Limited Maximum Flow Problem With Multiple Sink Vertices

Anonymous
Faculty of Electronic and Information Engineering
Xi'an Jiaotong University
Xi'an, China
**********@stu.xjtu.edu.cn

***Abstract*—We propose a stochastic algorithm for limited maximum flow problem with multiple sink vertices in directed acyclic graph, which is completely different from the traditional network flow algorithms. The algorithm shows significant performance advantages when computing the maximum flow for all vertices. Compared to the Dinic algorithm solution with complexity $O(nm^{\frac{3}{2}})$, our algorithm can optimize the complexity to $O(mk^2)$ with considerable theoretical correctness. In practice, its performance and correctness are better than the currently known bounds.**

***Index Terms*—graph theory, network flow, directed acyclic graph, linear basis, hash**

I. Introduction

A. Network Maximum Flow

The network maximum flow [1] is always an important problem in the study of graph theory and algorithms, which not only has some applications in traffic allocation and routing in networks, but also can be reduced from problems such as bipartite graph matching, minimum cut, task allocation, linear programming, etc. to solve difficult problems in other fields.

Definition 1. *Directed Graph. A directed graph $G = (V, E)$ consists of a vertex set V and an edge set E.*

The elements of $E \subseteq V^2$ are ordered pairs of vertices. The conventions $n = |V|, m = |E|$ are the number of vertices and edges of the graph, respectively.

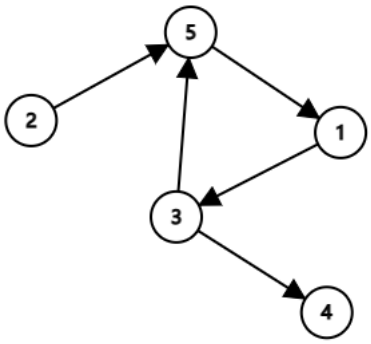

Fig. 1. A graph with a directed cycle.

Definition 2. *Directed Acyclic Graph. A directed acyclic graph is a directed graph without any directed cycle.*

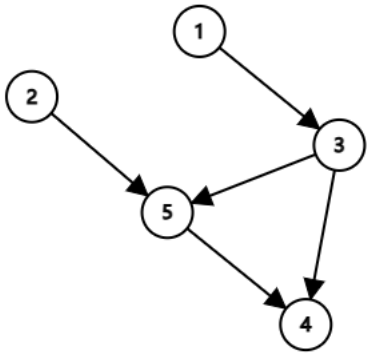

Fig. 2. A directed acyclic graph.

Definition 3. *Network Maximum Flow. Given a network (graph) $N = (V, E)$ with a source vertex $s \in V$, a sink vertex $t \in V$ and capacity for each of its edge $e \in E$, the maximum flow is the maximum value among all its flow from s to t.*

Formally, the capacity of edges is a mapping $c: E \to R^+$, and $c(e)$ is the maximum flow allowed over some edge $e \in E$.

A flow is a mapping $f: E \to R^+ \cup \{0\}$ satisfying the following restrictions:

- Capacity constraint: $f(e) \leq c(e), \forall e \in E$;
- Conservation of flows: $\forall v \in V \backslash \{s, t\}$,

$$\sum_{u:(u,v)\in E} f(u, v) = \sum_{u:(v,u)\in E} f(v, u).$$

The value of flow $|f|$, is the flow from the source to the sink:

$$|f| = \sum_{v:(s,v)\in E} f(s, v) = \sum_{u:(u,t)\in E} f(u, t).$$

Finally, the maximum flow f_{max} is the maximum value among all its flow:

$$f_{max} = \max |f|$$

In this paper, we only discuss networks with a constant capacity of 1. For edges with $c(u, v) \neq 1$, we may show that

replacing them with $c(u, v)$ number of multiple edges with capacity 1 does not affect the answer to the maximum flow.

Definition 4. *Limited Maximum Flow. An additional parameter $k > 0$ is specified for the network flow problem, and then it asks to compute the maximum flow of the network, or report that the maximum flow is at least k. Typically k is small, e.g., $k \leq 50$.*

Formally, the problem asks the value of:

$$F = \min\{f_{max}, k\}.$$

B. Network Maximum Flow with Multiple Sink Vertices

We note that in quite a few applications of network flows [2], we need to not only compute the maximum flow for a unique pair (s, t), e.g. in network routing, sometimes we want to know the maximum flow from the current vertex to every other vertex, and then we can estimate the channel capacity from the current vertex to all vertices accordingly.

