chapter 8: graph theory

You Me

date: Yesterday

目录

1 single-source shortest path algorithm

Comment 1. *额哦, 都没有引入, 这个 ppt 真的是...*

1.1 问题描述

简单来说, 给定一个图和图中的一个顶点, 我们要找到这个起点到其他节点的最短路径. 只不过说, 这里可以有很多种类别, 比如说

1. 是有向图还是无向图?
2. 是带权图还是无权图?

我们首先处理最为简单的情况, viz. 无向无权图.

1.2 一些性质

Theorem 1 (优化子结构)**.** *两个节点之间的一个最短路径, 包含着其他的最短子路径.*

viz. 给定一个 u, v 之间的最短路径 p i.e. $u \overset{p}{\rightsquigarrow} v$, 对于 p 上的任意两个点 μ, ν , p 中有一个子路径 p' 使得 $\mu \overset{p'}{\rightsquigarrow} \nu$

简单证明一下. 优化子结构都使用反证法, 我们说, μ, ν 之间如果说, p' 并不是最短的, 设最短的那个为 p''. 那么我们能够构造出一个比 p 要短的路径:

设 p_1 有 $u \overset{p_1}{\rightsquigarrow} \mu$ 然后有 p_2: $\nu \overset{p_2}{\rightsquigarrow} v$. 那么这样表示 $u \overset{p_1}{\rightsquigarrow} \mu \overset{p''}{\rightsquigarrow} \nu \overset{p_2}{\rightsquigarrow} v$. 这就是一个比 p 要短的路径. □

Theorem 2 (三角不等式)**.** *设 $\delta(u, v)$ 表示 u, v 之间的最短路径的长度 (aka 权重). 那么我们有:*

$$\delta(u, v) \leq \delta(u, v) + \delta(x, v)$$

Proposition 1. *负圈. 如果说, 存在一个权重为负值的圈, 那么我们说, 很可能不存在最短路径.*

1.3 松弛

最短路径的核心技术就是松弛, 这点比较好理解吧. 松弛的对象是两个顶点, 已知条件是这两个顶点之间的边的权重, 以及, 当前 "已知的" 到两个顶点的距离.

Definition 1. *松弛. 给定两个顶点 u, v, 松弛的操作是说, if $d(u) + w(u, v) \leq d(v)$ 我们就将这个 $d(v)$ 更新为 $d(u) + w(u, v)$ (并且可以进行记录, 记录 v 的 "上一个节点" 应该是 u)*

我们这里稍微写一下代码

```
void Relax (u,v,w(u,v)){
    if (d[v] > d[u] + w(u,v))
    d[v] = d[u] + w(u,v);
}
```

1.4 Bellman-Ford algorithm

2 intro and prequisition

2.1 Notation

我们研究最短路径的话, 我们必然会面对图的各种参数, 因为我们当然是在有权图上面寻找最短路径的, 如果说是那种将路径长度定义为路径所经过的节点个数的话, 正如 22 所讲的那样的话, 这种就不在我们这次的研究范围内了. 于是我们给定的是**有权, 有向图**.

Definition 2. *A graph is abbreviated as* $G = (V, E)$ *, 这是我们已经熟知的.*

Definition 3. *一个 path 记为* p *, 可以写为* $\langle v_0, v_1, \cdots, v_n \rangle$ *, 为了突出其终点和起点, 一个 path 可以记为*

$$p : u \rightsquigarrow v$$

Definition 4. $w : E \to \mathbb{R}, (u, v) \mapsto w(u, v)$ *, 将权重以函数的方式写出来当然是为了严谨. 虽然在一些人看来可能是脱裤子放屁, 但其实有很多东西的定义都是这样用函数定义的. 同时也定义了 path 的权重* $p \mapsto w(p) = \sum_{i=1}^{\infty} w(v_i)$

Definition 5. *最短路径的记号:*

$$\delta(u, v) = \begin{cases} \min\{w(p) : p : u \rightsquigarrow v\}, & \textit{if there is a path from } u \textit{ to } v \\ \infty, & \textit{otherwise} \end{cases}$$

2.2 some variants of single-source shortest paths

我们目前的问题称为 single-source shortest paths. 对于有权有向图, 给定了一个 source, 我们要找出从 source 到其他点的最短路径的大小, 以及可以求解出这个路径. single-source shortest paths 问题有多种变体, 当然这里只是介绍一下

Single-destination shortest-paths problems 给定一个图, 给定一个终点, 找到各个顶点到这个终点 (let's say t) 的最短路径. 我们将每一个边逆向, 我们就能将其转换为一个 single-source 问题

single-pair shortest-path problem 给定两个顶点 u, v , 我们说, 要找到这两个顶点之间的最短路径, 总之, 因为我们说, 优化子结构, 导致了求出了很多 u 为起点 μ 为终点的路径 (μ 是某些节点). 所以说, 我们不如直接面对 single source 问题.

All-pairs shortest-paths problem `Floyd`算法用来解决这个问题.

2.3 optimal structure of shortest paths

rt. 最短路径具有优化子结构, 即,

Theorem 3 (优化子结构)**.** *两个节点之间的一个最短路径, 包含着其他的最短子路径.*

viz. *给定一个* u, v *之间的最短路径* p *i.e.* $u \overset{p}{\rightsquigarrow} v$ *, 对于* p *上的任意两个点* μ, ν *,* p *中有一个子路径* p' *使得* $\mu \overset{p'}{\rightsquigarrow} \nu$

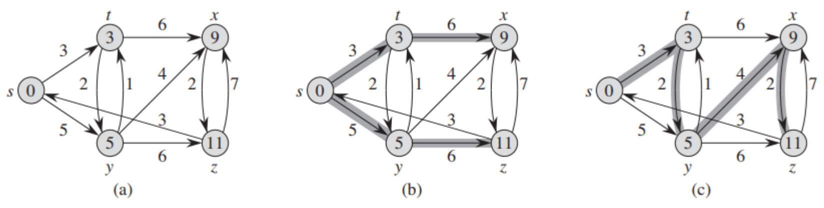


图 1: example of relaxation

简单证明一下. 优化子结构都使用反证法, 我们说, μ, ν 之间如果说, p' 并不是最短的, 设最短的那个为 p''. 那么我们能够构造出一个比 p 要短的路径:

