Lab09-Network Flow

CS214-Algorithm and Complexity, Xiaofeng Gao & Lei Wang, Spring 2021.

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:RenyangGuan Student ID:519021911058 Email: guanrenyang@sjtu.edu.cn

1. Consider there is a network consists n computers. For some pairs of computers, a wire i exists in the pair, which means these two computers can communicate with each other. When a signal passes through the wires, the noise in the signal will be amplified.If you know the magnification rate of noise $m_{i,j}$ of each wire (which must be greater than 1). Design an algorithm to find the route for each other computer to send signals to the computer v with the minimum total magnification rate of noise and analyze the time complexity.

Solution. In graph $G = (V, E)$, let $v \in V$ denote each computer and $e \in E$ denote the wire between computers.
Intuition: The intuition comes from $Dijkstra$ algorithm and is based on $Greedy$ algorithm. Firstly, we should re-define the shortest path as the path in which the product of the weight of each path is the minimum.
Then we maintain a set S of vertices whose shortest path distance from S are known. At each step add to S the vertex $v \in V - S$ whose distance estimate for S is minimal, Finally, update the distance estimates of vertices adjacent to v.
Since each wire is able to sent or receive signals, so G is an undirected graph. The shortest path of (u, v) is the same as that of (v, u). As a result of this, setting the sink vertex v as the source vertex in the $ModifiedDijkstra$ algorithm could get the result.

Algorithm 1: Modified Dijkstra Algorithm

Input: object vertex v receiving signals
Output: An array d maintaining result

```
1  create array d;
2  create min-heap Q;
3  foreach k ∈ V do
4      if k = v then
5          d[k] ← 0;
6      else
7          d[k] ← ∞;
8  foreach u ∈ V do
9      INSERT(Q, d[u]);
10 while Q ≠ ∅ do
11     u ← EXTRACT-MIN(Q);
12     S ← S ∪ {u};
13     foreach v ∈ adjacent(u) do
14         if d[v] > d[u] × w(u, v) then
15             d[v] ← d[u] × w(u, v);
16             DECREASE-KEY(Q, d[v]);
```

Time complexity algorithm: The for loop in line 3, the for loop in line 8, and the $while$ loop in line 10 will loop for $|V|$ times. The time complexity of $INSERT$ and that of $DECREASE - KEY$ are based on the data structure of Q, as well as the density of

the graph. By using the best data structure Fibonacci heap, the time complexity could be reduced to $O(|V|\log|V|+|E|)$.
Considering the density, since $|E| \leq \frac{|V|(|V|-1)}{2}$, the total time complexity won't exceed $O(|V|^2)$

□

2. Suppose that we wish to maintain the transitive closure of a directed graph $G=(V,E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.

   (a) Show how to update the transitive closure of a graph $G=(V,E)$ in $O(V^2)$ time when a new edge is added to G.

   (b) Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.

   (c) Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of m insertions, your algorithm should run in total time $\sum_{i=1}^{m} t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

Solution.

Notation:
In graph $G=(V,E)$, let boolean matrix $A=(a_{ij})_{|V|\times|V|}$ to maintain the transitive closure, in which $a_{ij}=1$ means there exists a directed path from vertex v_i to v_j.

Algorithm pseudocode:

Algorithm 2: $INSERT(v_i, v_j, A)$

Input: An edge from v_i to v_j, matrix A
Output: Boolean martix A maintaining the transtive closure

```
1  Create array Source = {0}, Sink = {0};
2  a_ij = 1;
3  foreach v_k ∈ V do
4      if a_ki = 1 and a_kj ≠ 1 then
5          Source ← k;
6      else
7          Sink ← k;
8  foreach i ∈ Source do
9      foreach j ∈ Sink do
10         a_ij ← 1;
```

Example: As is shown in Fig. **??**, for each inserted edge e from v_i to v_j, we must check each node to determine whether it has a path toward v_i or a path coming from v_j, which takes $O(|V|)$ time. Let set $ToVi$ denote the set of vertices having a path to v_i and $FromVj$ denote the set of vertices having a path from v_j. For any p,q, where $v_p \in ToVi$ and $v_q \in FromVj$, there exists a path from v_p to v_q. As a result, we must update a_{pq} which takes time $\Theta(|ToVi| \times |FromVj|)$. On average, $ToVi \sim V$ and $FromVj \sim V$, so the time complexity is at least $\Omega(|V|^2)$

