

# 最短路

## 定义

(还记得这些定义吗？在阅读下列内容之前，请务必了解 [图论相关概念](#) [../concept/] 中的基础部分。)

- 路径
- 最短路
- 有向图中的最短路、无向图中的最短路
- 单源最短路、每对结点之间的最短路

## 性质

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的结点。

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的边。

对于边权为正的图，任意两个结点之间的最短路，任意一条的结点数不会超过  $n$ ，边数不会超过  $n - 1$ 。



## 记号

为了方便叙述，这里先给出下文将会用到的一些记号的含义。

- $n$  为图上点的数目， $m$  为图上边的数目；
- $s$  为最短路的源点；
- $D(u)$  为  $s$  点到  $u$  点的 **实际** 最短路长度；
- $dis(u)$  为  $s$  点到  $u$  点的 **估计** 最短路长度。任何时候都有  $dis(u) \geq D(u)$ 。特别地，当最短路算法终止时，应有  $dis(u) = D(u)$ 。
- $w(u, v)$  为  $(u, v)$  这一条边的边权。

## Floyd 算法

是用来求任意两个结点之间的最短路的。

复杂度比较高，但是常数小，容易实现。（我会说只有三个 `for` 吗？）

适用于任何图，不管有向无向，边权正负，但是最短路必须存在。（不能有个负环）



## 实现

我们定义一个数组  $f[k][x][y]$ ，表示只允许经过结点  $1$  到  $k$ （也就是说，在子图  $V' = 1, 2, \dots, k$  中的路径，注意， $x$  与  $y$  不一定在这个子图中），结点  $x$  到结点  $y$  的最短路长度。

很显然， $f[n][x][y]$  就是结点  $x$  到结点  $y$  的最短路长度（因为  $V' = 1, 2, \dots, n$  即为  $V$  本身，其表示的最短路径就是所求路径）。

接下来考虑如何求出  $f$  数组的值。

$f[0][x][y]$ ： $x$  与  $y$  的边权，或者  $0$ ，或者  $+\infty$ （ $f[0][x][y]$  什么时候应该是  $+\infty$ ？当  $x$  与  $y$  间有直接相连的边的时候，为它们的边权；当  $x = y$  的时候为零，因为到本身的距离为零；当  $x$  与  $y$  没有直接相连的边的时候，为  $+\infty$ ）。

$f[k][x][y] = \min(f[k-1][x][y], f[k-1][x][k] + f[k-1][k][y])$ （ $f[k-1][x][y]$ ，为不经过  $k$  点的最短路径，而  $f[k-1][x][k] + f[k-1][k][y]$ ，为经过了  $k$  点的最短路）。

上面两行都显然是对的，所以说这个做法空间是  $O(N^3)$ ，我们需要依次增加问题规模（ $k$  从  $1$  到  $n$ ），判断任意两点在当前问题规模下的最短路。

```

1 // C++ Version
2 for (k = 1; k <= n; k++) {
3     for (x = 1; x <= n; x++) {
4         for (y = 1; y <= n; y++) {
5             f[k][x][y] = min(f[k-1][x][y], f[k-1][x][k] + f[k-1][k][y]);
6         }
7     }
8 }

```

```

1 # Python Version

```



```

2  for k in range(1, n):
3      for x in range(1, n):
4          for y in range(1, n):
5              f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y])

```

因为第一维对结果无影响，我们可以发现数组的第一维是可以省略的，于是可以直接改成  $f[x][y] = \min(f[x][y], f[x][k] + f[k][y])$ 。

#### 证明第一维对结果无影响

我们注意到如果放在一个给定第一维  $k$  二维数组中， $f[x][k]$  与  $f[k][y]$  在某一行和某一列。而  $f[x][y]$  则是该行和该列的交叉点上的元素。

现在我们需要证明将  $f[k][x][y]$  直接在原地更改也不会更改它的结果：我们注意到  $f[k][x][y]$  的涵义是第一维为  $k-1$  这一行和这一列的所有元素的最小值，包含了  $f[k-1][x][y]$ ，那么我在原地进行更改也不会改变最小值的值，因为如果将该三维矩阵压缩为二维，则所求结果  $f[x][y]$  一开始即为原  $f[k-1][x][y]$  的值，最后依然会成为该行和该列的最小值。