设 p_1 有 $u \overset{p_1}{\rightsquigarrow} \mu$ 然后有 p_2: $\nu \overset{p_2}{\rightsquigarrow} v$. 那么这样表示 $u \overset{p_1}{\rightsquigarrow} \mu \overset{p''}{\rightsquigarrow} \nu \overset{p_2}{\rightsquigarrow} v$. 这就是一个比 p 要短的路径. □

2.4 representation of shortest paths

这涉及到前面我还没看的部分, 即无向无权图的最短路径求解. 这里我们涉及 **predecessor**, 符号 π 的出现说明其和 **predecessor** 有关. 比如说: $v.\pi$ 是 v 的一个前驱

Definition 6 (predecessor subgraph)**.** *$G_\pi = (V_\pi, E_\pi)$ 是 predecessor subgraph , 其中*

$$V_\pi = \{v \in V : v.\pi \neq \varnothing\} \cup \{s\}$$

s 是 source, 并且

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

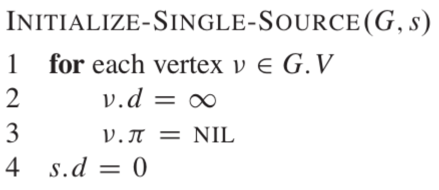
直观的来说, 就是这个 G_π 是通过 π 这个东西导出的. 那么就是, V_π 就是说 π 说明的, 能够抵达的顶点. 然后 E_π 就是一些边. 我们能够知道, 这个 G_π 是一个树. 因为一个节点只有前驱.

single-source shortest paths 问题其实就是求出下面这个子图
$G' = (V', E')$
V' 是所有能够达到的点的集合, 即 $\delta(s, v) \neq \infty$
并且 G' 是一个树, 并且这个树上的任意一个节点 v , 则 v 和 s 之间的距离最短.

2.5 relaxation

2.5.1 initializing single source

我们使用 $v.d$ 表示目前已知的 v, s 之间的距离. 称为 **shortest path estimate**. 这时可以补充上面的 $v.\pi$ 了: $v.\pi$ 就是目前已知的" 最短路径上" v 的前驱.



```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi = \text{NIL}$
4  $s.d = 0$
```

图 2: 初始化的伪代码

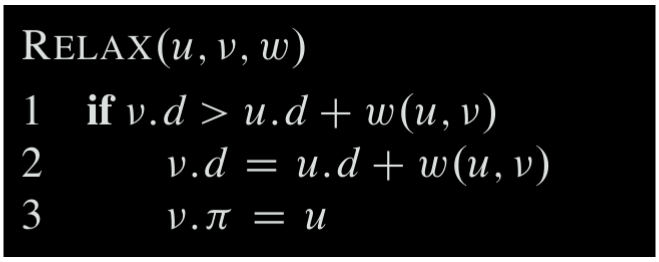
2.5.2 relaxation: code and definition

最短路径的核心技术就是松弛, 这点比较好理解吧. 松弛的对象是两个顶点, 已知条件是这两个顶点之间的边的权重, 以及, 当前 "已知的" 到两个顶点的距离.

Definition 7. *松弛. 给定两个顶点 u, v, 松弛的操作是说, if $d(u) + w(u, v) \leq d(v)$ 我们就将这个 $d(v)$ 更新为 $d(u) + w(u, v)$ (并且可以进行记录, 记录 v 的 "上一个节点" 应该是 u)*

我们这里稍微写一下代码

```
void Relax (u,v,w(u,v)){
    if (d[v] > d[u] + w(u,v))
    d[v] = d[u] + w(u,v);
    d.\pi= u;
}
```



```
RELAX($u, v, w$)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

图 3: code

意思即为, u 到 v 的一个松弛, 如果说走到 u 然后走到 v 的长度更短, 我们就更新 $v.d$ 目前最短路径长度; $v.\pi$ v 的前驱, i.e. 更新为 u 而 `relaxation` 只需要常数时间.

下面是一点 quote

> Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest path estimates and predecessors change. The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges. Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs relax each edge exactly once. The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

2.6 一些性质

Theorem 4 (优化子结构). *两个节点之间的一个最短路径, 包含着其他的最短子路径.*

viz. 给定一个 u, v 之间的最短路径 p i.e. $u \overset{p}{\rightsquigarrow} v$, 对于 p 上的任意两个点 μ, ν , p 中有一个子路径 p' 使得 $\mu \overset{p'}{\rightsquigarrow} \nu$

简单证明一下. 优化子结构都使用反证法, 我们说, μ, ν 之间如果说, p' 并不是最短的, 设最短的那个为 p''. 那么我们能够构造出一个比 p 要短的路径:

设 p_1 有 $u \overset{p_1}{\rightsquigarrow} \mu$ 然后有 p_2: $\nu \overset{p_2}{\rightsquigarrow} v$. 那么这样表示 $u \overset{p_1}{\rightsquigarrow} \mu \overset{p''}{\rightsquigarrow} \nu \overset{p_2}{\rightsquigarrow} v$. 这就是一个比 p 要短的路径. □

Theorem 5 (三角不等式). *设 $\delta(u, v)$ 表示 u, v 之间的最短路径的长度 (aka 权重). 那么我们有:*

$$\delta(u, v) \leq \delta(u, v) + \delta(x, v)$$

Proposition 2. 负圈. 如果说, 存在一个权重为负值的圈, 那么我们说, 很可能不存在最短路径.

2.7 an outline copied from textbook

Section 24.1 presents the Bellman-Ford algorithm, which solves the single-source shortest-paths problem in the general case in which edges can have negative weight. The Bellman-Ford algorithm is remarkably simple, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 24.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 24.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 24.4 shows how we can use the Bellman-Ford algorithm to solve a special case of linear programming. Finally, Section 24.5 proves the properties of shortest paths and relaxation stated above.

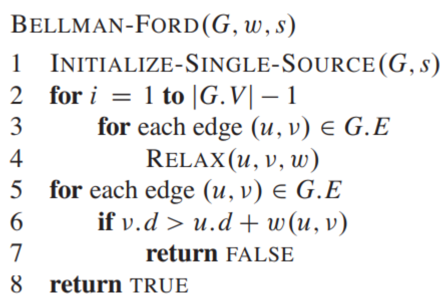
We require some conventions for doing arithmetic with infinities. We shall assume that for any real number $a \neq -\infty$, we have $a + \infty = \infty + a = \infty$. Also, to make our proofs hold in the presence of negative-weight cycles, we shall assume that for any real number $a \neq +\infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$

All algorithms in this chapter assume that the directed graph G is stored in the adjacency-list representation. Additionally, stored with each edge is its weight, so that as we traverse each adjacency list, we can determine the edge weights in $O(1)$ time per edge.

3 Bellman-Ford algorithm

3.1 an introd

3.2 algorithm



