Research on Optimization and Problem Reduction of Network Maximum Flow Algorithms

Qixuan Chen, Junqi Mao, Qifan Wang

Abstract

Solving the network maximum flow problem is a significant combinatorial optimization problem and a hot research topic in graph theory. We first provide the relevant symbols and definitions for the maximum flow problem, then summarize the previously proposed maximum flow algorithms and optimizations into two categories. We propose an improved method for the HLPP algorithm, test its performance, and evaluate its applicability. Finally, we introduce several types of problems that can be reduced to the maximum flow problem.

In the maximum flow algorithm research section, we classify maximum flow algorithms into two categories based on their workflow characteristics: augmenting path algorithms and preflow-push algorithms. For augmenting path algorithms, we introduce classical methods represented by the EK algorithm and Dinic's algorithm and summarize their optimization strategies. Then, we introduce the MPM and ISAP algorithms, which are improvements based on Dinic's algorithm, and summarize their optimization methods. In the preflow-push algorithm section, we first summarize the characteristics and general templates of preflow-push algorithms, then focus on the Highest Label Preflow Push Algorithm (HLPP), which is widely recognized in academia as the most efficient method. We introduce two common optimization methods. At the end of the preflow-push section, we discuss and analyze the challenges of improving the HLPP algorithm's efficiency, namely the "ping-pong effect." We introduce and reproduce the Budget Algorithm, which effectively avoids the "ping-pong effect" (source code is not provided in the original paper), and compare the performance of the above algorithms using test data.

At the end of the algorithm research, we summarize recent advances in network flow algorithms through several related papers. We learn from these papers about the existence of approximate linear-time complexity network flow algorithms and present future possible research directions for network flow algorithms.

In the problem reduction section, we summarize four categories of problems that can be reduced to the maximum flow problem: bipartite graph maximum matching, minimum-cost maximum flow, bipartite graph maximum weight matching, and the minimum cut problem. We provide proof of correctness for the reduction and transformation of each type of problem.

In this work, all content, including mainstream algorithm research and problem reduction, is implemented using C++ programs and tested with data. Finally, we include the program code in the appendix.

Keywords: *Network Maximum Flow; Dinic Algorithm; Highest Label Preflow Push Algorithm; Budget Algorithm*

I Introduction

1.1 The Maximum Flow Problem

Given a vertex set $V = \{v_1, v_2...v_n | n < \infty\}$ and an ordered pair set $E = \{e_1, e_2...e_n | n < \infty\}$ consisting of elements from the vertex set, the tuple $G_e = \{V, E\}$ is called a simple directed graph, where the ordered pair set E is called the edge set.

By adding the set $C = \{c_{ij} | i, j <= n\}$ to G_e, where c_{ij} defines the capacity between vertices v_i and v_j, the resulting $G = \{V, E, C\}$ is called a capacity network. Furthermore, a strictly defined capacity network must satisfy the condition that there is exactly one node with an out-degree of 0 (the sink node) and one node with an in-degree of 0 (the source node).

Establishing a mapping $f(u, v)$ from the edge set of the capacity network to the set of real numbers, where $u \in V, v \in V$. If this real function satisfies capacity constraints, anti-symmetry, and flow conservation, this function is called the flow function of the capacity network. The formal definition of the flow function is as follows:

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

$$s.t. \begin{cases} f(u,v) < c(u,v) \\ f(u,v) = -f(u,v) \\ \sum_{(u,x)\in E} f(u,x) = \sum_{(x,v)\in E} f(x,v), \quad \forall x \in V - \{s,t\} \end{cases}$$

The core problem of network maximum flow is: for a capacity network and a pair of source and sink points, find the maximum flow size flowing through the network. That is to say, find

$$M = max f(s,t)$$

1.2 Residual network

Definition 1.1 (Residual capacity) *Let the capacity and flow of the edge with endpoints u, v be $c(u, v)$ and $f(u, v)$ respectively, then $c_f(u, v) = c(u, v) - f(u, v)$ is the residual capacity of the edge.*

Definition 1.2 (Residual network) *The subgraph composed of all nodes and edges with positive residual capacity in the capacity network G is the residual network of G. Formally, the definition of the residual network G_f is:*

$$G_f = (V_f = V, E_f = \{(u,v) \in E, c_f(u,v) > 0\})$$

It is worth noting that the edges with residual capacity greater than 0 may not be in the original graph G, that is, the residual network contains additional virtual edges in the original capacity network.

1.3 Augmenting Path

Definition 1.3 (Augmenting Path) *An augmenting path is a path in G from the source to the sink where the residual capacity of any edge is greater than 0.*

In contrast, since the residual network contains the edge set consisting of all positive capacity edges, any path from the source to the sink in the residual network is an augmenting path.

II Augmenting Path Algorithm

The representative of the traditional network maximum flow algorithm is the augmenting path algorithm. Augmenting path algorithms often use breadth-first search to continuously find feasible augmenting paths for augmentation, and stop the algorithm when there are no more augmenting paths in the network. Most traditional network flow algorithms such as the EK algorithm, Dinic algorithm, MPM algorithm, and ISAP algorithm are augmenting path algorithms.

Algorithms	Optimization Methods	Time Complexity
EK	None	$O(nm^2)$
Dinic	Multi-path Augmentation and Current Arc Optimization	$O(n^2m)$
MPM	Heap-based Priority Queue	$O(n^2 \log n)$
MPM	BFS Augmenting Path Search	$O(n^3)$
ISAP	GAP Optimization and Current Arc Optimization	$O(n^2m)$
HLPP	GAP Optimization and BFS Optimization	$O(n^2\sqrt{m})$
Budget	None	$\Omega(nm)$

2.1 Edmonds-Karp kinetic algorithm (EK algorithm)

Edmonds-Karp kinetic algorithm is a classic augmented path algorithm with a time complexity of $O(nm^2)$.

2.1.1 Edmonds-Karp kinetic algorithm process

Specifically, the process of the EK algorithm is as follows.

i. Initialize the network G, that is, for all $(u_i, u_j) \in E$, $c_f(u_i, u_j) = c_{ij}$, $c_f(u_j, u_i) = 0$; where the edge (u_j, u_i) is called a reverse edge, which is used to cancel the non-optimal local optimal strategy;

ii. Find the augmenting path w through the breadth-first algorithm; if w does not exist, terminate the algorithm;

iii. Find $(u_0, v_0) \in w$, so that for any $(u, v) \in w, c_f(u_0, v_0) \leq c_f(u, v)$;

iv. For all $(u, v) \in w$, $c_f(u, v) \leftarrow c_f(u, v) - c_f(u_0, v_0)$, $c_f(v, u) \leftarrow c_f(v, u) + c_f(u_0, v_0)$, and $M \leftarrow M + c_f(u_0, v_0)$; repeat all steps from the second step.