However, computing the maximum flow for multiple source-sinks simultaneously is more difficult, even if the computational requirements have the same source or sink. Well-known network flow algorithms almost exclusively focus on solving maximum flow problem for unique (s, t) pairs, and rarely generalize the problem to computing the maximum flow when each vertices acts as a sink. The only way to compute the maximum flow between q pairs of source-sinks is to run the above algorithm $\Omega(q)$ times, whose time complexity is usually unsatisfactory. We can also note that the computation of a lot of flow information is repeated during the process, which is wasteful in terms of time and space overhead.

C. Our result

For the limited maximum flow problem with multiple sink vertices, we propose a Monte Carlo algorithm with a high accuracy and a time complexity of $O(mk^2)$, where k is the additional parameter. When k is small, our algorithm can exhibit much higher performance than running n traditional network flow algorithms directly.

The main idea of the algorithm is to hash the streams using vectors and to further optimize the algorithm using linear bases; to ensure sufficient accuracy and to make the correctness estimable, we use the multidimensional vectors in a finite field of modulo prime residue classes. We give a description of the algorithm in Section III and a proof of its complexity and correctness in Section IV. In Section V, we perform comparison experiments to validate our algorithm.

II. Related Works

A. Ford-Fulkerson Augmentation

Ford-Fulkerson augmentation [1] is a general term for a class of algorithms that compute maximal flows. The algorithm computes the maximum flow by greedily finding augmentation paths.

For a given graph G and flow f, The algorithm defines a residual network G_f whose capacity of each edge is the original capacity minus the flow of the corresponding edge in f. Thus, the process of finding flows with maximum value can be reduced to continuously searching for augmented paths in the residual network, i.e. paths that go from the source to the sink and have a non-zero flow through the route paths. The algorithm also introduces flow withdrawal as a backtracking greedy method. It can be shown that after the introduction of flow withdrawal, the augmentation path cannot be found if and only if the flow of the current stream has reached the maximum value. Therefore, it is sufficient to define the initial flow f as an empty flow and iteratively search for augmentation paths, updating the flow scheme and the residual network, until no new augmentation paths can be found.

The complexity of the algorithm varies due to the different implementations, and the following are the two main implementations.

a) Edmonds-Karp Algorithm: Intuitively, to implement Ford-Fulkerson augmentation, it is sufficient to find augmentation paths in a residual network using BFS (breadth-first search) and further assign flow to each edge in the found augmentation paths [3]. In this algorithm, the upper bound on the complexity of a single augmentation is $O(m)$, while the upper bound on the number of rounds of augmentation is $O(nm)$, and consequently the upper bound on the theoretical complexity is $O(nm^2)$.

Running the algorithm multiple times to solve a limited maximum flow problem with multiple sink vertices, has a complexity of $O(n^2m^2)$.

b) Dinic Algorithm: This implementation [4] introduces multi-path augmentation and current arc optimization to reduce the upper bound on the number of rounds for Ford-Fulkerson augmentation.

When performing BFS, the algorithm layers the residual network and subsequently performs multi-path augmentation using DFS (depth-first search) to compute the maximum augmented flow on that layered graph. During DFS, the algorithm maintains a pointer to the earliest edge for which flow currently exists at each vertex, using current arc optimization to ensure that full-flow edges are not visited repeatedly. Finally the algorithm is able to optimize the upper bound on the number of rounds of augmentation to $O(n)$ at the cost of the upper bound on the complexity of augmentation becoming $O(nm)$.

However, the running time of the algorithm does not reach the theoretical upper bound in practice, and it can be shown to have even better complexity on some well-characterized graphs. For example, on networks of unit capacity, its complexity improves further to $O(m \min\{m^{\frac{1}{2}}, n^{\frac{2}{3}}\})$. The complexity is thus about $O(nm^{\frac{3}{2}})$ when using this algorithm to solve a limited maximum flow problem with multiple sink vertices.

B. Push-Relabel Algorithm

The push-relabel algorithm [5] solves the maximum flow by performing update operations on individual vertices until there are no vertices to update.

The flow function maintained by the algorithm does not necessarily maintain flow conservation; for a vertex that is not a source or sink, we allow the flow into the vertex to exceed the flow out of the vertex, and the exceeding portion is referred to as the excess flow of such vertex u, noted as $e(u)$; the algorithm maintains the height of each vertex, $h(u)$, and specifies that the vertex with the excess flow can only push the flow to a vertex whose height is less than u (Push); if it is not possible to push the flow, then re-modify the vertex height (Relabel).

a) HLPP Algorithm: The HLPP algorithm adds the following restriction to the generic push-relabel algorithm: each time a vertex is selected, the overflow vertex with the highest height is preferred. The upper bound on the complexity of this algorithm is $O(n^2\sqrt{m})$. The complexity is $O(n^3\sqrt{m})$ when using this algorithm to solve a limited maximum flow problem with multiple sink vertices.