故可以压缩。

```

1  // C++ Version
2  for (k = 1; k <= n; k++) {
3      for (x = 1; x <= n; x++) {
4          for (y = 1; y <= n; y++) {
5              f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
6          }
7      }
8  }

```

```

1  # Python Version

```

```
2  for k in range(1, n):
3      for x in range(1, n):
4          for y in range(1, n):
5              f[x][y] = min(f[x][y], f[x][k] + f[k][y])
```

综上时间复杂度是  $O(N^3)$ ，空间复杂度是  $O(N^2)$ 。

## 应用

? 给一个正权无向图，找一个最小权值和的环。

首先这一定是一个简单环。

想一想这个环是怎么构成的。

考虑环上编号最大的结点  $u$ 。

$f[u-1][x][y]$  和  $(u,x), (u,y)$  共同构成了环。

在 Floyd 的过程中枚举  $u$ ，计算这个和的最小值即可。

时间复杂度为  $O(n^3)$ 。

? 已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通。

该问题即是求 图的传递闭包。

我们只需要按照 Floyd 的过程，逐个加入点判断一下。

只是此时的边的边权变为  $1/0$ ，而取 **min** 变成了 **或** 运算。

再进一步用 bitset 优化，复杂度可以到  $O(\frac{n^3}{w})$ 。

```
1 // std::bitset<SIZE> f[SIZE];
2 for (k = 1; k <= n; k++)
3     for (i = 1; i <= n; i++)
4         if (f[i][k]) f[i] = f[i] | f[k];
```

## Bellman-Ford 算法

Bellman-Ford 算法是一种基于松弛（relax）操作的最短路算法，可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断。

在国内 OI 界，你可能听说过的“SPFA”，就是 Bellman-Ford 算法的一种实现。

### 流程

先介绍 Bellman-Ford 算法要用到的松弛操作（Dijkstra 算法也会用到松弛操作）。



对于边  $(u, v)$ ，松弛操作对应下面的式子： $dis(v) = \min(dis(v), dis(u) + w(u, v))$ 。

这么做的含义是显然的：我们尝试用  $S \rightarrow u \rightarrow v$ （其中  $S \rightarrow u$  的路径取最短路）这条路径去更新  $v$  点最短路的长度，如果这条路径更优，就进行更新。

Bellman-Ford 算法所做的，就是不断尝试对图上每一条边进行松弛。我们每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止。

每次循环是  $O(m)$  的，那么最多会循环多少次呢？

在最短路存在的情况下，由于一次松弛操作会使最短路的边数至少  $+1$ ，而最短路的边数最多为  $n - 1$ ，因此整个算法最多执行  $n - 1$  轮松弛操作。故总时间复杂度为  $O(nm)$ 。

但还有一种情况，如果从  $S$  点出发，抵达一个负环时，松弛操作会无休止地进行下去。注意到前面的论证中已经说明了，对于最短路存在的图，松弛操作最多只会执行  $n - 1$  轮，因此如果第  $n$  轮循环时仍然存在能松弛的边，说明从  $S$  点出发，能够抵达一个负环。

#### ⚠ 负环判断中存在的常见误区

需要注意的是，以  $S$  点为源点跑 Bellman-Ford 算法时，如果没有给出存在负环的结果，只能说明从  $S$  点出发不能抵达一个负环，而不能说明图上不存在负环。

因此如果需要判断整个图上是否存在负环，最严谨的做法是建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman-Ford 算法。