```
BELLMAN-FORD$(G, w, s)$
1  INITIALIZE-SINGLE-SOURCE$(G, s)$
2  for $i = 1$ to $|G.V| - 1$
3      for each edge $(u, v) \in G.E$
4          RELAX$(u, v, w)$
5  for each edge $(u, v) \in G.E$
6      if $v.d > u.d + w(u, v)$
7          return FALSE
8  return TRUE
```

图 4: pseus code of BF alg

Hey maybe we can rewrite the code.

```
boolen bellman_Ford (G , s , w) {
    for each v in V
        d[v] = infty;
    d[s] = 0;
```

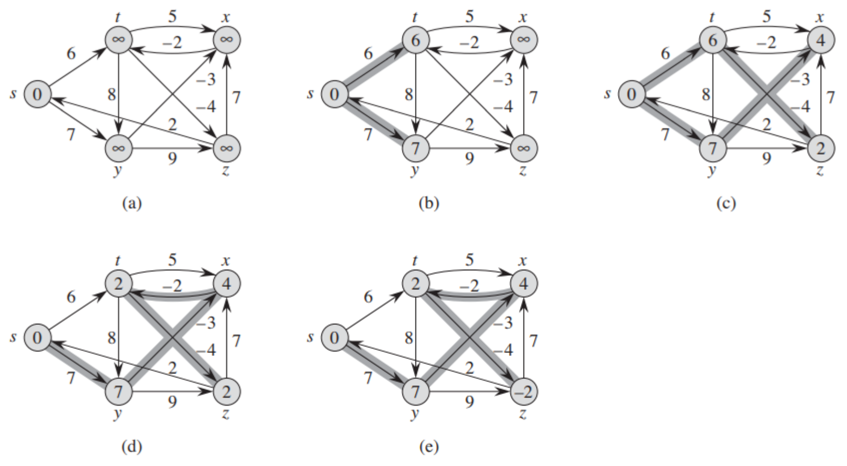


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

图 5: the procedure of bellman-Ford algorithm

```
    for every vertex in G {
        for each edge (u,v) in E {
            relax (u,v,w(u,v));
        }
    }
    for each edge (u,v) in E{
        if (d[v] > d[u] + w(u,v)){
            return NO_SOLUTION;
        }
    }
    reture true;
}
```

Comment 2. *Note that the algorithm returns a boolen value. And if it returns a* `false`*, then it means that the graph contains a negative weighted cycle which means that the solution does not exist. And if it returns a* `true` *value, then the solution does exist. The lengths of the shortest paths are stored in the d array.*

Figure 5 shows a simple procedure of bellman-ford algorithm that is carried on a simple graph. The picture on 'textbook' and those on ppt are the same. No pressure.

3.3 negative weighted cycles

The negative weighted cycles can make it that the solution does not exist.

So how does the algorithm detect the cycles?

You may see that in the algorithm, the condition `d[v] > d[u]+w(u,v)`

3.4 the prf of algorithm

Theorem 6. *given $G = (V, E)$, and a source s. It is known that the G has no negative weighted cycles.*

After the `line 5-9` *have been carried out for $|V| - 1$ times, for every reachable vertice v:*

$$d[v] = \delta(s, v)$$

proof by contradiction. 如果说 $d[v] \neq \delta(s, v)$, 那么, 就有 $d[v] < \delta(s, v)$ because $d[v] \leq \delta[s, v]$ holds all the time.

In the procedure of algorithm, there exist a vertex, let's say u, s.t. the following equality holds:

$$d[v] = d[u] + w(u, v)$$

Next, use the triangle inequality.

$$d[v] < \delta(s, v) \leq \delta(s, u) + w(u, v) \leq d[u] + w(u, v)$$

which leads to that $d[v] < d[u] + w(u, v)$

Next the proof on the ppt remains unreadable. Maybe it is saying that very `for` cycle makes some shortest paths' lengths are added with 1. $\square$

proof of negative weighted cycles. Suppose that there is a negative weighted cycle denote as $c = \langle v_0, \cdots, v_k \rangle$. It is indeed a cycle with length k, because $v_0 = v_k$. We have:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

Next we are going to prove it by contradiction. If the algorithm returns true. Then we have

$$d[i] \leq d[i-1] + w(v_{i-1}, v_i)$$

where $i = 1, 2, 3 \cdots, k$. Sum them up. Then we have:

$$\begin{aligned} \sum_{i=1}^{k} d[i] &\leq \sum_{i=1}^{k} (d[i-1] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[i-1] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Since $v_0 = v_k$, $\sum_{i=1}^{k} d[i-1] = \sum_{i=1}^{k} d[i]$. Then we have

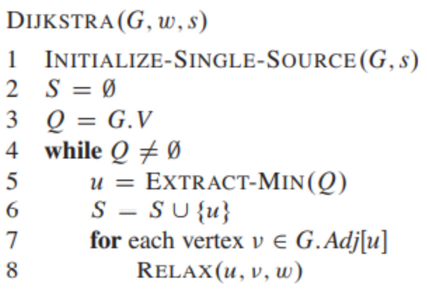
$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

which contradicts to the fact that the cycle has negative weight, viz. $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ $\square$

3.5 some blahblah

The time complexity of Bellman-Ford algorithm is $O(VE)$, which is easily to deduce. Trivial!

You can use topo ordering (whatever it calls) to make the algorithm more efficient.