C. Other Cutting-edge Algorithms

a) Maximum Flow in Almost-linear Time: Li Chen et al [6] proposed an algorithm that can compute the minimum cost while solving the maximum flow and has almost linear time complexity. The complexity is at least $O(n^2)$ when using this algorithm to solve a limited maximum flow problem with multiple sink vertices.

b) Solution Using Expander Graphs: Cheung et al [7] proposed an algorithm that is similar to our approach and optimized the complexity to $O(mk^{\omega-1})$ using expander graph and matrix multiplication techniques, where $\omega \approx 2.37$ is the matrix multiplication exponent.

III. Main Algorithm

A. Naive Algorithm

The main idea of the algorithm is to hash the flow using k-dimensional vectors. In the implementation of the algorithm, in order to ensure the decidability of the zero vector and the accurate computation of the rank, we choose the finite field $\mathbb{Z}_p$ as the components of vectors, i.e. $\mathbf{e} \in \mathbb{Z}_p^k$ holds for every vector $\mathbf{e}$, where p is a prime number chosen in advance.

During the algorithm, we first perform topological sort on the directed acyclic graph and process the vertices and edges in corresponding order. For each edge in the graph, we compute the hash vector corresponding to it according to the following:

1) For an edge $(s, u) \in E$ from s, its hash vector $\mathbf{e}_{su}$ is selected uniformly at random in $\mathbb{Z}_p^k$;
2) For other edges $(u, v) \in E$, the hash vector $\mathbf{e}_{uv}$ is selected uniformly at random in the in-edges space of u, E_u.

where the in-edges space of a vertex is defined as the linear space spanned by the hash vectors of all the in-edges of the vertex:

$$E_u = span\{\mathbf{e}_{vu} : (v, u) \in E\}.$$

We may consider $E_s = \mathbb{Z}_p^k$ to be a special case of the in-edges space, therefore the rules for the computation of hash vectors can be uniformly given by rule 2. In particular, if a vertex u has no in-edges, then E_u is a zero vector space.

After computation of all hash vectors, we report $\dim(E_u)$ as the answer at vertex u, i.e. we consider that:

$$\dim(E_u) = F_u$$

holds with high probability. Here F_u is the answer of the limited maximum flow problem when the sink is set to u.

Since our algorithm runs on a directed acyclic graph, the computation of hash vectors of the in-edges for each vertex must have been completed by the time it is processed in topological sort, so we can immediately compute the in-edges space for that vertex and then compute the hash vectors of all out-edges of it. Therefore, we only need to process each vertex and each edge according to the topological order to get the approximate answer for all vertices:

Algorithm 1 Hash Vector Algorithm

Input: A directed acyclic network $G = (V, E)$ with unit capacity, and the source vertex s
Output: The limited maximum flow $F(u)$ for each vertex $u \in V \setminus \{s\}$
Initialize queue Q as empty;
for each vertex $u \in V$ **do**
  $d(u) \leftarrow |\{v|(v, u) \in E\}|$; {In-degree of vertex u}
  if $d(u) = 0$ **then**
    Add u to the end of queue Q;
  end if
end for
while Q is not empty **do**
  $u \leftarrow$ front element of Q; {Dequeue the front element}
  Remove the front element from Q;
  if $u = s$ **then**
    $E_u \leftarrow \mathbb{Z}_p^k$;
  else
    $E_u \leftarrow \text{span}\{\mathbf{e}_{vu} : (v, u) \in E\}$; {Span of in-edges}
  end if
  $F(u) \leftarrow \dim(E_u)$; {Dimension of E_u}
  for each $v : (u, v) \in E$ **do**
    $\mathbf{e}_{uv} \leftarrow$ a uniformly random vector from E_u;
    $d(v) \leftarrow d(v) - 1$; {Decrement in-degree of v}
    if $d(v) = 0$ **then**
      Add v to the end of queue Q;
    end if
  end for
end while

The relationship between our naive algorithm and the maximum flow of the network is not obvious. The following sections further illustrate the connection by proving some simple properties of hash vectors and in-edges spaces.

B. Unreachable vertex

We first consider excluding the effects of unreachable vertices.

Theorem 1. *For all vertices t which is unreachable form s,*

$$\dim(E_t) = F_t = 0$$

holds.

Proof. Since there does not exist any path from s to t, it is clear that there is a maximum flow of 0. The following proof of $\dim(E_t) = 0$ is inductive by topological order.

For an unreachable vertex t with no in-edges, its in-edges space is a zero vector space by definition.