The above process can be expressed as pseudo code as follows.

Algorithm 1: Edmonds-Karp Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

for edge (u_i, v_i) in E, **do**
    $c_f(u_i, u_j) \leftarrow c_{ij}$;
    $c_f(u_j, u_i) \leftarrow 0$;
end
Create a queue q; Then append s to it;
while a route from s to t can be found by BFS **do**
    Find the minimun capacity value c_{min} among each vertex in the route;
    for Each vertex (u, v) in the route, **do**
        $c_f(u, v) \leftarrow c_f(u, v) - c_f min$;
        $c_f(v, u) \leftarrow c_f(v, u) + c_f min$;
    end
    $M \leftarrow M + c_{min}$;
end

2.1.2 Edmonds-Karp Kinetic Algorithm Time Complexity Analysis

We first prove that each augmentation of the EK algorithm increases the shortest distance $d(u)$ from all vertices $u \in (V - \{s, t\})$ to s. We can use the proof by contradiction to prove this point:

We call the augmented distance function d', assuming that there is a point $u \in (V - \{s, t\})$ in the network such that $d'(u) < d(u)$. Assume that the predecessor node of u is v, so we have

$$d(v) = d(u) - 1, d'(v) \geq d(v)$$

Then we have $(v, u) \notin V$, because if there is $(v, u) \in V$, then there must be

$$d(u) \leq d(v) + 1 \leq d'(v) + 1 = d'(u)$$

This contradicts the assumption, so each augmentation of the EK algorithm increases the shortest distance $d(u)$ from all vertices $u \in (V - \{s, t\})$ to s.

Let us introduce the concept of critical edge:

Definition 2.1 (Critical edge) *If the remaining capacity of an edge e on an augmented path P is equal to the remaining capacity of the augmented path, then e is called the critical edge of P.*

Next, we use the lemma to prove that each edge can be a key edge for at most $n/2 - 1$ times:

For the key edge (u, v), since (u, v) is on the shortest path, we have

$$d(v) = d(u) + 1$$

After augmentation, (u, v) will disappear from the network. The condition for reappearance is that (v,u) appears on the augmented path, so we have

$$d'(u) = d'(v) + 1$$

From the lemma, we know:

$$d'(v) \geq d(v)$$

So we have

$$d'(u) \geq d(v) + 1 = d(u) + 2$$

So each appearance will at least make the shortest distance +2, and the maximum distance is n-2, so each edge can be a key edge for at most $n/2 - 1$ times, and the time complexity of augmentation is $O(nm)$. If BFS is used for augmentation, the complexity is $O(nm^2)$.

2.2 Dinic algorithm

2.2.1 Dinic algorithm process

The Dinic algorithm introduces hierarchical operations based on breadth-first search on the basis of the EK algorithm, and uses hierarchical operations and depth-first search to find the shortest augmenting path. It can be proved that the time complexity of the Dinic algorithm is $O(n^2m)$, which is significantly better than the EK algorithm ($O(nm^2)$) in dense graphs.

Specifically, the process of the Dinic algorithm is as follows.

1. Initialize the network G, that is, for all $(u_i, u_j) \in E$, $c_f(u_i, u_j) = c_{ij}$, $c_f(u_j, u_i) = 0$;
2. Starting from the source point s, a breadth-first search is used to define the depth $d(u)$ for all nodes, where an edge (u, v) can be passed if and only if its remaining capacity is greater than zero; $Thedepthofu$ represents the length of the shortest path from s to u, and $d(s) = 0$; if $d(t)$ does not exist, terminate the algorithm;
3. Find an augmenting path w through depth-first search; where an edge (u, v) can become part of an augmenting path if and only if $d(u) + 1 = d(v)$;
4. Find $(u_0, v_0) \in w$ such that for any $(u, v) \in w, c_f(u_0, v_0) \leq c_f(u, v)$;
5. For all $(u, v) \in w$, $c_f(u, v) \leftarrow c_f(u, v) - c_f(u_0, v_0)$, $c_f(v, u) \leftarrow c_f(v, u) + c_f(u_0, v_0)$, and $M \leftarrow M + c_f(u_0, v_0)$; repeat all steps from the second step.

2.2.2 Dinic algorithm optimization

There are two important optimizations in the Dinic algorithm. These two optimizations can significantly improve the efficiency of the Dinic algorithm, so they are often adopted in practice.

Multi-path augmentation

After confirming an augmenting path w using depth-first search, there may be a subsequence of w $w_0 = \{e_i\}, i = 1, 2, \ldots, k_0 < k$ that satisfies $\forall e_i \in w_0, c_f(e_i) > 0$, at this time, the depth-first search can continue to find the augmenting path until the unconfirmed augmenting path no longer exists. This optimization allows the Dinic algorithm to find multiple augmenting paths in the same depth-first search, thereby improving the efficiency of the algorithm.

Current arc optimization

In the same round of depth-first search, the ith edge (u, v_i) of node u is searched if and only if all $(u, v_j), j = 1, 2, \ldots, i-1$ has been searched, which is equivalent to the above (u, v_j) no longer being part of the newly added wide path. At this time, you can record the index i of the edge that node u is being searched. When you visit u again, you can directly start the traversal search from (u, v_i), thereby improving efficiency.

2.2.3 Time complexity analysis of Dinic algorithm

As mentioned earlier, the time complexity of Dinic's algorithm is $O(n^2m)$. The proof is given below.

First, consider the time complexity of a single round of augmentation. When the current arc optimization is applied, for any node, the current arc changes at most m times, so the worst time complexity of a single round of augmentation is $O(nm)$.

Now we prove that to obtain the maximum flow, at most $n-1$ rounds of augmentation are required.

Any augmenting path w of Dinic's algorithm satisfies $\forall(u, v) \in w, d(v) - 1 = d(u)$. At the same time, it can be found that after each round of augmentation, $d(t)$ will definitely increase.

Assume that after one round of augmentation, $d(t)$ remains unchanged. This shows that there is an augmenting path that satisfies $\forall(u, v) in w, d(v) - 1 = d(u)$. The former just meets the conditions of the augmenting path in the previous round of augmentation, so It should have been augmented in the previous round. Therefore, after one round of augmentation, the aforementioned augmentation path cannot exist, that is, after one round of augmentation, $d(t)$ must increase. Combined with Dinic's termination condition, we can know that the augmentation process is performed at most $n-1$ times.

To sum up, the worst time complexity of Dinic's algorithm is $O(n^2m)$.