```
DIJKSTRA$(G, w, s)$
1  INITIALIZE-SINGLE-SOURCE$(G, s)$
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN$(Q)$
6      $S - S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX$(u, v, w)$
```

图 6: Dijkstra

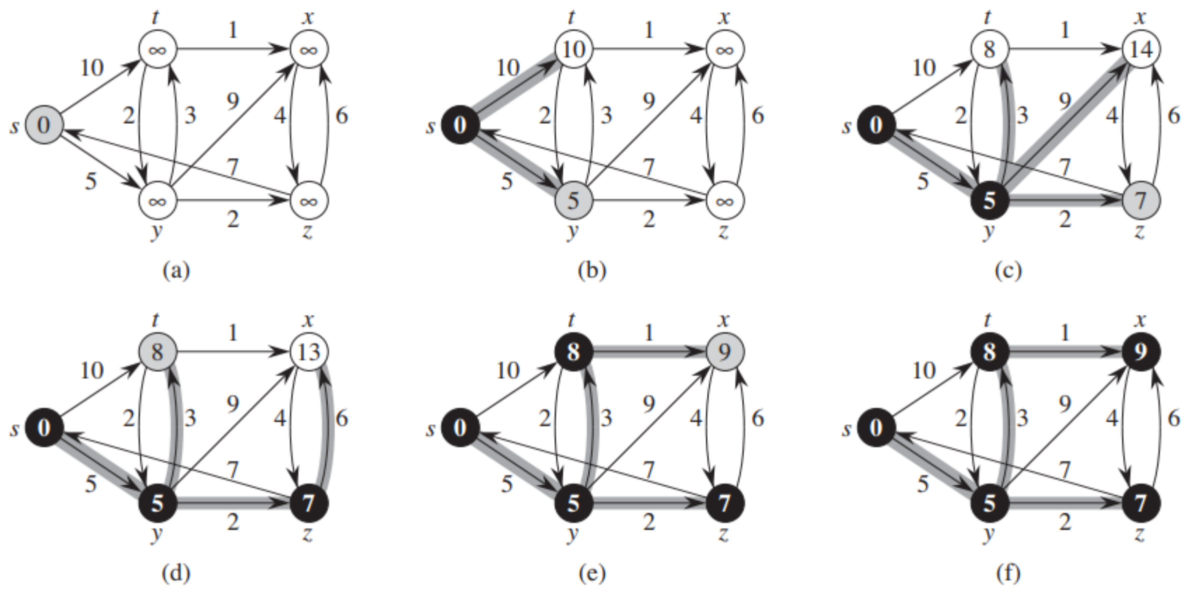


图 7: Dijkstra procedure

4 Dijkstra algorithm

4.1 A review

Dijkstra algorithm requires that the graph has no negative weighted edges. Under such circumstance, the Dijkstra can reach a much more efficient algorithm.

The procedure is like prim algorithm for smallest spanning tree.

4.2 algorithm

Figure 6 shows the algorithm in the 'textbook'. The Q mentioned is a min-priority queue of vertices, keyed by their d values.

Comment 3. *Note that there is a typo:* `line 6` *should be* $S = S \cup \{u\}$

Figure 7 shows a procedure of how Dijkstra proceed to the answer.

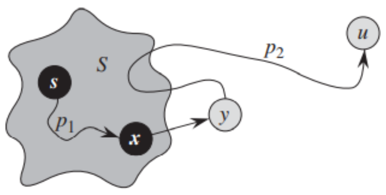


Figure 24.7 The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. We decompose a shortest path p from source s to vertex u into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, where y is the first vertex on the path that is not in S and $x \in S$ immediately precedes y. Vertices x and y are distinct, but we may have $s = x$ or $y = u$. Path p_2 may or may not reenter set S.

图 8: proof of Dji

4.3 The prf of algorithm

证明. We need to prove that $d[u] = \delta(s, u)$ if u is extracted from Q.

We prove by contradiction and a little bit induction maybe. If $d[u] \neq \delta(s, u)$, then $d[u] > \delta(s, u)$

I mean, emmm, is that possible?

Have a look at Figure 8. There exist $p : s \overset{p}{\leadsto} u$, p is shortest. We say that the shortest to the vertices in S have been found, that is supposition from the induction. That is saying

$$\forall v \in S, d[v] = \delta(s, u)$$

Let's say that a vertex x is in S. And let's say that vertex y is outside of the S, and moreover, $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$ is the shortest path to u.

We are going to do is to find a contradiction. How? The length of the shortest path is

$$\delta(s, x) + w(x, y) + \delta(y, u) = d[y] + \delta(y, u) < d[u]$$

which is not possible to be lower than the $d[u]$, because it is the shortest member amoung d's.

$$d[u] \leq d[y]$$

□

Comment 4. *Here is a very crucial step: we assume that for the vertices in S, we have*

$$\forall x \in S, d[x] = \delta(s, x)$$

which is part of the induction. As for the complete proof, you can check out the proof in the 'textbook', where the author provide a rigorous procedure of proof which consist of three sections that are introduced in the former chapter of the book, viz. ***initialization, maintainence, termination****.*

4.4 the analysis of Dijkstra algorithm

5 all-pairs shortest-paths algorithm

问题描述: Given a weighted acyclic graph $G = (V, E)$. Find the shortest paths of all pair (u, v)

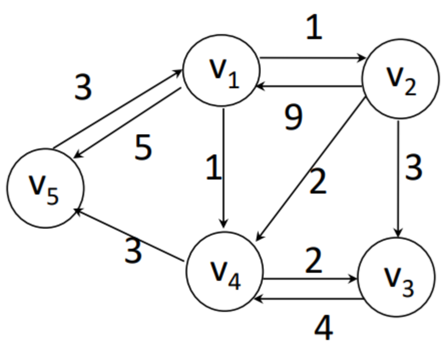


图 9: a graph

Here, for the Floyd algorithm, the Graph contains negative weighted edge but does not contain the negative weighted cycles.

It is obvious that you can just carry out the single-source algorithm on each of the vertices.

BUT, what we are introducing here, is the Floyd algorithm, which looks more elegent and neat (Maybe).

5.1 the matrix and graph

As we have learnt in the Graph Theory, the edges in the graph can be expressed in the form of matrix.

Example 1. *Here is a matrix:*

$$\begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

The corresponding graph is showed in Figure 9

w suit that

$$w_{ij} = \begin{cases} 0 & i = j \\ \text{weight} & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

The idea of Floyd algorithm is actually dynamic programming. Here we can remind of how to design a dp algorithm or what need to be verified in dp.

1. optimal substructure
2. overlapped subproblem
3. 自底向上地计算优化解的代价并且将其保存. 并且获取构造最优解的信息.
4. construct the solution based on the information given in the previous step.

5.2 前置工作

The output of the all-pairs shortest-paths algorithms presented here is an $n \times n$ matrix $D = (d_{ij})$, where d_{ij} stands for the length of shortest path from i to j. And if we are going to store the information of the shortest paths, we need not the 1-dimension array but another matrix, denoted as $\Pi = (\pi_{ij})$ which is called **predecessor matrix**, where π_{ij} stands for the predecessor of j in the shortest path from i to j. It is clear that, if you want to recover the path, let's say $i \rightsquigarrow j$ from the matrix, you need to find π_{ij} and then find $\pi_{i,\pi_{ij}}$