## 代码实现

### 📝 参考实现

```
1 // C++ Version
```

```
2 struct edge {
3     int v, w;
4 };
5 vector<edge> e[maxn];
6 int dis[maxn];
7 bool bellmanford(int n, int s) {
8     memset(dis, 63, sizeof(dis));
9     dis[s] = 0;
10    bool flag;
11    for (int i = 1; i <= n; i++) {
12        flag = false;
13        for (int u = 1; u <= n; u++) {
14            for (auto ed : e[u]) {
15                int v = ed.v, w = ed.w;
16                if (dis[v] > dis[u] + w) {
17                    dis[v] = dis[u] + w;
18                    flag = true;
19                }
20            }
21        }
22        // 没有可以松弛的边时就停止算法
23        if (!flag) break;
24    }
25    // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
26    return flag;
27 }
```



```
1 # Python Version
2 class Edge:
3     v = 0
```



```
4     w = 0
5
6     e = [[Edge() for i in range(maxn)] for j in range(maxn)]
7     dis = [63] * maxn
8
9     def bellmanford(n, s):
10         dis[s] = 0
11         for i in range(1, n + 1):
12             flag = False
13             for u in range(1, n + 1):
14                 for ed in e[u]:
15                     v = ed.v; w = ed.w
16                     if dis[v] > dis[u] + w:
17                         flag = True
18             # 没有可以松弛的边时就停止算法
19             if flag == False:
20                 break
21         # 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
22         return flag
```

## 队列优化：SPFA

即 Shortest Path Faster Algorithm。



很多时候我们并不需要那么多无用的松弛操作。

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。

那么我们用队列来维护“哪些结点可能会引起松弛操作”，就能只访问必要的边了。

SPFA 也可以用于判断  $s$  点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少  $n$  条边时，说明  $s$  点可以抵达一个负环。

#### 参考实现

```
1 // C++ Version
2 struct edge {
3     int v, w;
4 };
5 vector<edge> e[maxn];
6 int dis[maxn], cnt[maxn], vis[maxn];
7 queue<int> q;
8 bool spfa(int n, int s) {
9     memset(dis, 63, sizeof(dis));
10    dis[s] = 0, vis[s] = 1;
11    q.push(s);
12    while (!q.empty()) {
13        int u = q.front();
14        q.pop(), vis[u] = 0;
15        for (auto ed : e[u]) {
16            int v = ed.v, w = ed.w;
17            if (dis[v] > dis[u] + w) {
18                dis[v] = dis[u] + w;
19                cnt[v] = cnt[u] + 1; // 记录最短路经过的边数
20                if (cnt[v] >= n) return false;
21                // 在不经负环的情况下，最短路至多经过 n - 1 条边
22                // 因此如果经过了多于 n 条边，一定说明经过了负环
            }
```

```
23         if (!vis[v]) q.push(v), vis[v] = 1;
24     }
25 }
26 }
27 return true;
28 }
```

```
1  # Python Version
2  class Edge:
3      v = 0
4      w = 0
5
6  e = [[Edge() for i in range(maxn)] for j in range(maxn)]
7  dis = [63] * maxn; cnt = [] * maxn; vis = [] * maxn
8
9  q = []
10 def spfa(n, s):
11     dis[s] = 0; vis[s] = 1
12     q.append(s)
13     while len(q) != 0:
14         u = q[0]
15         q.pop(); vis[u] = 0
16         for ed in e[u]:
17             if dis[v] > dis[u] + w:
18                 dis[v] = dis[u] + w
19                 cnt[v] = cnt[u] + 1 # 记录最短路经过的边数
20                 if cnt[v] >= n:
21                     return False
22     # 在不经过负环的情况下，最短路至多经过 n - 1 条边
23     # 因此如果经过了多于 n 条边，一定说明经过了负环
```



```
24         if vis[v] == True:
25             q.append(v)
26             vis[v] = True
```

虽然在大多数情况下 SPFA 跑得很快，但其最坏情况下的时间复杂度为  $O(nm)$ ，将其卡到这个复杂度也是不难的，所以考试时要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman-Ford 算法无法通过的数据范围）。

#### Bellman-Ford 的其他优化

除了队列优化（SPFA）之外，Bellman-Ford 还有其他形式的优化，这些优化在部分图上效果明显，但在某些特殊图上，最坏复杂度可能达到指数级。

- 堆优化：将队列换成堆，与 Dijkstra 的区别是允许一个点多次入队。在有负权边的图可能被卡成指数级复杂度。
- 栈优化：将队列换成栈（即将原来的 BFS 过程变成 DFS），在寻找负环时可能具有更高效率，但最坏时间复杂度仍然为指数级。
- LLL 优化：将普通队列换成双端队列，每次将入队结点距离和队内距离平均值比较，如果更大则插入至队尾，否则插入队首。
- SLF 优化：将普通队列换成双端队列，每次将入队结点距离和队首比较，如果更大则插入至队尾，否则插入队首。
- D'Esopo-Pape 算法：将普通队列换成双端队列，如果一个节点之前没有入队，则将其插入队尾，否则插入队首。