2.3 MPM algorithm

The MPM algorithm is similar to the Dinic algorithm in overall structure, but the augmentation method is different. After being optimized using the BFS solution, its time complexity is better than the Dinic algorithm, reaching $O(n^3)$.

To introduce the augmentation method of the MPM algorithm, we need to introduce the concepts of point capacity and reference nodes.

Definition 2.2 (Point capacity) *The minimum value of the incoming residual and the outgoing residual of a point v is the point capacity of the point v. Formally, the point capacity $p(v)$ is defined as:*

$$p_{in}(v) = \sum_{(u,v)\in L} (c(u, v) - f(u, v))$$

$$p_{out}(v) = \sum_{(v,u)\in L} (c(u, v) - f(u, v))$$

$$p(v) = min(p_{in}(v), p_{out}(v))$$

Definition 2.3 (reference node) *A node r is a reference node if and only if* $p(r) = minp(v)$

Compared with Dinic algorithm, the objects augmented by MPM algorithm are vertices rather than edges. Since L is an acyclic graph and the node capacity of any node v_i in L is at least $p(v_i)$, for each reference node v_i, the flow through it can be increased by $p(v_i)$. The increased flow $p(v_i)$ is allocated to a directed path starting from s, passing through v_i and reaching t.

There are many ways to find the augmenting path of the reference node. Using a heap-based priority queue has a time complexity of $O(n^2 logn)$, while using BFS for augmenting path search can reduce the time complexity to $O(n^2)$.

Since the number of reference nodes does not exceed n, the augmented iterations are performed at most n^2 times, so the time complexity of the MPM algorithm can be optimized to $O(n^3)$.

The core process of the MPM algorithm can be expressed as pseudo code. For the pseudo code of the core process of the MPM algorithm, please see Algorithm 2 below.

2.4 ISAP algorithm

2.4.1 ISAP algorithm flow

ISAP algorithm, namely Improved Shortest Augmenting Path, is based on Dinic algorithm and reduces the number of breadth-first searches in the algorithm to one by dynamically changing the depth of nodes.

Specifically, the process of the ISAP algorithm is as follows.

1. Initialize the network G, that is, for all $(u_i, u_j) \in E$, $c_f(u_i, u_j) = c_{ij}$, $c_f(u_j, u_i) = 0$;
2. Define the depth $d(u)$ for all nodes from the sink t through breadth-first search, where the depth of u represents the length of the shortest path from t to u, and $d(t) = 0$; if $d(s)$ does not exist, terminate the algorithm;
3. Find an augmenting path w through depth-first search; where an edge (u, v) can become part of an augmenting path if and only if $d(u) - 1 = d(v)$;
4. Find $(u_0, v_0) \in w$, so that for any $(u, v) \in w, c_f(u_0, v_0) \leq c_f(u, v)$;
5. For all $(u, v) \in w$, $c_f(u, v) \leftarrow c_f(u, v) - c_f(u_0, v_0)$, $c_f(v, u) \leftarrow c_f(v, u) + c_f(u_0, v_0)$, and $M \leftarrow M + c_f(u_0, v_0)$;
6. For each point u in $w/\{t\}$, judge from small to large depth: if there is an adjacent node v such that $c_f(u, v) > 0$ and $d(u) - 1 = d(v)$, continue to search for augmenting paths; otherwise, find the point v_0 in the adjacent node set $\{v\}$ of the residual network u such that $\forall v \in \{v\}, d(v_0) \leq d(v)$, let $d(u) \leftarrow d(v_0) + 1$; in particular, if $\{v\} = \varnothing$, let $d(u) = n$; if $d(s) \geq n$, terminate the algorithm; otherwise, repeat from the third step;

Algorithm 2: Malhotra, Pramodh-Kumar and Maheshwari Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

for vertice v_i in V, **do**
  $c_f(u_i, u_j) \leftarrow c_{ij}$;
  $c_f(u_j, u_i) \leftarrow 0$;
end
for vertice v_i in V, **do**
  for vertice pair (u, v) in V, **do**
    $p_{in}(v) += (c(u, v) - f(u, v))$;
    $minp(v) = min(p_{in}(v_i), min_p(v))$;
  end
  for vertice pair (v, u) in V, **do**
    $p_{out}(v) += (c(u, v) - f(u, v))$;
    $minp(v) = min(p_{out}(v_i), min_p(v))$;
  end
end
Create a vector v;
for vertice r_i in V, **do**
  if $p(r_i) == minp(v)$ **then**
    $v.push\_back(r_i)$;
  end
end
while $!v.empty()$ **do**
  Do BFS to find the route from s to t through r_i;
  $ans += p(r_i)$;
  $v.pop(r_i)$;
end

2.4.2 ISAP algorithm optimization

ISAP algorithm also has two important optimizations.

Current arc optimization
Current arc optimization is also valid for ISAP algorithm. It should be noted that when the depth of a node u changes, the current arc needs to be reset to the first edge.

GAP optimization
When performing breadth-first search, record the number of nodes $n_d(i)$ at each depth. Since any edge (u, v) in the augmented path w satisfies $d(u) - 1 = d(v)$, if $\exists n_d(i) = 0$, it is no longer possible to have a new augmented path in the graph. At this point, the algorithm can be terminated directly to improve efficiency.

III Preflow Push Algorithm

The preflow push algorithm is an efficient algorithm that pushes overflow nodes to the next height as much as possible to solve the maximum flow. Unlike the previous augmented path algorithm, the preflow push algorithm does not search for augmented paths, and the capacity and residual networks do not follow the flow conservation property. The time complexity of the highest-numbered preflow push algorithm is **compact**$O(n^2\sqrt{m})$, which is currently the most efficient network flow algorithm.

Before introducing the general pre-flow promotion algorithm, we need to introduce the general related definitions:

3.1 symbol definition

Definition 3.1 (Excess flow) *If the capacity network allows the flow into a node u to exceed the flow out of it, the excess part is called the excess flow of the node $u(u \in V - \{s, t\},)$. Formally, the definition of the excess flow $e(u)$ of point u is:*

$$e(u) = \sum_{(x,u)\in E} f(x, u) - \sum_{(u,y)\in E} f(u, y)$$

When $e(u) > 0$, we call the node u**overflow**. In the pre-flow promotion algorithm, the overflow of nodes and the occurrence of excess flow are allowed.

Definition 3.2 (Height function) *Compared with the concept of height function in traditional network flow algorithms, it is broader. A function that maps from a point set to a real number set and satisfies certain conditions is called a height function. Formally, the height function h of the residual network $G_f = (V_f, E_f)$ must meet the following conditions:*

$$h(s) = |V|, h(t) = 0$$

$$\forall (u, v) \in E_f, h(u) \leq h(v) + 1$$

From the definition of the height function, we can draw an important conclusion, that is, the push execution of the pre-flow push algorithm only occurs on the edge of $h(u) = h(v) + 1$.