In the previous chapter, we know that π array can induce a shortest path tree G_π. Similarly, the i th row of the Π matrix can induce a shortest path tree.

In the textbook, the **predecessor subgraph** with respect to i is defined as G_{π_i} that is the subgraph induced by i th row of Π.

We can list all these definitions. I mean it, that is good habit to do that.

Definition 8 (the solution of apsp). *the solution value is stored in $D = d_{ij}$ and d_{ij} once the algorithm terminates.*

Definition 9 (predecessor matrix). *The information of the solution are store in a matrix called predecessor matrix, denoted as $\Pi = (\pi_{ij})$, where π_{ij} stands for the predecessor of the j in the shortest path from i to j*

Definition 10 (predecessor subgraph). *好像没什么用，这玩意.*

$$V_{\pi_i} = \{j \in V : \pi_{ij} \neq \varnothing\} \cup \{i\}$$

and

$$E_{\pi_i} = \{(\pi_{ij}, j) : j \in V_{\pi_i} - \{i\}\}$$

5.2.1 A chapter outline copied from the book

Section 25.1 presents a dp algorithm based on matrix multiplication to solve the all-pairs shortest-paths problem. Using the technique of **repeated squaring** , we can achieve a running time of $\Theta\left(V^3 \log V\right)$.

Section 25.2 presents Floyd-Warshell algorithm, which runs in time $\Theta\left(V^3\right)$. It also covers the problem of finding the transitive closure of a directed graph.

Finally, section 25.3 presents Johnson's algorithm, which solves the apsp problme in $O\left(V^2 \log V + VE\right)$ time and is a good choice for large, sparse graphs.

Comment 5. *Unfortunately, here we are only dealing with the Floyd algorithm. Johnson's might looks attractive, but, emm, sorry.*

5.2.2 some conventions

The capital letter like W, L, D is used to denote a matrix. And the entries of the matrix is denoted as w_{ij}, or l_{ij}, where the subscribts i, j stands for the row and column of the entry.

Moreover, some matrices may have parenthesized superscripts, like $L^{(m)} = \left(l_{ij}^{(m)}\right)$, which is, used to indicate iterates.

5.3 An approuch

a review of dp Three steps:

1. characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion.

the structure of a shortest path The content on the 'textbook' is like blahblah and so on and so on. We just need to remember the theorem mentioned before.

Like, give two vertices i, j and then k is the predecessor of j, then the shortest path from i to j can be expressed as $i \overset{p'}{\rightsquigarrow} k \to j$ where p' is a subpath of p, and is a shortest path too. Then we have:

$$\delta(i, j) = \delta(i, k) + w_{kj}$$

where the length of p is m and the length of subpath p' is $m - 1$

the recursive solution Hey, can you figure out why we mention the length of the path?

What we are going to do is to **iterate** on the length of the path. That is, if the **shortest paths** with length **at most** $m-1$ are known, we can use them to figure out the **shortest paths** with length **at most** m

These sentences above are important. Let me have a few seconds to digest this huge amount information.

It is actually the main idea. Not so difficult, right?

And next step, how are we gonna iterate? Apparently, we can use the equation in the previous paragraph. That is

$$\delta(i, j) = \delta(i, k) + w_{kj}$$

Oh, the solution is stored in a matrix L. I forget to mention. Anyway,

$$l_{ij}^{(m)} = \min_{k \in V} \{\delta(i, k) + w_{kj}\}$$

The initial matirx $L^{(0)}$ is defined as follows:

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

Apparently, the number of edges in a shortest can not be longer than the $n - 1$ (there is no negative weighted cycle). So the iteration can terminate when it reach $n - 1$

5.4 the Floyd algorithm

It is quite surprising but we are now heading to the Floyd algorithm.

With similar approuch, but with a better recursive method, the Floyd algorithm runs a little bit faster.

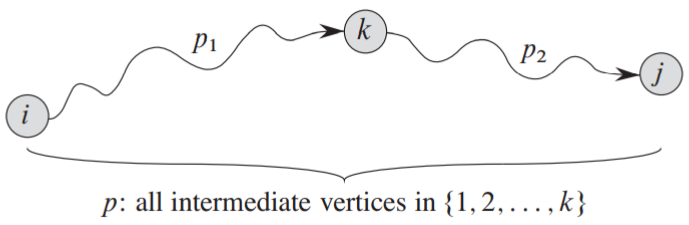
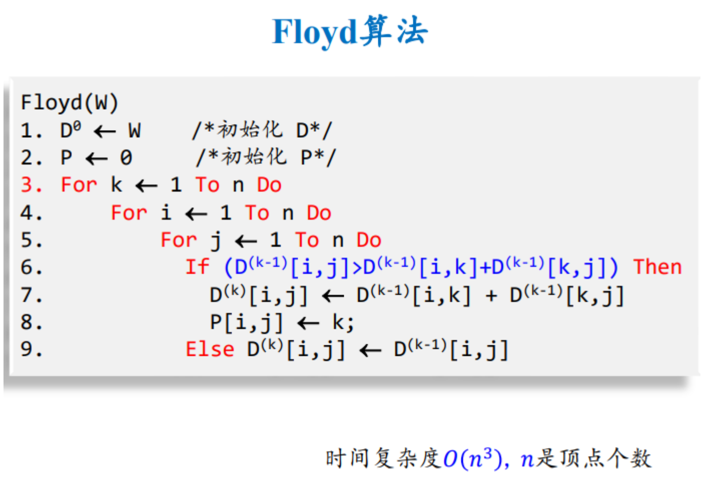


图 10: recursive structure of Floyd



Floyd算法