更多优化以及针对这些优化的 Hack 方法，可以看 [fstqwq 在知乎上的回答](https://www.zhihu.com/question/292283275/answer/484871888) [https://www.zhihu.com/question/292283275/answer/484871888]。



## Dijkstra 算法

Dijkstra (/ˈdɪkstrə/或/ˈdeɪkstrə/) 算法由荷兰计算机科学家 E. W. Dijkstra 于 1956 年发现，1959 年公开发表。是一种求解 **非负权图** 上单源最短路径的算法。

## 流程

将结点分成两个集合：已确定最短路长度的点集（记为  $S$  集合）的和未确定最短路长度的点集（记为  $T$  集合）。一开始所有的点都属于  $T$  集合。

初始化  $dis(s) = 0$ ，其他点的  $dis$  均为  $+\infty$ 。

然后重复这些操作：

1. 从  $T$  集合中，选取一个最短路长度最小的结点，移到  $S$  集合中。
2. 对那些刚刚被加入  $S$  集合的结点的所有出边执行松弛操作。

直到  $T$  集合为空，算法结束。

## 时间复杂度

有多种方法来维护 1 操作中最短路长度最小的结点，不同的实现导致了 Dijkstra 算法时间复杂度上的差异。

- 暴力：不使用任何数据结构进行维护，每次 2 操作执行完毕后，直接在  $T$  集合中暴力寻找最短路长度最小的结点。2 操作总时间复杂度为  $O(m)$ ，1 操作总时间复杂度为  $O(n^2)$ ，全过程的时间复杂度为  $O(n^2 + m) = O(n^2)$ 。
- 二叉堆：每成功松弛一条边  $(u, v)$ ，就将  $v$  插入二叉堆中（如果  $v$  已经在二叉堆中，直接修改相应元素的权值即可），1 操作直接取堆顶结点即可。共计  $O(m)$  次二叉堆上的插入（修改）操作， $O(n)$  次删除堆顶操作，而插入（修改）和删除的时间复杂度均为  $O(\log n)$ ，时间复杂度为  $O((n + m) \log n) = O(m \log n)$ 。

- 优先队列：和二叉堆类似，但使用优先队列时，如果同一个点的最短路被更新多次，因为先前更新时插入的元素不能被删除，也不能被修改，只能留在优先队列中，故优先队列内的元素个数是  $O(m)$  的，时间复杂度为  $O(m \log m)$ 。
- Fibonacci 堆：和前面二者类似，但 Fibonacci 堆插入的时间复杂度为  $O(1)$ ，故时间复杂度为  $O(n \log n + m) = O(n \log n)$ ，时间复杂度最优。但因为 Fibonacci 堆较二叉堆不易实现，效率优势也不够大<sup>[1]</sup>，算法竞赛中较少使用。
- 线段树：和二叉堆原理类似，不过将每次成功松弛后插入二叉堆的操作改为在线段树上执行单点修改，而 1 操作则是线段树上的全局查询最小值。时间复杂度为  $O(m \log n)$ 。

在稀疏图中， $m = O(n)$ ，使用二叉堆实现的 Dijkstra 算法较 Bellman-Ford 算法具有较大的效率优势；而在稠密图中， $m = O(n^2)$ ，这时候使用暴力做法较二叉堆实现更优。

## 正确性证明

下面用数学归纳法证明，在 **所有边权值非负** 的前提下，Dijkstra 算法的正确性<sup>[2]</sup>。

简单来说，我们要证明的，就是在执行 1 操作时，取出的结点  $u$  最短路均已经被确定，即满足  $D(u) = \text{dis}(u)$ 。

初始时  $S = \emptyset$ ，假设成立。

接下来用反证法。



设  $u$  点为算法中第一个在加入  $S$  集合时不满足  $D(u) = \text{dis}(u)$  的点。因为  $s$  点一定满足  $D(u) = \text{dis}(u) = 0$ ，且它一定是第一个加入  $S$  集合的点，因此将  $u$  加入  $S$  集合前， $S \neq \emptyset$ ，如果不存在  $s$  到  $u$  的路径，则  $D(u) = \text{dis}(u) = +\infty$ ，与假设矛盾。