3.2 Operation Definition

Definition 3.3 (Push) *The operation of pushing the excess flow of the overflow node u to the next node v is called a push operation (if v overflows, we need to mark v accordingly, which is put into the heap in HPLL). Formally, the applicable conditions for push are: for u, $\exists v((u,v) \in E_f, c(u,v) - f(u,v) > 0, h(u) = h(v) + 1)$.*

If $f(u,v)$ is exactly $c(u,v)$ after pushing, then (u,v) is called **full flow** and it is deleted from the residual network.

Definition 3.4 (Relabel) *The operation of increasing the height of the overflow node to solve the problem that the overflow node cannot push the flow (there is no node with a lower height function) is a relabel operation. Formally, the applicable conditions for the relabeling operation can be expressed as follows:*

$$\forall (u,v) \in E_f, h(u) < h(v),$$

$$where : e(u) = \sum_{(x,u)\in E} f(x,u) - \sum_{(u,y)\in E} f(u,y) > 0$$

The specific update operation is:

$$h(u) = min_{(u,v)\in E_f} h(v) + 1$$

Definition 3.5 (Initialization) *The initialization of the classic pre-flow push algorithm is to set the source point height function to $|V|$ (i.e. the number of points), set the height function of nodes other than the source point to zero, and calculate the excess flow of each node. Formally, the initialization operation is defined as follows:*

$$\forall (u,v) \in E, f(u,v) = \begin{cases} c(u,v), & u = s \\ 0, & u \neq s \end{cases}$$

$$\forall u \in V, h(u) = \begin{cases} |V|, & u = s \\ 0, & u \neq s \end{cases}$$

$$e(u) = \sum_{(x,u)\in E} f(x,u) - \sum_{(u,y)\in E} f(u,y)$$

3.3 General pre-stream push algorithm

The common template of the pre-stream push algorithm can be summarized as: for all nodes in the graph that meet the conditions for push and relabel operations, perform these two operations until there are no such nodes in the graph. The algorithm pseudocode is shown below:

Algorithm 3: General Preflow Push Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

```
1  init();// Initiate with steps given above;
2  do
3      Unmark all vertexs;
4      for vertex v in V do
5          Push if conditions for pushing are satisfied;
6          Relabel if conditions for relabeling are satisfied;
7          if Pushed or Relabeled then
8              Mark v;
9          end
10     end
11 while at least one marked vertex can be found;
12 M ← e(t);
```

3.4 Highest Label Preflow Push (HLPP)

The Highest Label Preflow Push algorithm is widely regarded as the best algorithm among preflow push algorithms. We have summarized the process of the classic HLPP algorithm by consulting papers. Specifically, the process of the classic HLPP algorithm is as follows.

1. Perform initialization operations according to the above definition;
2. Put the overflowed nodes in the network into the big root heap with height as the key value;
3. Select the node u_i with the highest height, push all its pushable edges, and add the nodes that overflow after pushing to the big root heap;
4. If u_i still overflows, relabel and return to step 3;
5. If u_i does not overflow, then u_i is removed from the big root heap;
6. When the heap is empty, the algorithm ends;

For its pseudo code, see 4.

3.5 Common HLPP Optimization

The Relabel and Push operations of the classic HLPP algorithm are relatively redundant, especially the Relabel operation can be done through some Simple optimization is significantly reduced. Next, we will introduce the commonly used BFS optimization and GAP optimization. The HLPP algorithm used today often uses these simple optimizations.

3.5.1 BFS optimization

In the practical application of the HLPP algorithm, improving the initialization method can greatly reduce the number of Relabel operations.

Algorithm 4: Highest Label Preflow Push Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

```
for vertice pair (u_i, v_i) in E, do
    if u == s then
        f(u, v) ← c(u, v);
    end
    else
        f(u, v) ← 0;
    end
end
for vertice u_i in V, do
    if u == s then
        h(u, v) ← |V|;
    end
    else
        h(u, v) ← 0;
    end
    e(u) ← sum(f(x, u)) − sum(f(u, y));
end
Create a max heap s;
for vertice u_i in V, do
    if u_i is overflow then
        s.insert(u_i);
    end
end
while !s.empty() do
    let u_i be the highest vertice in s;
    while u_i is overflow do
        Push (u_i, v_i) or Relabel u_i;
        if v_i is overflow then
            s.insert(v_i);
        end
    end
    s.delete(u_i);
end
```

Specifically, we will modify the operation of initializing the height. When initializing the breadth-first search, We initialize the height function of the node as the distance from the node to the sink. In particular, the height function value of the source point is still the size of the point set. Formally, the initialization of the HPLL after BFS optimization is defined as follows:

$$\forall u \ inV, h(u) = \begin{cases} |V|, & u = s \\ d(u,t), & u \neq s \end{cases}$$

3.5.2 GAP optimization

The HLPP algorithm also has GAP optimization. From the process of the classic HLPP algorithm, it is not difficult to see that the overflow node that cannot push the excess flow downward will eventually increase in height as the Relabel continues until the height exceeds the source point, pushing this part of the excess flow downward. To the source point.

Therefore, at a certain moment in the algorithm, if there is a height h_g and the number of nodes with a height function value of h_g is 0, it means that a "fault" has occurred, that is, a GAP. Overflow nodes of h_g cannot push nodes with heights lower than h_g. We can relabel the heights of these nodes to $n+1$ so that they can directly push excess flows back to the sink.

One implementation method It was proposed by [1], which adopts the bucket idea and opens $2*N-1$ buckets to record the mapping from height to node, that is, bucket $B[i]$ records the current height of i Overflow nodes.