```
Floyd(W)
1. D⁰ ← W       /*初始化 D*/
2. P ← 0        /*初始化 P*/
3. For k ← 1 To n Do
4.     For i ← 1 To n Do
5.         For j ← 1 To n Do
6.           If (D^(k-1)[i,j]>D^(k-1)[i,k]+D^(k-1)[k,j]) Then
7.             D^(k)[i,j] ← D^(k-1)[i,k] + D^(k-1)[k,j]
8.             P[i,j] ← k;
9.           Else D^(k)[i,j] ← D^(k-1)[i,j]
```

时间复杂度$O(n^3)$, n是顶点个数

图 11: code of Floyd

the recursive solution in Floyd Figure 10 shows the recursive structure of shortest paths. But here we iterate on what ?

You can try to figure that. Moreover, what does ‘iteration’ mean in the first place? So what does the parenthesized superscripts means here? the k in $D^{(k)}$ is actually suggesting the shortest paths only path through some vertices in $\{1, 2, \cdots, k\}$

Anyway, the recursive function is like

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & k \geq 1 \end{cases}$$

the psedu code The code is shown in Figure 11

5.5 获取构造最优解的信息，构造最优解

We can write some codes here.

```
int floyd (W) { // W is a matrix
    D = W;
    P = 0; // P is a matrix
    for (k = 1 to n){
        for (i = 1 to n){
            for (j = 1 to n){
                if (D[i][j] > D[i][k]+D[k][j]){
                    D[i][j] = D[i][k]+D[k][j];
                    P[i][j] = k;
                }
            }
        }
    }
}
```

The code to print