于是存在路径  $s \rightarrow x \rightarrow y \rightarrow u$ ，其中  $y$  为  $s \rightarrow u$  路径上第一个属于  $T$  集合的点，而  $x$  为  $y$  的前驱结点（显然  $x \in S$ ）。需要注意的是，可能存在  $s = x$  或  $y = u$  的情况，即  $s \rightarrow x$  或  $y \rightarrow u$  可能是空路径。

因为在  $u$  结点之前加入的结点都满足  $D(u) = dis(u)$ ，所以在  $x$  点加入到  $S$  集合时，有  $D(x) = dis(x)$ ，此时边  $(x, y)$  会被松弛，从而可以证明，将  $u$  加入到  $S$  时，一定有  $D(y) = dis(y)$ 。

下面证明  $D(u) = dis(u)$  成立。在路径  $s \rightarrow x \rightarrow y \rightarrow u$  中，因为图上所有边边权非负，因此  $D(y) \leq D(u)$ 。从而  $dis(y) \leq D(y) \leq D(u) \leq dis(u)$ 。但是因为  $u$  结点在 1 过程中被取出  $T$  集合时， $y$  结点还没有被取出  $T$  集合，因此此时有  $dis(u) \leq dis(y)$ ，从而得到  $dis(y) = D(y) = D(u) = dis(u)$ ，这与  $D(u) \neq dis(u)$  的假设矛盾，故假设不成立。

因此我们证明了，1 操作每次取出的点，其最短路均已经被确定。命题得证。

注意到证明过程中的关键不等式  $D(y) \leq D(u)$  是在图上所有边边权非负的情况下得出的。当图上存在负权边时，这一不等式不再成立，Dijkstra 算法的正确性将无法得到保证，算法可能会给出错误的结果。

## 代码实现

这里同时给出  $O(n^2)$  的暴力做法实现和  $O(m \log m)$  的优先队列做法实现。

### 暴力实现

```
1 // C++ Version
2 struct edge {
3     int v, w;
4 };
5 vector<edge> e[maxn];
```

```
6  int dis[maxn], vis[maxn];
7  void dijkstra(int n, int s) {
8      memset(dis, 63, sizeof(dis));
9      dis[s] = 0;
10     for (int i = 1; i <= n; i++) {
11         int u = 0, mind = 0x3f3f3f3f;
12         for (int j = 1; j <= n; j++)
13             if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
14         vis[u] = true;
15         for (auto ed : e[u]) {
16             int v = ed.v, w = ed.w;
17             if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
18         }
19     }
20 }
```

```
1  # Python Version
2  class Edge:
3      v = 0
4      w = 0
5  e = [[Edge() for i in range(maxn)] for j in range(maxn)]
6  dis = [63] * maxn; vis = [] * maxn
7  def dijkstra(n, s):
8      dis[s] = 0
9      for i in range(1, n + 1):
10         u = 0; mind = 0x3f3f3f3f
11         for j in range(1, n + 1):
12             if vis[j] == False and dis[j] < mind:
13                 u = j; mind = dis[j]
14         vis[u] = True
```





```
15         for ed in e[u]:
16             v = ed.v; w = ed.w
17             if dis[v] > dis[u] + w:
18                 dis[v] = dis[u] + w
```

### 优先队列实现

```
1  struct edge {
2      int v, w;
3  };
4  struct node {
5      int dis, u;
6      bool operator>(const node& a) const { return dis > a.dis; }
7  };
8  vector<edge> e[maxn];
9  int dis[maxn], vis[maxn];
10 priority_queue<node, vector<node>, greater<node> > q;
11 void dijkstra(int n, int s) {
12     memset(dis, 63, sizeof(dis));
13     dis[s] = 0;
14     q.push({0, s});
15     while (!q.empty()) {
16         int u = q.top().u;
17         q.pop();
18         if (vis[u]) continue;
19         vis[u] = 1;
20         for (auto ed : e[u]) {
21             int v = ed.v, w = ed.w;
22             if (dis[v] > dis[u] + w) {
```

```
23         dis[v] = dis[u] + w;  
24         q.push({dis[v], v});  
25     }  
26 }  
27 }  
28 }
```