3.6 Ping-Pong Effect

Even with common optimizations (BFS optimization, GAP optimization), HLPP still cannot overcome the common problem of pre-flow push algorithms known as the "Ping-Pong Effect". In today's academic circles, there are many Scholars in the algorithm field are committed to improving HLPP to avoid this effect in order to make HLPP more efficient. We will use a simple example to describe the ping-pong effect, and then give a formal definition of the ping-pong effect. To this end, we give the following initial figure:

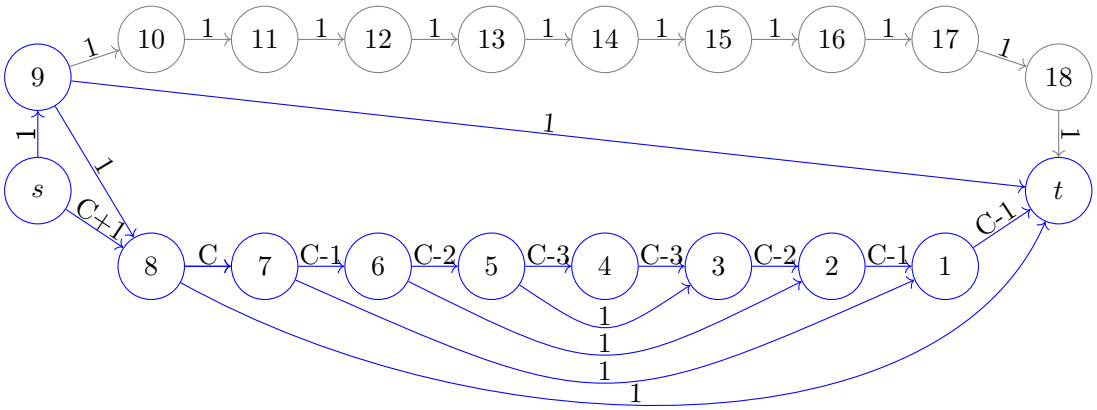

Figure 1. Initial graph of the HLPP algorithm

In Figure Running the HLPP algorithm on the graph, after several iterations, we get the following residual network:

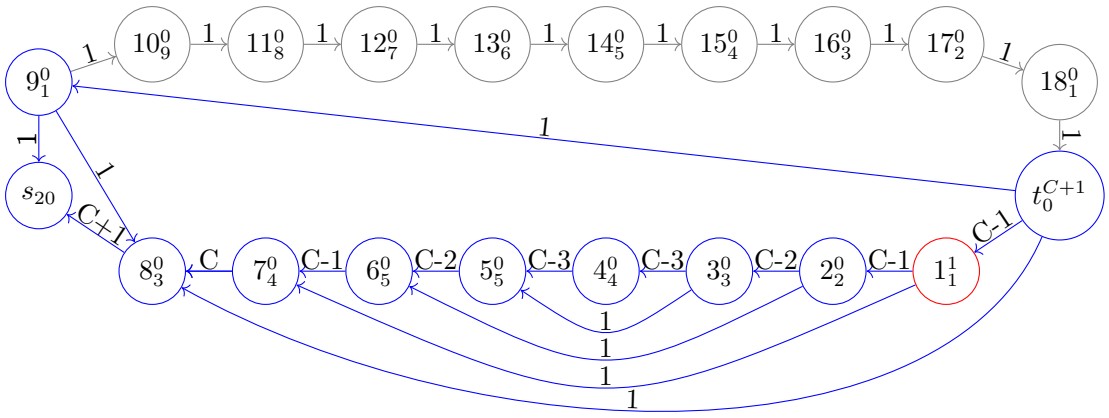

Figure 2. Residual network after several operations

For each node u_i in the graph, we mark its Specifically, for each node, we record it as p_d^e, where p is the node number, e is the excess flow, and d is its height. The number on the edge represents its remaining capacity.

At this time, the overflow node with the highest height function value in the graph is node 1. When the algorithm continues to execute, node 1 will be selected, the height function value will become 3, and Node 2 pushes one unit of traffic, causing node 2 to overflow, and node 1 itself no longer overflows.

Subsequently, node 2 will be selected by the algorithm, the height function value will become 4, and a unit of traffic will be pushed to node 3, causing node 3 to overflow and node 2 itself to no longer overflow.

As the HLPP algorithm continues, this unit of traffic will be pushed to node 8. The flow and height functions and the remaining capacity of each edge are shown in Figure 3.

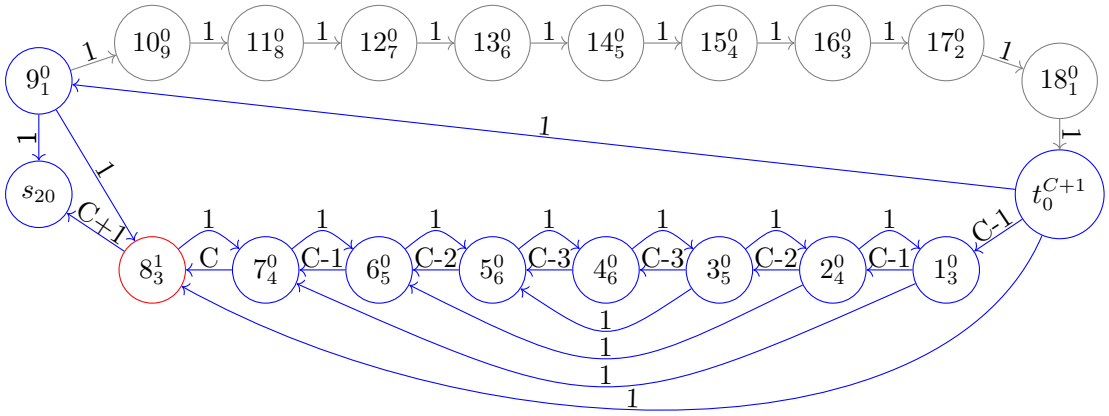

Figure 3. Residual network after streaming is completed

Since only node 8 can be selected in the figure , the algorithm will select node 8, change its height function value to 5, and push one unit of traffic to node 7, so that node 7 overflows and node 8 itself no longer overflows. Then, the algorithm will select node 7 again and push one unit of traffic to node 6. Push one unit of traffic to make node 6 overflow, and node 7 itself no longer overflows. The algorithm performs Relabel and Push operations on each node on the path $\{6, 5, 4, 3, 2, 1\}$ in

turn. After reaching node 1, the algorithm will again perform Relabel and Push operations on each node on the path $\{1, 2, 3, 4, 5, 6\ 7\ 8\}$ in turn. The above two rounds of streaming can be seen in Figures 4 and 5.