```
int print_path (index q , r) {
    if (P[q,r] != 0){
        print_path (q, P[q,r]);
        printf ("v", P[q,r]);
        print_path(P[q,r],r);
        return ;
    }
    esle return -1;
}
```

6 mf

6.1 some definition about flow

2) capacity of edges
3) flow’s property

source and sink 首先我们给定一个图, 其不存在相反的边, viz. $(u, v) \in E \implies (v, u) \notin E$, 其次, 这个图是一个交换图 (commute diagram) , 这是范畴论的概念. 起点称为 s, source, 终点为 t, (s for sink t for terminal)

capacity 存在一个权值函数 w, 使得 $\exists (u, v) \in E, w(u, v) > 0$. 这个值称为边的 capacity, 为什么起这个名字呢? 这是因为, 我们要定义一个流, 这个流的意思是, 我们从起点出发, 有一堆货物, 或者说是水, 要流到终点, 这个流的量就是 flow 的值. 边的权重就是说这个边 viz. 承载流的能力.

flow’s property 直观的讲, 我们的货物最后可别停在中间, 应流向终点, 这说明: 对于中间的节点, “进来的” 等于 “出来的”, viz. 对于节点 v 有:

$$\sum f(u_i, v) = \sum f(v, w_i)$$

LHS 是进的, RHS 是出的. 我们定义 $\sum f(s, v_i)$ (或者是 $\sum f(v_i, t)$, 不妨证明一下为什么这两个是相等的) 为 flow 的值, 记为 $|f|$

6.2 residual networks

我们给定一个图和一个 flow, 我们下面能够自然的引出 residual networks 的定义. 记为 G'_f 这个图中的边, 表示了在 f 已经存在的情况下, 各个边的 capacity. 并且存在 G 中原本没有的边, 用其相反的方向表示 "减少原 flow 某个边的值".

一个 flow, f 是使用函数来表示的.

$$f : E \to \mathbb{R}, (u, v) \mapsto f(u, v)$$

其中 $f(u, v) < c(u, v)$, 不然的话直接爆炸了.

Definition 11 (residual graph)**.**

$$G_f = \{e \in E : c(e) - f(e) > 0\}$$

并且*:*

$$c_f\left(u, v\right) = \begin{cases} c(u, v) - f(u, v) & \textit{if } \ (u, v) \in E \\ \\ f\left(v, u\right) & \textit{otherwise} \end{cases}$$

Comment 6. *注意上面边的反向. 这说明 residual graph 上的流, 最多可以将原 flow 的边给抵消掉. 并且, residual graph 的 residual edge 的数量变多了, E 中的一个边最多在 E_f 中贡献两个. 于是有:*

$$|E_f| \leq 2|E|$$

Trivial !

A flow in a residual network provides a roadmap for adding flow to the original flow network. 我们现在想做的, 就是对当前这个流, 进行一个增广. 当其足够大的时候, 他就是最大流了 (确信).

Observe that the residual network G_f is similar to a flow network with capacities given by c_f . It does not satisfy our definition of a flow network because it may contain both an edge (u, v) and its reversal (v, u). Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities c_f in the network G_f.

6.3 augmented flow

Definition 12 (augmented flow)**.** *Let's denote a flow on the residual graph G_f as f'. Meanwhile augmented flow of the original graph G, which augmented by f' with the respective to f, is denoted as $f \uparrow f'$.*

$$f \uparrow f'\left(u, v\right) = \begin{cases} f(u, v) + f'\left(u, v\right) - f\left(v, u\right), & \textit{if } \ (u, v) \in E \\ 0 & \textit{otherwise} \end{cases}$$

Let $G = (V, E)$ be a flow network with source s and sink t , and let f be a flow in G. Let G_f be the residual network of G induced by f , and let f' be a flow in G_f. Then the function $f \uparrow f'$ defined in equation (上面介绍的) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$

Comment 7. *The conclusion that $|f \uparrow f'| = |f| + |f'|$ needs proof.*

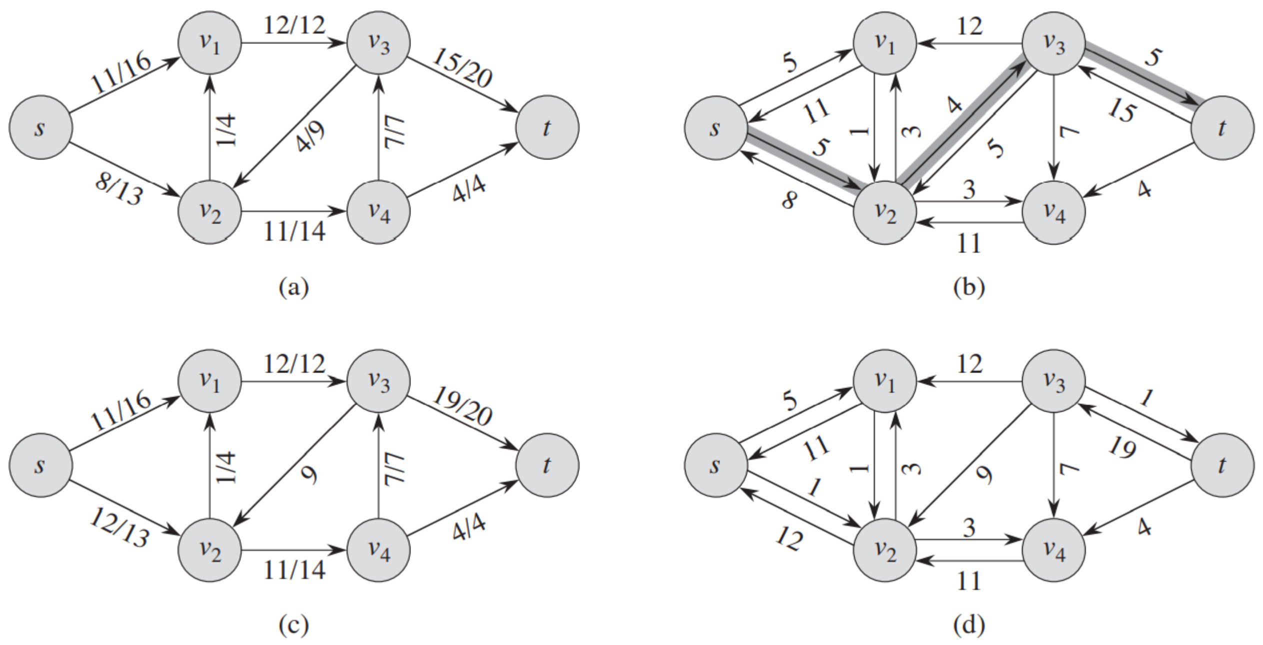


图 12: flow and augmenting path

Figure 12 是一个图例, 其中 (a) 是原图以及 flow, (b) 是 residual graph, 然后 (c) 是 augmented flow, (d) 是新的 flow 的对应的 residual graph

The intuition behind this definition follows the definition of the residual network. We increase the flow on (u, v) by $f'(u, v)$ but decrease it by $f'(v, u)$ because pushing flow on the reverse edge in the residual network means decreasing the flow in the original network.

Pushing flow on the reverse edge in the residual network is also known as **cancellation**. For example, if we send 5 crates of hockey pucks from u to v and send 2 crates from v to u, we could equivalently (from the perspective of the final result) just send 3 creates from u to v and none from v to u. Cancellation of this type is crucial for any maximum-flow algorithm.

6.4 augmenting paths

Smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

Lemma 1 (Lemma 26.2)**.** *Let $G = (V, E)$ be a flow network , let f be a flow in G , and let p be an augmenting path in G_f , Define a function $f_p : V \times V \to \mathbb{R}$ by*

$$f_p(u, v) = \begin{cases} c_f(p) & \textit{if } (u, v) \textit{ is on } p \\ 0 & \textit{otherwise} \end{cases}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$

The following corollary show that if we augment f by f_p, we get another flow in G whose value is closer to the maximum. Figure shows the result of augmenting the flow f from Figure by hte flow f_p in Figure and Figure shows the ensuing residual network.

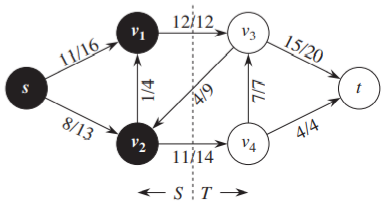


Figure 26.5 A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

图 13: a cut

Proposition 3 (Corollary 26.3)**.** *Let $G = (V, E)$ be a flow network, let f be a flow in G , and let p be an augmenting path in G_f . Let f_p be the flow defined above, and suppose that we augment f by f_p . Then the function $f \uparrow f_p$ is a flow in G with value: $|f \uparrow f_p| = |f| + |f_p| > |f|$*

6.5 cut of flow networks

How do we know Ford-Fulkerson method are proceed towards the right answer? The max flow min cut theorem tells us that the flow is maximum iff residual network contains no augmenting path. However, we shall introduce some definitions about **cut**.

Definition 13 (cut)**.** *A cut (S, T) is a partition of V of G, viz. $S \cup T = V$ while $S \cap T = \varnothing$*

With the definition of cut, we can define a **net flow** with the respect to the cut.

Definition 14 (net flow)**.** *Given a flow f, the net flow $f(S, T)$ with the respect to the cut (S, T) is defined to be*

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

Comment 8. *Doesn't it look like the definition of **flux** !?*

Definition 15 (capacity of cut)**.** *The capacity of (S, T) is*

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

The minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Figure 13 shows a example of cut. You could try to work out the capacity and net flow of (S, T) in the figure.

Lemma 2. *f is a flow in G with source s and sink t. (S, T) is an arbitrary cut of G. We have that*

$$f(S, T) = |f|$$

证明. The proof is intensionally left blank. □

Proposition 4. *For all f in G, $|f|$ is bounded above.*

证明.

$$\begin{aligned}|f| &= f(S,T)\\ &\leq \cdots\\ &\leq \cdots\\ &= c(S,T)\end{aligned}$$

□

6.6 Max flow min cut theorem

Theorem 7 (Max flow min cut theorem)**.** *f is a flow in $G=(V,E)$ with source s and sink t. We have three equivalent conditions:*

1. *f is at maximum.*
2. *The residual graph G_f has no augmenting paths.*
3. *$|f| = c(S,T)$ for some cut (S,T) of G.*

*proof by **miracle**.* We shall prove 1 to 2, 2 to 3, and then 3 to 1. We shall write the proof in the following enumeration.

1 to 2: The proof is easy. We shall use contradictions to prove it. If there exist an augmenting path then we can augment the original path, so that the original one is not at maximum.

2 to 3: If G_f has no augmenting paths. Then you can say that G_f at least have two components, which allow us to claim that 1. if $(u,v)\in E$ then $f(u,v)$ should be $c(u,v)$ since otherwise (u,v) would be in E_f. 有点难捏.

3 to 1: which is obvious, since the $|f|$ is bounded above, viz

$$|f| \leq c(S,T)$$

and if $|f| = c(S,T)$ then it is at maximum.

□

6.7 Ford-Fulkerson algorithm

Finally we are arriving here. Now we shall introduce the Ford-Fulkerson algorithm.

The algorithm select a augmenting path in the residual graph and then augment the original flow with the path. The procedure keep going until there is no augmenting path in the residual path, which means that the flow is at maximum.

Hey, maybe we can write some code.