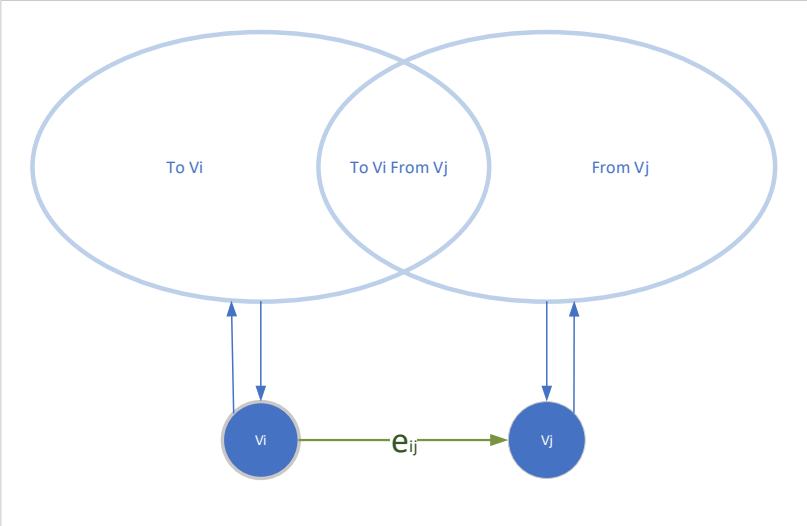

图 1: The example for time complexity of insertion

Time complexity analysis: Since the graph G is empty initially, the sequence of m operation will all at most m edges. Additionally, the number of edges is at most $\frac{|V|\times(|V|-1)}{2}$, which is the possible range of m.
Case 1: $m = O(|V|)$. Since the worst case time complexity of a single operation is $O(|V|^2)$, the total time complexity of the sequence of m operations is $O(|V|^3)$.
Case 2: $m = O(|V|^2)$. Only when $a_{ki} = 1$ and $a_{kj} \neq 1$, vertex k will be added into $Source$. The same is $Sink$. In such case, the time complexity of the for loop in the line 8 is $O(|V|)$ and total time complexity is $O(|V|)$
When the number of edges of G is equivalent of $\Omega(|V|)$, the number of operations where $|Source| \sim O(|V|)$ and $|Sink| \sim O(|V|)$ is $\Theta(1)$. As a result of this, the amortized total time complexity is

$$O(|V|) \times \Theta(|V|^2) + O(|V|^2) \times \Theta(|V|) = O(|V|^3)$$

□

3. An $n \times n$ grid is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 26.11. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1, i = n, j = 1$, or $j = n$. Given $m \leqslant n^2$ starting points $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$ in the grid, the escape problem is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary such that every vertex in V is included in at most one of the m paths. For example, the grid in Figure **??**(a) has an escape, but the grid in **??**(b) does not.

   (a) Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size. That is, the sizes of the two graph are in the same order of magnitude.

   (b) Describe an efficient algorithm to solve the escape problem, and analyze its running time.

Solution.

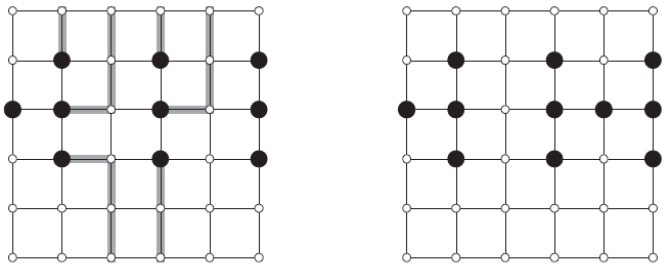

图 2: Grids for the escape problem. Starting points are black, and other grid vertices are white. (a) A grid with an escape, shown by shaded paths. (b) A grid with no escape.

(a) Transforming the network with vertices as well as edges having capacities into ordinary network could be done by *splitting* vertices.
Let v be the vertex with capacity w_v. The *split* operation is shown in Fig. **??**.

i. Delete the vertex v and add two vertices v_i, v_j with infinity capacity.
ii. Add edge e_{ij} from v_i to v_j with capacity W_v into G.
iii. For each edge $e_{u,v}$ from vertex u to v, traverse it to e_{u,v_i}. Similarly, traverse each out-edge $e_{v,k}$ to $e_{v_j,k}$.

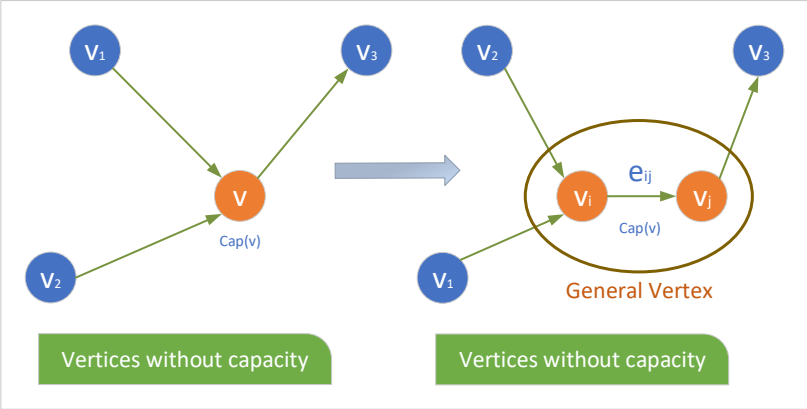

图 3: Transform the question into ordinary network flow

Correctness:
The transforming method maintains the twp properties of flow network.

i. The capacity of each remaining edge is maintained.
ii. Each edge is transformed into general vertex. The in-flow and out-flow of each general vertex is the same as before. The edge in the general edge constraint the capacity of it.