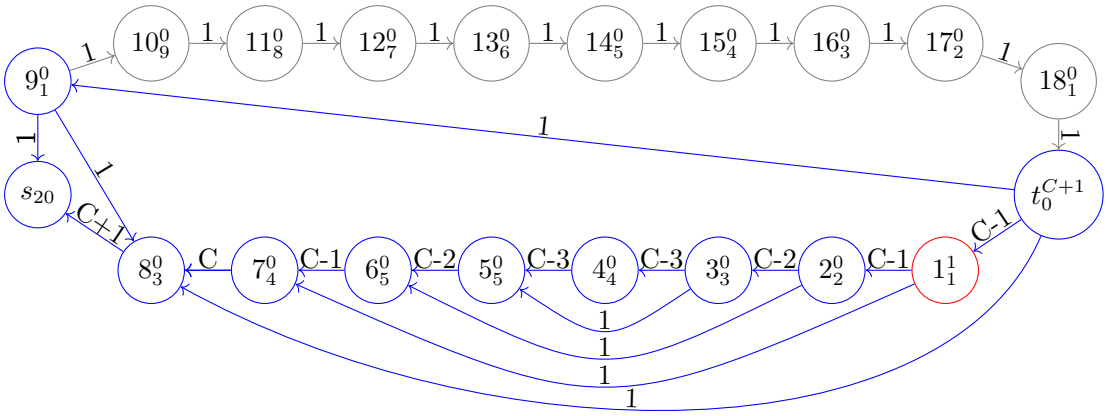

Figure 4. Residual network when pushing traffic to node 1

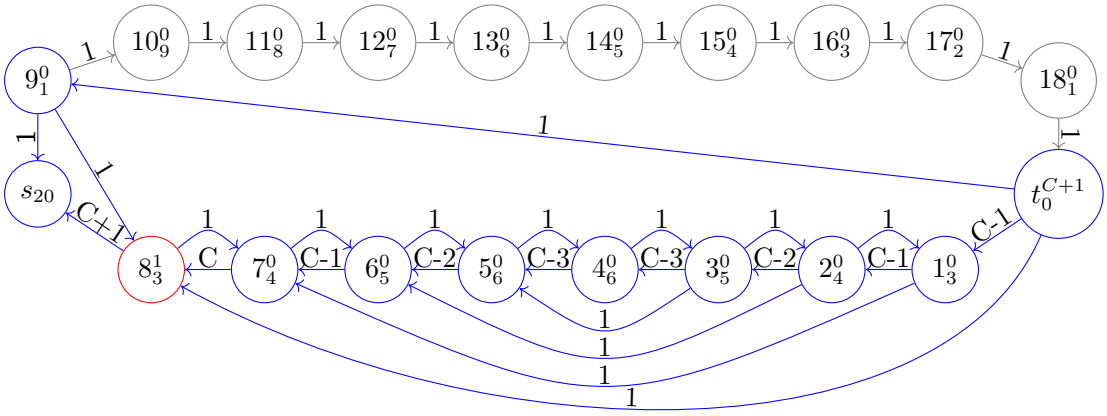

Figure 5. Residual network when pushing the stream to node 8 again

It is not difficult to find that The residual network after n+2 times of Relabel-Push operation from one end to the other end has the same structure as the residual network after n times, and only the height function label has increased by 2. That is, In other words, HLPP repeatedly forms a completely identical residual network in the process of operation, except that the height label rises slowly. In this process, the network does not meet the conditions for GAP optimization. This operation will continue until the figure The height of the middle node exceeds n, and it is executed n times in total. This phenomenon is called the "ping-pong effect".

Generally, the network diagram that will produce the ping-pong effect is shown in Figure 6. When HLPP is run on this graph, the push occurs along the paths $\{1, 2, 3, ..., n/2 - 2\}$, $\{n/2 - 2, n/2 - 3, ..., 2, 1\}$ Repeat the Relabel-Push operation n times, and each time along the path will perform $n/2 - 3$ push operations, the push operation is $\Omega(n^2)$.

3.7 Budget algorithm

The Budget algorithm was proposed by [2]. It is a pre-flow push algorithm that combines some features of augmented paths and can effectively avoid the ping-pong effect. When introducing the

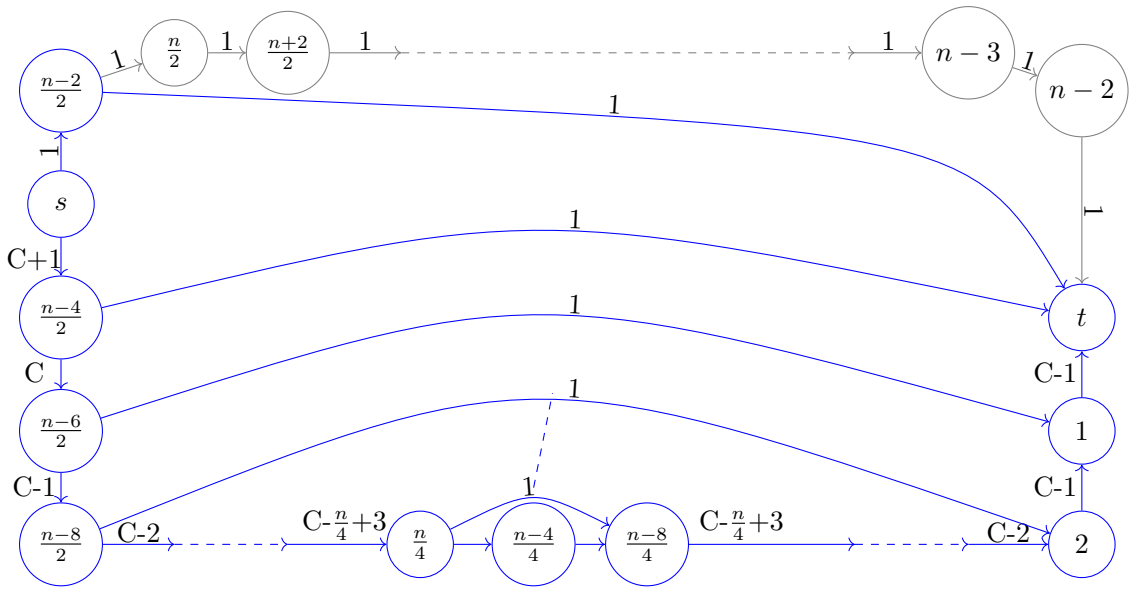

Figure 6. Ping-Pong Effect in General

ping-pong effect, it is not difficult to find that HLPP will simply push along the allowed arc without considering whether the successor node can really "digest" the pushed traffic. A natural solution is to search for a length of at least k starting from the current highest overflow node and push along this path.

To this end, we can introduce the following definition:

Definition 3.6 (Budget) *When initialized, corresponding to a constant k, the Budget of node u_i is $kd(u_i)$.*

When the algorithm selects the highest overflow node u_i, u_i will initialize the Budget and start trying to build an allowed path toward the sink. The path does not have to reach the confluence point. It can also extend only part of the path on another node u_j and reduce the Budget by $d(u_j)$. When the Budget is negative, push the node to which it is extended.

If such a path cannot be found from the node u_i, that is, when the Budget is positive, it is impossible to continue looking for the next admissible arc. At this time, the algorithm shrinks the path, deletes the terminal vertex, and tries to extend the path from other nodes.