```
int Ford-Fulkerson (G, s, t) {
    for each edge (u,v) in E{
        f.(u,v) = 0
    } // initialization
    while there exists an augmenting path in the residual graph{
        c_f(p) = lowest capacity in p;
```

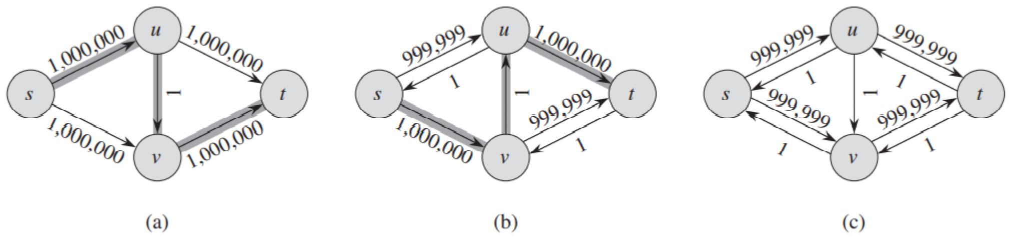


图 14: fig:an extreme example

```
        for (each edge (u,v) in p){
            if (u,v) in E   // if (u,v) in E , we augment it.
                f.(u,v) = f(u,v) + c_f(p);
            else            // if (v,u) not in E, we reduce it.
                f.(v,u) = f(v,u) - c_f(p);
        }
    }
}
```

Comment 9. *Note that in the code, no steps about the update of residual graph are written.*

6.8 the time complexity of Ford-Fulkerson algorithm

Usually the maximum-flow problem often arises with integer capacities. If the capacities are rational numbers, we can make them all integer with certain scalar. And every time the path is added to the original flow, the value of the flow grow at least one. So, if f^* stands for the maximum flow, the exercute times of adding paths is $|f^*|$ times.

We can use a linear time algorithm to find an augmenting path, but I don't know how to do that. Maybe, the readers can find out yourselves. Wonderful.

Anyway, we need $O(E)$ to find a path. So the total running time is $O(E|f^*|)$

Example 2. *Here is an extreme example show in the Figure 14 of how slow the Ford-Fulkerson can be.*

Hey maybe you can use some greedy approuch to make it better. Because choosing a path with capacity 1 *instead of choosing one with capacity* $1,000,000$ *seems absolutely stupid.*

Comment 10. *Some readers may have noticed that there are lots of definitions mentioned only once which precisely when it was introduced.*

They may be useless but that is not my fault.

7 Maximum bipartite matching

7.1 some definitions

Definition 16 (match)**.** *Let's say that M is a match of G. M is subset of E where for all edges in M, they have no common vertices, viz.*

$$\forall m_i, m_j \in M, \text{他们没有公共顶点}$$

总之我们的目的是要找出匹配边, 至于说匹配是用来干什么的? 我目前还不太清楚. Anyway, 下面给出一系列的定义.

Definition 17 (匹配)**.** 匹配是图的子图, 设为 $G' = (V', E')$, 其中 $V' = V$, $E' = \{e \in E : e\text{互不相邻}\}$

画图就不画了, 你可以自己画一画, latex 里画图好几把麻烦. 关键在于不相邻这个条件.

Definition 18 (最大匹配)**.** 最大匹配是边数最多的匹配

其实还有极大匹配, 就是说当前情况, 并不能再直接加边了的匹配, 但是明显, 极大匹配不一定是最大匹配.

Definition 19 (完美匹配)**.** 每一个顶点都是 M 中的边的顶点.

Definition 20 (匹配边, 非匹配边)**.** E' 即为匹配边, $E - E'$ 为非匹配边.

Definition 21 (交错路径)**.** 如果说 $p = \langle e_1, \cdots, e_n \rangle$ 中 e_i 交错地是匹配边和非匹配边, *i.e.* e_i 是匹配边, 那么 e_{i+1}, e_{i-1} 都是非匹配边, 那么称这个路径是交错路径.

Definition 22 (增广路径)**.** 如果说一个路径, 是交错路径, 并且非匹配边多于匹配边, 那么这个路径是增广路径.

进一步说，增广路径不是一个圈，并且，两端一定是非匹配边.

如果我们已知一个增广路径, 我们可以将其增广, viz. 将他们匹配和非匹配的身份调换, 这样匹配边数量加一.

7.2 algorithm

对于一个二部图, 我们有定理:

Theorem 8. 一个匹配是最大匹配 $\iff$ 其没有增广路径.

那么这个定理足以证明下面算法的正确性:

1 找到 augmenting 路径.[1]

2 将 augmenting 路径 augment.

3 直到不存在 augmenting 路径.

总之, 我们使用 ppt 上的语言将其再描述一遍:

1. 令 M 为空
2. 找到一个增广路径 P, 通过将其取反, 得到更大的匹配
3. 重复步骤 2, 直到我们找不到增广路径.

[1]一个不和匹配边相邻的边, 也能称为 augmenting 路径

	任务 1	任务 2	任务 3	任务 4
A	1	0	1	0
B	1	0	0	0
C	1	1	0	0
D	0	0	1	1

表 1: 1 表示能成, 0 表示不能成

7.3 application

Example 3. *现在要给 4 个工人 A, B, C, D 分配任务. 每一个工人可以完成特定的一些任务, 但是最多只能够接受一个任务, 总共有 4 个任务, 每个任务只能分配给一个工人. 请问最多能够分配多少个任务给工人.*

我们用 Table 1 来表示这些工人能做什么任务.

总之, 这个问题就能够转化为一个二部图求解最大匹配的问题, 工人之间没有连线, 任务之间也没有连线, 所以是二部图. 而连线代表的是任务分配. 就是说, 我们将某个工人和某个任务之间连了起来, 则说明我将这个任务分配了给他.

因为一个工人不能同时做两份工作, 也不能两个工人同时做一个任务, 那么我们这里就是求一个最大匹配.