## Johnson 全源最短路径算法

Johnson 和 Floyd 一样，是一种能求出无负环图上任意两点间最短路径的算法。该算法在 1977 年由 Donald B. Johnson 提出。

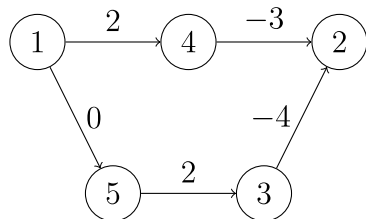
任意两点间的最短路可以通过枚举起点，跑  $n$  次 Bellman-Ford 算法解决，时间复杂度是  $O(n^2m)$  的，也可以直接用 Floyd 算法解决，时间复杂度为  $O(n^3)$ 。

注意到堆优化的 Dijkstra 算法求单源最短路径的时间复杂度比 Bellman-Ford 更优，如果枚举起点，跑  $n$  次 Dijkstra 算法，就可以在  $O(nm \log m)$ （取决于 Dijkstra 算法的实现）的时间复杂度内解决本问题，比上述跑  $n$  次 Bellman-Ford 算法的时间复杂度更优秀，在稀疏图上也比 Floyd 算法的时间复杂度更加优秀。

但 Dijkstra 算法不能正确求解带负权边的最短路，因此我们需要对原图上的边进行预处理，确保所有边的边权均非负。

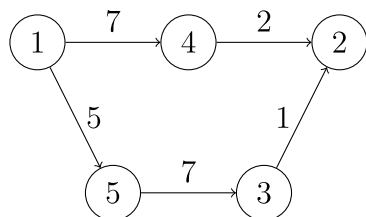
一种容易想到的方法是给所有边的边权同时加上一个正数  $x$ ，从而让所有边的边权均非负。如果新图上起点到终点的最短路经过了  $k$  条边，则将最短路减去  $kx$  即可得到实际最短路。

但这样的方法是错误的。考虑下图：



$1 \rightarrow 2$  的最短路为  $1 \rightarrow 5 \rightarrow 3 \rightarrow 2$ ，长度为  $-2$ 。

但假如我们把每条边的边权加上  $5$  呢？



新图上  $1 \rightarrow 2$  的最短路为  $1 \rightarrow 4 \rightarrow 2$ ，已经不是实际的最短路了。

Johnson 算法则通过另外一种方法来给每条边重新标注边权。

我们新建一个虚拟节点（在这里我们就设它的编号为  $0$ ）。从这个点向其他所有点连一条边权为  $0$  的边。

接下来用 Bellman-Ford 算法求出从  $0$  号点到其他所有点的最短路，记为  $h_i$ 。

假如存在一条从  $u$  点到  $v$  点，边权为  $w$  的边，则我们将该边的边权重新设置为  $w + h_u - h_v$ 。

接下来以每个点为起点，跑  $n$  轮 Dijkstra 算法即可求出任意两点间的最短路了。



一开始的 Bellman-Ford 算法并不是时间上的瓶颈，若使用 `priority_queue` 实现 Dijkstra 算法，该算法的时间复杂度是  $O(nm \log m)$ 。

## 正确性证明

为什么这样重新标注边权的方式是正确的呢？

在讨论这个问题之前，我们先讨论一个物理概念——势能。

诸如重力势能，电势能这样的势能都有一个特点，势能的变化量只和起点和终点的相对位置有关，而与起点到终点所走的路径无关。

势能还有一个特点，势能的绝对值往往取决于设置的零势能点，但无论将零势能点设置在哪里，两点间势能的差值是一定的。

接下来回到正题。

在重新标记后的图上，从  $s$  点到  $t$  点的一条路径  $s \rightarrow p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_k \rightarrow t$  的长度表达式如下：

$$(w(s, p_1) + h_s - h_{p_1}) + (w(p_1, p_2) + h_{p_1} - h_{p_2}) + \cdots + (w(p_k, t) + h_{p_k} - h_t)$$