Many scholars have proposed specific implementation methods based on this idea. For example, [5] proposed an implementation using a dynamic tree data structure for storage, with an overhead of $O(nm \log n^2/m)$, while [6] introduced an algorithm for fixed extension path length (according to [2], this approach is not efficient).

We show the pseudo code of the Budget algorithm as follows:

IV Network flow algorithm with approximate linear complexity

[3] presents an algorithm that can solve the maximum flow and cost flow algorithm in integer flow networks in $m^{1+o(1)}$ time. The algorithm constructs the flow through a series of $m^{1+o(1)}$ approximate undirected minimum ratio loops, each of which uses a new type of dynamic data

Algorithm 5: Budget Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

```
1  init(G);// Initiate with steps given above;
2  while there exists overflow nodes;
3   do
4      i ← argmax{d(i)|i ∈ N, e(i) > 0};
5      budget ← kd(i);
6      i_t ← i;
7      while budget ⩾ 0 and i_t ≠ s and i_t ≠ t;
8       do
9          if there exists an admissible arc (i_t, j) ∈ δ⁺_G(i_t);
10          then
11             succ(j) ← NIL;
12             pred(j) ← i_t;
13             succ(i_t) ← j;
14             i_t ← j;
15             budget ← budget − d(j);
16         end
17         else
18             d′ ← min{d(j) + 1|(i, j) ∈ δ⁺_G(i_t)};
19             if i_t ≠ i then
20                budget ← budget + d(i_t);
21                i_t ← pred(i_t);
22             end
23             else
24                budget ← kd′;
25             end
26             d(i_t) ← d′;
27             succ(i_t) ← NIL;
28         end
29     end
30     augment(i);
31 end
32 M ← e(t);
```

structure for processing and calculation.

4.1 Improved IPM

The application of IPM (interior point method) in maximum flow originated from [4]. The article combines the interior point method with the fast Laplace system solver to implement the $O(m^{1.5}log^2U)$ algorithm for cost flow. The advantage of IPM applied to network flow problems is that it can reduce network flow problems on directed graphs to problems on undirected graphs that are easier to handle.

The paper proposes an improved IPM algorithm, denoted as ℓ_1IPM algorithm (here ℓ_1 is the first normal form).

Theorem 1 *The ℓ_1IPM algorithm can decompose the cost flow problem into $m^{1+o(1)}$ instances of sequentially solving undirected minimum ratio cycles, each of which reaches $m^{o(1)}$ approximation.*

Furthermore, the problem instances generated by the decomposition are "stable", i.e. they satisfy:

1) The direction from the current flow to the unknown optimal flow is a good enough solution for each instance

2) The length and gradient parameters of the input instance change only for the amortized $m^{o(1)}$ edges at each iteration.

The classic IPM approach reduces the minimum cost flow to solving $\widetilde{O}(\sqrt{m})$ instances to ensure accuracy, which is a ℓ_2 minimization problem. The undirected minimum ratio cycle is a ℓ_1 minimization problem, which is much simpler and can afford the $m^{o(1)}$ approximation factor in the subproblem.

4.2 Dynamic extension data structure

The ℓ_1 IPM algorithm rebuilds only a portion of the low extension spanning tree after each IPM iteration to accommodate the change in length. To do this, a new data structure is needed that maintains a recursive sequence of minimum ratio cycle problem instances on graphs with fewer edges and vertices. These smaller instances give worse approximations but are cheaper to maintain. We exploit the stability of IPM to partially embed vertices into trees.

Theorem 2 *The dynamic extension data structure enables edge updates (insertions/deletions/vertex splits) to an unweighted, undirected graph G. Maintain a subgraph H with $O(n)$ edges and an explicit path embedding of each $e \in H$ into H of length $m^{o(1)}$. For each edge update, the amortized number of edge changes in H is $m^{o(1)}$. Furthermore, the set of edges that embed a fixed edge $e \in H$ is decreasing for all edges e.*

4.3 pseudocode

See 6 for pseudocode of the approximate linear complexity cost flow proposed in the paper.

Algorithm 6: MinCostFlow($G, d, c, u+, u-, f(0), F*$)

Input: graph G, demands d, costs c, upper/lower capacities $u+$,$u-$, initial feasible flow $f(0)$, the optimal flow F^*

Output: maximum flow f^t

$\alpha \leftarrow 1/(1000logmU)$;
$\kappa \leftarrow exp(-O(log7/8mloglogm))$;
$d \leftarrow O(log^{1/8}n)$;
$D^{(HSFC)}$; // Dynamic datastructure;
$T1, T2, ..., Tsfors \leftarrow O(logn)^d$;
$\epsilon \leftarrow \kappa\alpha/(1000s)$;
$r \leftarrow \infty$;
while $c^T > f(t) - F^* (mU)^{-10}$ **do**
  if t is a multiple of b $\in \lfloor \epsilon c \rfloor$;
  then
    Explicitly compute $f^{(t)} \leftarrow f^{(0)} + \sum_{i\in[s]} f_i, \tilde{f}(t) \leftarrow f(t)$;
    $r \leftarrow c^T\tilde{f}(t) - F^*$;
    $g^{(t)} \leftarrow g(\tilde{f}(t)), l^{(t)} \leftarrow l^{\tilde{f}(t)}$;
    Rebuild $D^{(HSFC)}$ *and update the* T_i;
  end
  $U^{(t)} \leftarrow \bigcup_{i\in[s]} D^{(Ti)}.Detect()$;
  for $e \in U^{(t)}$ **do**
    $Set\tilde{f}_e^{(t)} \leftarrow f_e^{(t)} = f_e^{(0)} + \sum_{i\in[s]}(f_i)_e, l_e^{(t)} \leftarrow l_e^{\tilde{f}(t)}$;
    $g_e^{(t)} \leftarrow 20mc_e/r + \alpha(u_e^+ - \tilde{f}_e^{(t)})^{(-1-\alpha)} - \alpha(\tilde{f}_e^{(t)} - u_e^-)^{(-1-\alpha)}$;
  end
  for $e \notin U^{(t)}$ **do**
    $g_e^{(t)} \leftarrow g_e^{(t-1)}, l_e^{(t)} \leftarrow l_e^{(t-1)}, \tilde{f}_e^{(t)} \leftarrow \tilde{f}_e^{(t-1)}$;
  end
  $D^{(HSFC)}.Update(U^{(t)}, g^{(t)}, l^{(t)}), and\ update\ the\ T_i\ for\ i \in [s]$;
  $(i, \Delta) \leftarrow D^{(HSFC)}.Query(), where\ i \in [s]\ and$;
  $\Delta = (u1, v1) \oplus T_i[v1, u2] \oplus (u2, v2) \oplus ... \oplus (ul, vl) \oplus T_i[v_l, u_1]\ for\ edges\ (u_i, v_i)\ and\ l\ m^{o(1)}$;
  $\Delta \leftarrow \eta\Delta for \eta \leftarrow -\kappa^2\alpha^2/(800hg^{(t)}, \Delta_i)$;
  $f_i \leftarrow f_i + \Delta\ using D^{(Ti)}$;
  $t \leftarrow t + 1$;
  $return\ f^{(t)}$;
end
$M \leftarrow e(t)$;

V Problem specification

5.1 Bipartite graph maximum matching problem

A bipartite graph is a graph whose nodes consist of two sets and there are no edges between the two sets. Given a bipartite graph with n left points, m right points and e edges, find the maximum number of edges for its maximum matching.