For some unreachable vertex t, if $\forall u : (u,t) \in E, u$ is unreachable, and $\dim(E_u) = 0$ holds:

Then there must be $\mathbf{e}_{ut} = 0$. And since the unreachable vertex t cannot have a reachable predecessor vertex, E_t is also a zero vector space by definition, thus $\dim(E_t) = 0$. □

Corollary 1. *For all edges (u,v) which is unreachable from s, $\mathbf{e}_{uv} = 0$ holds.*

It follows immediately from the fact that the starting vertex u of an unreachable edge must be an unreachable vertex, combined with Theorem 1.

Theorem 1 shows that our algorithm is always correct for any vertex that is unreachable from s.

Since the zero vector does not affect the in-edges spaces for the other vertices, we can ignore all vertices unreachable from s and their out-edges in the discussion that follows, and they have no effect on the answers for the remaining vertices.

C. Minimum Cut

Minimum cut [1] is the dual of the maximum flow problem and has many good properties related to maximum flow. To facilitate further proofs, we introduce here the definition of minimum cut and prove some properties related to it.

Definition 5. *Minimum Cut. The minimum cut of a graph G is the minimum value among capacities of all cuts for the graph.*

where a cut of graph G is a partition on vertices, $\{S,T\}$, where $T = V \setminus S$, and $s \in S, t \in T$ holds; the capacity of cut, $c(S,T)$ is the sum of capacities of all the edges from S to T, i.e:

$$c(S,T) = \sum_{u \in S, v \in T} c_{uv}.$$

Lemma 1. *Maximum Flow Minimum Cut Theorem. For any graph G with source s and sink t, it holds that $f_{max} = c_{min}$.*

Proof. Firstly, we prove that $f_{max} \leq c_{min}$.

For some minimum cut scheme $\{S,T\}$, since $s \in S, t \in T$, due to flow conservation it can be obtained that f_{max} is the flow from S to T. By the capacity limitation $f_{uv} \leq c_{uv}$, we have:

$$\begin{aligned} f_{max} &= \sum_{u \in S, v \in T} f_{uv} - \sum_{u \in S, v \in T} f_{vu} \\ &\leq \sum_{u \in S, v \in T} c_{uv} \\ &= c_{min}. \end{aligned}$$

Then we proof that $f_{max} \geq c_{min}$.

For some maximum flow scheme of f_{max}, there must be no augmentation paths on its residual network, i.e. there does not exist a non-zero flow path from s to t that satisfies the capacity constraint. Let S be all the vertices reachable from s on the residual network, then $\{S, V \setminus S\}$ must be a cut; and for any edge (u,v) in the cut, since u is reachable and v is not, it can be known that this edge is full, i.e. $f_{uv} = c_{uv}, f_{vu} = 0$. From the conservation of flow, we obtain:

$$\begin{aligned} f_{max} &= \sum_{u \in S, v \in T} f_{uv} - \sum_{u \in S, v \in T} f_{vu} \\ &= \sum_{u \in S, v \in T} f_{uv} \\ &= \sum_{u \in S, v \in T} c_{uv} \\ &\geq c_{min}. \end{aligned}$$

□

Lemma 2. *For any cut $\{S,T\}$ of any sink t and any edge $(u,v) \in E$ satisfying $u, v \in T$, $\mathbf{e}_{uv}$ can be linearly represented by the hash vectors of the cut edges.*

Proof. We proof by induction on topological order.

For a vertex u whose in-edges are all cut edges, it is clear that any vector in E_u can be linearly represented by the hash vectors of the set of its in-edges, plus the set of its in-edges is a subset of the cut edges, so that any outgoing edge of u must satisfies that $\mathbf{e}_{uv} \in E_u$, and it can be linearly represented by the hash vectors of the full set of cut edges.

For a vertex u, if all its in-edges $(v,u) \in E, v \in T$ from the interior of T satisfies that $\mathbf{e}_{vu}$ is linearly representable by the hash vectors of the cut edges, since any vector in E_u can be linearly represented by the hash vectors of u's in-edges, which consists of cut edges and edges from the interior of T, both can be linearly represented by the vector set of cut edges; therefore, any out-edge of u must also be linearly representable by the hash vectors of the full cut edges. □

Theorem 2. *The rank of the in-edges space does not exceed the maximum flow. i.e. for any sink t, $f_{max} \geq \dim(E_t)$.*

Proof. By Lemma 1, the maximum flow is equal to the minimum cut, so it suffices to prove $c(S,T) \geq \dim(E_t)$ for the minimum cut scheme $\{S,T\}$.

