_MODULAR instances

N/k M/m	16 0.5	32 1.4	65 4.2	130 12	260 33.6
$Dinic$	1.580	4.960	17.820	67.900	xxxx
HI	0.031	0.088	0.262	0.796	2.468
PF	0.008	0.019	0.044	0.110	0.294
Our	0.025	0.068	0.232	0.663	1.929

7 Conclusion and Future Work

The objective of *k*-vertex connectivity between two vertices s, t is to know the least number of vertices chosen from *V* \ *{s, t}*, whose deletion from the graph would

Table 10: Running times of Reading phase for WASH_LINE_MODULAR instances

N/k M/m	16 0.5	32 1.4	65 4.2	130 12	260 33.6
$Dinic$	—	—	—	—	—
HI	0.782	2.415	7.583	24.460	84.261
PF	0.500	1.395	4.010	11.562	42.070
Our	0.279	0.785	2.270	6.428	18.621

Table 11: Running times of Algorithm phase for RANDOM instances

N/k M/m	2.0 $0.049k$	1 $0.054k$	8.1 3.3	4.1 8.4
$Dinic$	0.299	0.320	181.158	2158.250
HI	0.002	0.002	0.157	0.366
PF	0.002	0.001	0.134	0.002
Our	0.001	0.001	0.096	0.360

Table 12: Running times of Reading phase for RANDOM instances

N/k M/m	2.0 0.049	1 0.054	8.1 3.3	4.1 8.4
$Dinic$	—	—	—	—
HI	0.052	0.054	4.452	10.294
PF	0.047	0.049	3.217	7.766
Our	0.023	0.023	1.633	4.010

destroy every path between s and t. Our work is of significance because it was widely accepted until now that Dinic algorithm was the fastest algorithm, because of its low time complexity at the *k*-vertex connectivity problem, but the other algorithms; push-relabel and pseudoflow outperform it in practice.

This paper introduces a new algorithm for *k*-vertex connectivity in *Õ(m)* time complexity, and shows that its running time is the best for most problems and is close to that of the others implementations in the other problems family, on the other hand, our algorithm outperforms all of the three algorithms if we include the reading phase of each algorithms, because the adaptation of Max-Flow/Min-Cut algorithms to solve *k*-vertex connectivity problem multiplies the number of vertices and edges by two, which leads also to more memory consumption. As part of our future research, we will use BFS just one time along with some heuristic functions to speed up our algorithm implementation, because the BFS is considered expensive to perform it for every vertex-disjoint path in 'Our' algorithm.

References

[1] Boykov, Y. and Kolmogorov, V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137, 2004.

[2] Chandran, B. G. and Hochbaum, D. S. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009.

[3] Cherkassky, B. V. and Goldberg, A. V. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[4] Dinic, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

[5] Esfahanian, A.-H. and Hakimi, S. L. On computing the connectivities of graphs and digraphs. *NETWORKS*, 14:355–366, 1984.

[6] Even, S. Graph algorithms. *Computer Science Press*, 1979.

[7] Even, S. and Tarjan, R. E. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[8] Gabow, H. N. Using expander graphs to find vertex connectivity. *J. ACM*, 53(5):800–844, 2006.

[9] Galil, Z. Finding the vertex connectivity of graphs. *SIAM J. Comput.*, 9(1):197–199, 1980.

[10] Galil, Z. and Italiano, G. F. Reducing edge connectivity to vertex connectivity. *ACM SIGACT News*, 22:57–61, 1991.

[11] Goldberg, A. V. and Tarjan, R. E. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[12] Henzinger, M. R., Rao, S., and Gabow, H. N. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.

[13] Hochbaum, D. S. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.

[14] Johnson, D. S. and McGeoch, C. C. Network flows and matching: First dimacs implementation challenge. *AMS*, pages 1–18, 1993.

[15] Kleitman, D. J. Methods for investigating connectivity of large graphs. *IEEE Trans.Circuit Theory.*, 16(2):232–233, 1969.

[16] Lerchs, H. and Grossmann, I. F. Optimum design of open-pit mines. *Trans., Canadian Inst. Mining and Metallurgy*, 68:17–24, 1965.

[17] Menger, K. Zur allgemeinen kurventheorie. *Fund. Math*, 10:96–115, 1927.

[18] Sankowski, P. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126, 2007.

[19] Setubal, J. C. New experimental results for bipartite matching. In *NETFLOW93*, pages 211–216, 1993.

[20] Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[21] Yoshida, Y. and Ito, H. Property testing on k-vertex-connectivity of graphs. *Algorithmica*, 62(3-4):701–712, 2012.