The bipartite graph maximum matching problem can be converted into a network flow model.

Connect the source point to all points on the left and all points on the right to the sink, and the capacity is 1. The original edge connects from left to right, and the capacity is also 1. The maximum flow is the maximum matching.

If the Dinic algorithm is used to find the maximum flow of the network, it can be found in $O(\sqrt{n}m)$.

The Dinic algorithm is divided into two parts. The first part uses $O(m)$ time BFS to establish the network flow, and the second step is $O(nm)$ time DFS for augmentation.

But because the capacity is 1, the actual time complexity is $O(m)$.

Next, the first $O(\sqrt{n})$ rounds have a complexity of $O(\sqrt{n}m)$. After $O(\sqrt{n})$ rounds, the length of each augmented path is at least $\sqrt{n}$, and such a path does not exceed $\sqrt{n}$, so at this time, at most $\sqrt{n}$ rounds are needed, and the overall complexity is $O(\sqrt{n}m)$.

5.2 Minimum cost maximum flow

Given a network $G = (V, E)$, each edge has a capacity limit $c(u, v)$ and a unit flow cost $w(u, v)$.

When the flow of (u, v) is $f(u, v)$, it costs $f(u, v) \times w(u, v)$.

w also satisfies the skew symmetry, that is, $w(u, v) = -w(v, u)$.

Then the maximum flow with the minimum total cost in the network is called the minimum cost maximum flow, that is, minimizing $\sum_{(s,v)\in E} f(s, v)$ under the premise of maximizing $\sum_{(s,v)\in E} f(u, v) \times w(u, v)$.

Simply replace the process of finding augmenting paths in the EK algorithm or Dinic algorithm with using the shortest path algorithm to find augmenting paths with the minimum unit cost.

5.3 Maximum Weight Matching of Bipartite Graph

The maximum weight matching of a bipartite graph refers to the matching with the maximum sum of edge weights in the bipartite graph.

Given a bipartite graph, the number of its left points is n, the number of its right points is m, the number of edges is e, and the edge weight of each edge is wi.

The maximum weight matching of a bipartite graph can be transformed into a cost flow model.

Add a source point and a sink point to the graph.

Connect an edge with a flow of 1 and a cost of 0 from the source point to each left point of the bipartite graph, and connect an edge with a flow of 1 and a cost of 0 from each right point of the bipartite graph to the sink point.

Next, for each edge in the bipartite graph that connects the left point u and the right point v with an edge weight of w, connect an edge from u to v with a flow of 1 and a cost of wi.

Find the maximum cost maximum flow of this network to get the answer.

5.4 Minimum cut problem

For a network flow graph $G = (V, E)$, its cut is defined as a point partitioning method: all points are divided into two sets S and $T = V - S$, where the source point $s \in S$ and the sink point $t \in T$.

We define the capacity $c(S, T)$ of the cut (S, T) to represent the sum of the capacities of all edges from S to T, that is, $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$. Of course, we can also use $c(s, t)$ to represent $c(S, T)$.

The minimum cut is to find a cut (S, T) that minimizes the capacity $c(S, T)$ of the cut.

It can be proved that $f(s, t)_{\max} = c(s, t)_{\min}$

For any cut (S, T) of a feasible flow $f(s, t)$, we can get:

$$f(s,t) = S\text{total outgoing flow} - S\text{total incoming flow} \leq S\text{total outgoing flow} = c(s,t)$$

If we find the maximum flow f, then there must be no augmented path from s to t in the residual network, that is, the outgoing edge of S must be full flow, and the incoming edge of S must be zero flow, so we have:

$$f(s,t) = S\text{total outgoing flow} - S\text{total incoming flow} = S\text{total outgoing flow} = c(s,t)$$

Combined with the previous inequality, we can know that f has reached its maximum at this time.

If you need to minimize the number of cut edges under the premise of minimum cut, then first find the minimum cut, change the capacity of the edge without full flow to inf, and the capacity of the edge with full flow to 1, and run the minimum cut again to find the minimum number of cut edges; if there is no premise of minimum cut, directly set the capacity of all edges to 1, and find the minimum cut to get the answer.

VI Algorithm test

6.1 Data set description

We use a total of 5 data sets for algorithm performance testing. The data sets used are all randomly generated, and the solid time consumption is lower than the theoretical time complexity.

6.2 Running results

Algorithm	n	m	Notes	Time consumption
Dinic algorithm	1200	120000	Test data 1	531ms
HLPP algorithm	1200	120000	Test data 1	46ms
Budget algorithm	1200	120000	Test data 1	46ms
Dinic algorithm	2000	200000	Test data 2	1468ms
HLPP algorithm	2000	200000	Test data 2	78ms
Budget algorithm	2000	200000	Test data 2	78ms
Dinic algorithm	1000	400000	Test data 3	15656ms
HLPP algorithm	1000	400000	Test data 3	171ms
Budget algorithm	1000	400000	Test data 3	187ms
Dinic algorithm	1500	600000	Test data 4	31281ms
HLPP algorithm	1500	600000	Test data 4	250ms
Budget algorithm	1500	600000	Test data 4	265ms
Dinic algorithm	3000	600000	Test data 5	17187ms
HLPP algorithm	3000	600000	Test data 5	296ms
Budget algorithm	3000	600000	Test data 5	312ms

references

[1] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, 97(3):509–542, 1997.

[2] R. Cerulli, M. Gentili, and A. Iossa. Efficient preflow push algorithms. *Computers & Operations Research*, 35(8):2694–2708, 2008. Queues in Practice.

[3] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. 2022.

[4] S. I. Daitch and D. A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. *arXiv e-prints*, 2008.

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

[6] Giuseppe Mazzoni, Stefano Pallottino, and Maria Grazia Scutellà. The maximum flow problem: A max-preflow approach. *European Journal of Operational Research*, 53(3):257–278, 1991.