Consider the vector set consisting of the hash vectors of the cut edges in the minimum cut, and from Lemma 2, the elements of the in-edges vectors of t that do not belong to the vector set above can be linearly represented by the vector set,

and thus for the unit capacity graph we have:

$$\begin{aligned} f_{max} &= c_{min} \\ &= \sum_{u\in S, v\in T} c_{uv} \\ &= |\{\mathbf{e}_{uv} : u \in S, v \in T, (u,v) \in E\}| \\ &\geq rank\{\mathbf{e}_{uv} : u \in S, v \in T, (u,v) \in E\} \\ &\geq rank\{\mathbf{e}_{ut} : (u,t) \in E\} \\ &= \dim(E_t). \end{aligned}$$

□

The above theorem shows that $\dim(E_t)$ is a lower bound for f_{max} at t. Intuitively, $\dim(E_t) = k'$ implies that there exist at least k' in-edges vectors of t that are linearly independent, which in turn, by Lemma 2, shows that these vectors must belong to different flow paths in some flow scheme.

Note that when generating vectors uniformly in the space E_u of a vertex u, any vector sets with less than $\dim(E_u)$ vectors have a high probability of being linearly independent. When the out-edges are less than $\dim(E_u)$, the major limitation on the dimension of the subsequent out-edges space is the number of out-edges of u; otherwise the major limitation is $\dim(E_u)$. This corresponds to the fact that when the maximum flow of u is f_{max}, we can assign the flow to any f_{max} or less number of out-edges. When the out-edges are less than f_{max}, the major limitation of the subsequent maximum flow is the number of out-edges of u; otherwise, the major limitation is maximum flow of u. In addition, since the computation of $\dim(E_u)$ only needs the local information of u, it can be computed directly by topological order without considering the allocation scheme of the subsequent flows at the current vertex, so it has a significant performance advantage.

Thus, we may consider the hash vector presented in the algorithm as defining a hash of flow schemes up to the limitation k. Except for some collision with little probability, we have compressed multiple flow allocation schemes into the hash vector of edges, so we needn't select the edges and paths for the flow scheme in computation, and when computing the dimensionality of the vector space, we can immediately find out the value of the maximum flow up to k at the current vertex.

In Section IV-B we will strictly prove the correctness of computing limited maximum flows with hash vectors.

IV. Performance Analysis

A. Complexity Analysis

In the naive algorithm from Section III-A, we have given the pseudo-code for the implementation of the algorithm; however, the pseudo-code is still vague in the way some details are handled, such as how to represent and manage E_u, how to compute $\dim(E_u)$, and how to generate the successor hash vectors in E_u uniformly at random. In some even more intuitive implementations, these processes can become bottlenecks in the algorithm and finally fail to yield a better time complexity; therefore, we first need to introduce a data structure that can efficiently maintain a multidimensional linear space.

a) Linear Basis: A linear basis [8] is a maximal linearly independent set in a linear space. We can use a greedy strategy to maintain a set of bases of a linear space so that it can efficiently perform the vector operations in the algorithm. The greedy strategy is specified as follows:

Whenever a new vector is added to the basis, the first non-zero component of the vector is considered: if no vector exists in the basis with that component as the first non-zero one, the vector is added to the basis and then the process terminated; otherwise, the first non-zero component of the vector is eliminated with the vector found, generated a new vector with zero at the mentioned component, using the linear combination of the two vectors. The newly generated vectors are considered recursively until the vector is added to the basis or the vector is a zero vector.

We can note that the space complexity of the linear basis is $O(k^2)$, and there are at most k rounds of greedy maintenance when inserting a new vector, and at each round we need to find the inverse element required for the elimination and perform $O(k)$ operations to update the current vector. Here it is sufficient to compute the inverse of $a^{p-2} \bmod p$ using fast exponentiation according to Fermat's Little Theorem, with a total complexity of $O(k(k + \log(p-2))) = O(k^2)$. Since $\dim(E_u)$ is the number of linearly independent vectors in the linear basis, it can also be obtained by $O(1)$ immediately after adding all in-edges vectors at vertex u to the linear basis; since the vectors in E_u are one-to-one corresponded with linear combinations of linear bases, it only requires to uniformly randomize the coefficients of vectors in the linear basis, then the linear combination of which is a uniformly random vector in E_u. The time complexity of obtaining a random vector in space is also $O(k^2)$. In addition, the time complexity of clearing the linear basis is also at most $O(k^2)$.

b) Improved Algorithm: With the use of linear basis, we can obtain some improved algorithms with excellent complexity. An intuitive implementation is to maintain a linear basis for each vertex, and once we generated a hash vector, we immediately add it to the linear basis of the succeeding vertices. In this way, we have automatically obtained E_u each time we take out vertex u as the front element of queue, and thus also directly obtain $F(u)$ and can randomize the successor vectors directly. For s, it is straightforward to add k standard orthogonal bases to E_s at the beginning.