(b)

Step 1: Delete starting points in the boundary, as is shown in Fig. **??**.
Firstly we should delete the starting points in the boundary, because it could be neither the ending point nor the middle point of a path. Additionally, the edge connecting this kind of vertex and the inner vertex can never be passed by. As a result of this, deleting it makes sense.
Step 2: Abstract the boundary into one node, as is shown in Fig. **??**.

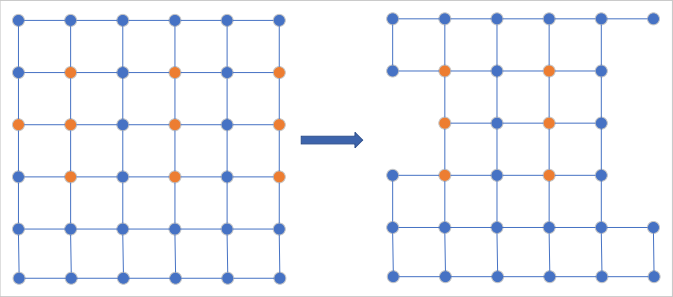

图 4: Step1: Delete starting points in the boundary.

Abstract the remaining outer vertices into one vertex with each vertex in the sub outer layer having an edge into it.

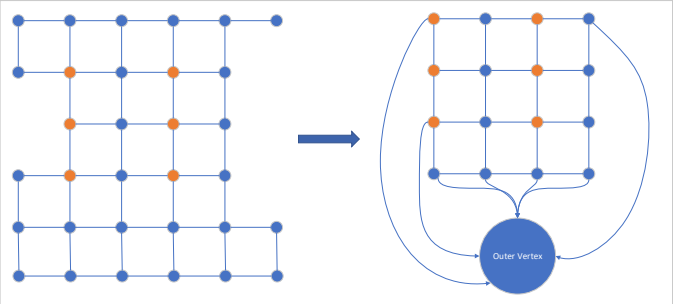

图 5: Step2: Abstract the boundary into one node.

Step 3: Generalization, as is shown in Fig/ **??**. Each vertex is regarded as a vertex with capacity of one, and it is transformed into a generalized node by the method of the previous question.
Furthermore, each undirected edge is transformed into two directed edges with opposite directions between the same pair of vertices.
Except for the $Outer\ Vertex$, there are $2(n-1)^2$ vertices left, since each remaining vertex is *split* into two vertices.

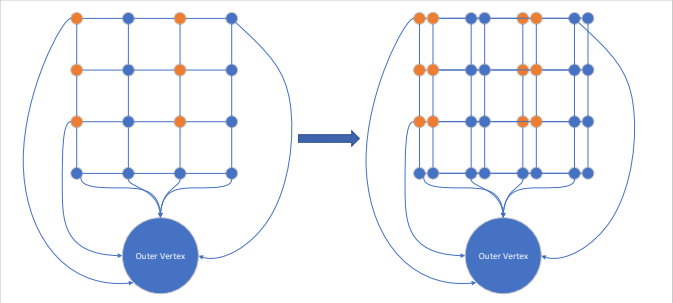

图 6: Step3: Generalization.

Step 4: Solve the problem by modified Ford-Fulkerson algorithm.
Let V to denote the set of remaining vertices, E to denote the set of directed edges, and S to denote the set of remaining starting vertices.

Algorithm 3: Solving escape problem

Input: $G = (V, E)$,S

```
1  foreach e ∈ E do
2      f(e) ← 0;
3  G_f ← residual graph;
4  t ← Outer Vertex;
5  foreach s ∈ S do
6      while An augmenting path P_{s,t} exists do
7          f ← AUGMENT(f, 1, P);
8          update G_f;
9  capacity← flow;
10 foreach s ∈ S do
11     if the path from s to t doesn't exist then
12         return False;
13 return True;
```

Time complexity analysis: Considering the time complexity of searching a path(or augmenting path) is $O(|V|+|E|)$ and that of $Ford-Fulkerson$ algorithm is $O(|E|\times|V|)$, the time complexity of the for loop in line 5 is $O(|S| \times |V| \times |E|)$. Furthermore, the time complexity of the for loop in line 10 is $O(|S| \times (|V| + |E|))$.

As a result of this, the time complexity of the algorithm is

$$O(|S| \times |V| \times |E|)$$

Since $|S| = m$, $|V| = 2(n-1)^2 + 1$, and $|E| \sim n^2$, the total time complexity is

$$O(|S| \times |V| \times |E|) = O(mn^4)$$

□

Remark: Please include your .pdf, .tex files for uploading with standard file names.