The initialization of $d(u)$ can be done in $O(n+m) = O(m)$ time using common structures such as chain forward stars to maintain the graph. Since each vector will only be generated once and added to the linear basis once, the complexity of processing the vertices in topological order is $O(n + mk^2) = O(mk^2)$ and the total time complexity is $O(mk^2)$.

Note that maintaining a linear basis for each vertex achieves a space complexity of $O(nk^2)$ which may become a bottleneck in performance. Therefore, we consider the following further optimization:

Processing u in topological order reuses a linear basis, enumerating the in-edges of u to compute E_u, and for E_s we still use the standard orthogonal basis; $\dim(E_u)$ can then be computed to generate the successor vectors. A similar analysis also gives a time complexity of $O(mk^2)$, while the space complexity becomes $O(k^2 + mk) = O(mk)$.

We thus obtain a final algorithm with $O(mk)$ space complexity and $O(mk^2)$ time complexity, which performs significantly better than traditional network flow algorithms on the limited maximum flow problem with multiple sink vertices.

The following Table I provides an intuitive comparison.

TABLE I
TIME COMPLEXITY COMPARISON

Algorithm	Performance	
	Complexity	$\mathbf{n = 10^5, m = 2 \times 10^5, k = 50}$
Ours	$\mathbf{O(mk^2)}$	$\mathbf{5 \times 10^8}$
EK	$O(n^2m^2)$	4×10^{20}
Dinic	$O(nm^{\frac{3}{2}})$	8.94×10^{12}
HLPP	$O(n^3\sqrt{m})$	4.47×10^{17}

B. Correctness Analysis

This section analyses the correctness of the algorithm.

Lemma 3. *The probability that some k' number of out-edges $(k' \leq k)$ of s are linearly independent is*

$$\prod_{i=k-k'+1}^{k} (1 - \frac{1}{p^i}).$$

Proof. Since each vector is independently randomly generated from $\mathbb{Z}_p^k$, we may assume that the above vectors are randomly generated sequentially. Consider the probability of the following event: the i-th vector $\mathbf{e}_i$ is randomly generated conditioned on that the first $i-1$ vectors linearly independent, and the newly generated vector can be linearly represented by the previous vectors.

From the previous set of linearly independent vectors we can obtain a basis $\{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k\}$ of $\mathbb{Z}_p^k$ by extending the vector set, where $\mathbf{b}_j = \mathbf{e}_j, \forall j < i$. If we randomly select some coefficients $\theta_j \in \mathbb{Z}_p$ on the basis, then the different sets of $\{\theta_j\}$ corresponds to each the vectors in $\mathbb{Z}_p^k$ one-to-one, so the process of randomly generating $\mathbf{e}_i$ is equivalent to independently and uniformly randomly generate $\theta_j \in \mathbb{Z}_p$ and let $\mathbf{e}_i = \sum_{j=1}^{k} \theta_i \mathbf{b}_j$.

Note that $\mathbf{e}_i$ is linearly dependent with $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{i-1}\}$ if and only if $\forall j \geq i, \theta_j = 0$, so the probability that the mentioned event occurs is $\frac{1}{p^{k-i+1}}$. The probability of linear independent for some k' vectors is thus the probability of linear independent of the first $k' - 1$ vectors multiplies $(1 - \frac{1}{p^{k-k'+1}})$. Thus some simple induction is sufficient to prove the lemma. □

Theorem 3. *For any vertex t,*

$$\Pr[\dim(E_t) = F_t] \geq (1 - \frac{1}{p})^m \prod_{i=1}^{k} (1 - \frac{1}{p^i})$$

holds.

Proof. Consider the intuitive meaning of F_t, $F_t = k'$ implies that there exists a flow of size $k' \leq f_{max}$, on a unit capacity network, i.e. there are $k' \leq k$ disjoint paths from s to t. Let's denote the hash vectors of all edges in the path as follow:

The vectors of the first path are denoted as $\mathbf{e}_{11}, \mathbf{e}_{12}, \ldots, \mathbf{e}_{1m_1}$ respectively, and vectors of the second are denoted as $\mathbf{e}_{21}, \mathbf{e}_{22}, \ldots, \mathbf{e}_{2m_2}$, ... , vectors of the k'-th path are denoted as $\mathbf{e}_{k'1}, \mathbf{e}_{k'2}, \ldots, \mathbf{e}_{k'm_{k'}}$, where m_i is the length of the i-th path. Note that $\dim(E_t) = \dim(span\{\mathbf{e}_{im_i}\})$.

Let's consider the following process:

Maintain a vector set $\{\mathbf{e}_{ip_i}\}$, initially with $p_i = 1, \forall 1 \leq i \leq k'$. Each time we choose an arbitrary j such that $p_j < m_j$ and let $p'_i = p_i + [i == j]$, change the vector set to $\{\mathbf{e}_{ip'_i}\}$. The process ends when $\forall i, p_i = m_i$.

Then the probability that $\dim(E_t) = F_t = k'$ is at least the probability that $rank\{\mathbf{e}_{ip_i}\} = k'$ always holds for the process. For the maintenance, since each vector is chosen independently at random, the probability above is the probability of $rank\{\mathbf{e}_{i1}\} = k'$ multiplies the probability of rank conservation at each update. Lemma 3 shows that the probability of the initial condition is at least $\prod_{i=1}^{k} 1 - \frac{1}{p^i}$.

Now let's consider the probability of $rank\{\mathbf{e}_{ip'_i}\} = k'$, conditioned on $rank\{\mathbf{e}_{ip_i}\} = k'$. We may denote the vertex that pointed by edge of $\mathbf{e}_{jp_j}$ as u, then the update is equivalent to substituting $\mathbf{e}_{jp_j}$ with a uniformly random vector, i.e. $\mathbf{e}_{jp'_j}$, from E_u.

Considering the subspace $E' = span\{\{\mathbf{e}_{ip_i}\} \backslash \{\mathbf{e}_{jp_j}\}\} \cap E_u$, we can see that $\mathbf{e}_{jp'_j}$ is linearly dependent to the other vectors if and only if $\mathbf{e}_{jp'_j} \in E'$. By the fact that $\mathbf{e}_{jp_j} \in E_u$ can't be linearly represented by other vectors, it is clear that $\dim(E') < \dim(E_u)$, and because generating $\mathbf{e}_{jp'_j}$ is equivalent to generating coefficients randomly on a basis of E_u, we may assume that this basis is extended from some basis on E', then there is at least one extended base vector; at this point, in order to make the generated new vector $\mathbf{e}_{jp'_j}$ to be linearly dependent to the other vectors, all the linear combination coefficients of the extended basis vectors must be zero, with probability at most $\frac{1}{p}$. i.e. the probability that rank conservation holds for the update is at least $(1 - \frac{1}{p})$.

In summary, multiplying the probabilities of the independent events gives the probability that $rank\{\mathbf{e}_{ip_i}\} = k'$ always holds is:

$$(1 - \frac{1}{p})^m \prod_{i=1}^{k} (1 - \frac{1}{p^i})$$

and it must be a lower bound of $\Pr[\dim(E_t) = F_t]$. □

Corollary 2. *The probability that the answer of all vertices are correct is at least*

$$1 - n(1 - (1 - \frac{1}{p})^m \prod_{i=1}^{k} (1 - \frac{1}{p^i}))$$

It follows immediately from Union Bound of Theorem 3.

Corollary 2 shows that the algorithm we propose has a high accuracy and it can be improved to any precision by enlarge

p or by running the algorithm multiple times and taking the maximum answer for each vertex, and is therefore also quite scalable.

V. Experiments

The comparison in Table I shows that the traditional network flow algorithm that gives the best complexity on this problem is Dinic. In this section, we implement our algorithm and Dinic, and a data generator to generate the data, and then compare their performance on an online judge system to evaluate the improvement of our algorithm. Also, by comparing the final answers given by the two algorithms, we can check the correctness of our algorithm.

A. Setup

We set a time limit of 15000ms. If it takes too long for a test, a *Time Limit Exceeded (TLE)* is returned in the evaluation.

To verify the efficiency of the algorithm on various graphs and to test the correctness of the algorithm, we constructed four different types of data:

1) **Random Graph.** Randomly add edges between two different vertices. To ensure that there are no loops in the graph, it is sufficient to let the directed edge always points to the vertex with a greater index number than the other. Random graphs are the easiest to implement and are commonly used construction in graph theory.
2) **Hierarchical Graph.** Given the parameters a, b, divide into a layers of b vertices each and add edges between neighbouring layers. Graphs constructed in layers can have better network flow properties and thus can provide tests with higher intensity.
3) **Special Construction.** Construct data that is unfavourable to the hash vector algorithm. Split into two major paths from the source, one of which will have a smaller capacity, but will eventually have a large number of edges connected to a vertex, while the other will have a slightly larger capacity and be the correct scheme for the maximum flow.
4) **Random Hierarchical Graph.** Combining random and hierarchical graphs to construct test data with even higher intensity.

The detailed setup for each test is specified in Table II.

TABLE II
Data generation setup

No.	Type	n	m	k	Remark
1	1	20	100	10	/
2	2	26	50	50	5 layers
3	1	1000	3000	20	/
4	2	1001	3000	50	50 layers
5	1	10000	20000	25	/
6	2	10001	20000	50	200 layers
7	4	10001	20000	50	200 layers
8	1	100000	200000	30	/
9	2	99951	199900	50	1999 layers
10	4	99900	199810	50	999 layers
11	3	100000	200000	50	/

B. Result Analysis

The results of the performance comparison experiments are shown in Fig. 3:

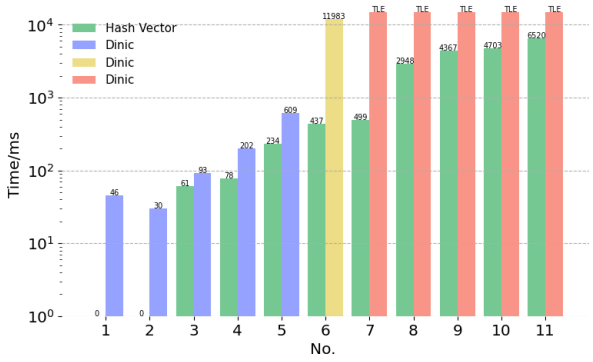

Fig. 3. Performance Comparison

We can see that the Dinic algorithm only passes tests with small data size on all types of graphs, even though Dinic's performance in practice is superior to the theoretical complexity. By contrast, our algorithm has a significant performance advantage and passes large-scaled tests.

The pairwise comparison was then conducted, running our algorithm and Dinic for solving the four types of medium-scaled data to compare the correctness of the answers. Our algorithm obtained correct answers in all tests during the pairwise comparison script running for more than 10 hours.

VI. Conclusion and Future Works

In this paper, we propose an algorithm for computing limited maximum flow in directed acyclic graphs with multiple sinks, which is quite different from traditional network flow algorithms. When computing the maximum flow of a network with multiple sinks, our method with a complexity of $O(mk^2)$ is able to significantly outperform other traditional algorithms, such as the most commonly used Dinic algorithm. Although our algorithm introduces randomness and may produce incorrect results, its correctness is still within acceptable limits and has good scalability. We validate the complexity and correctness of our proposed algorithm with both theoretical proofs and comparison experiments.

However, it is important to note that the correctness of our algorithm in practice is significantly better than the known theoretical lower bounds, and we conjecture that further proofs of the algorithm's tighter lower bounds on correctness could be made, for example by considering the correlation of correctness between answers of multiple vertices to further improve the bound from Union Bound.

In addition, the current algorithm only has a intuitive form on directed acyclic graphs, and it is not clear if there is a feasible approach to generate hash vectors for general directed graphs and solving for the network maximum flow using a similar approach. Future work will proceed in this direction.

ACKNOWLEDGMENT

This paper is a coursework of the Paper Writing and Publishing course, and I would first like to thank Prof. Minnan Luo for her guidance. I would also like to thank the Programming Contest Team of Xi'an Jiaotong University, since the topic of this research originated from the hardest problem of the Shaanxi Provincial Contest that we jointly organized. Other students in team also gave me a lot of help during my thinking and research, especially Yeyuan Chen's discussion of the hash vector method and Ziqian Chen's guidance in conducting experiments such as the pairwise comparison verification. Finally, I would also like to thank this coursework for giving me the opportunity to carry out paper writing practice, so that I can devote myself to scientific research more quickly and lay a solid foundation for my future scientific studies.

REFERENCES

[1] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," Can. J. Math., vol. 8, pp. 399–404, 1956.

[2] O. Trabelsi, A. Abboud, L. Georgiadis, and R. Krauthgamer, "Faster algorithms for all-pairs bounded min-cuts," in Proc. 46th Int. Colloq. Automata, Lang., Program. (ICALP), 2019, Art. no. 7, pp. 1–15.

[3] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," J. ACM, vol. 19, no. 2, pp. 248–264, 1972.

[4] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," Sov. Math. Dokl., vol. 11, no. 5, pp. 1277–1280, 1970.

[5] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," J. ACM, vol. 35, no. 4, pp. 921–940, 1986.

[6] L. Chen, R. Kyng, Y. P. Liu, and R. Peng, "Maximum flow and minimum-cost flow in almost-linear time," in Proc. 63rd Annu. IEEE Symp. Found. Comput. Sci. (FOCS), 2022, pp. 123–134.

[7] H. Y. Cheung, L. C. Lau, and K. M. Leung, "Graph connectivities, network coding, and expander graphs," in Proc. 52nd Annu. IEEE Symp. Found. Comput. Sci. (FOCS), 2011, pp. 190–199.

[8] J. Kleinberg and É. Tardos, Algorithm Design. Boston, MA, USA: Addison-Wesley, 2006, pp. 128–130.