化简后得到：

$$w(s, p_1) + w(p_1, p_2) + \cdots + w(p_k, t) + h_s - h_t$$

无论我们从  $s$  到  $t$  走的是哪一条路径， $h_s - h_t$  的值是不变的，这正与势能的性质相吻合！

为了方便，下面我们就把  $h_i$  称为  $i$  点的势能。



上面的新图中  $s \rightarrow t$  的最短路的长度表达式由两部分组成，前面的边权和为原图中  $s \rightarrow t$  的最短路，后面则是两点间的势能差。因为两点间势能的差为定值，因此原图上  $s \rightarrow t$  的最短路与新图上  $s \rightarrow t$  的最短路相对应。

到这里我们的正确性证明已经解决了一半——我们证明了重新标注边权后图上的最短路径仍然是原来的最短路径。接下来我们需要证明新图中所有边的边权非负，因为在非负权图上，Dijkstra 算法能够保证得出正确的结果。

根据三角形不等式，图上任意一边  $(u, v)$  上两点满足： $h_v \leq h_u + w(u, v)$ 。这条边重新标记后的边权为  $w'(u, v) = w(u, v) + h_u - h_v \geq 0$ 。这样我们证明了新图上的边权均非负。

这样，我们就证明了 Johnson 算法的正确性。

---

## 不同方法的比较



最短路算法	Floyd	Bellman-Ford	Dijkstra	Johnson
最短路类型	每对结点之间的最短路	单源最短路	单源最短路	每对结点之间的最短路
作用于	任意图	任意图	非负权图	任意图
能否检测负环?	能	能	不能	能
推荐作用图的大小	小	中/小	大/中	大/中
时间复杂度	$O(N^3)$	$O(NM)$	$O(M \log M)$	$O(NM \log M)$

注：表中的 Dijkstra 算法在计算复杂度时均用 `priority_queue` 实现。

## 输出方案

开一个 `pre` 数组，在更新距离的时候记录下来后面的点是如何转移过去的，算法结束前再递归地输出路径即可。

比如 Floyd 就要记录 `pre[i][j] = k`；，Bellman-Ford 和 Dijkstra 一般记录 `pre[v] = u`。



## 参考资料与注释

1. [Worst case of fibonacci heap - Wikipedia](https://en.wikipedia.org/wiki/Fibonacci_heap#Worst_case) [https://en.wikipedia.org/wiki/Fibonacci\_heap#Worst\_case]
2. 《算法导论（第3版中译本）》，机械工业出版社，2013年，第384 - 385页。

🔑 本页面最近更新：2021/9/27 10:32:11, [更新历史](https://github.com/OI-wiki/OI-wiki/commits/master/docs/graph/shortest-path.md) [https://github.com/OI-wiki/OI-wiki/commits/master/docs/graph/shortest-path.md]


✎ 发现错误？想一起完善？在 [GitHub](https://oi-wiki.org/edit-landing/?ref=/graph/shortest-path.md) 上编辑此页！ [https://oi-wiki.org/edit-landing/?ref=/graph/shortest-path.md]

👤 本页面贡献者：[PeterlitsZo](https://github.com/PeterlitsZo) [https://github.com/PeterlitsZo], [zyj-111](https://github.com/zyj-111) [https://github.com/zyj-111], [Enter-tainer](https://github.com/Enter-tainer) [https://github.com/Enter-tainer], [StudyingFather](https://github.com/StudyingFather) [https://github.com/StudyingFather], [H-Shen](https://github.com/H-Shen) [https://github.com/H-Shen], [iamtwz](https://github.com/iamtwz) [https://github.com/iamtwz], [Linhk1606](https://github.com/Linhk1606) [https://github.com/Linhk1606], [shenyouran](https://github.com/shenyouran) [https://github.com/shenyouran], [mcendu](https://github.com/mcendu) [https://github.com/mcendu], [renbaoshuo](https://github.com/renbaoshuo) [https://github.com/renbaoshuo], [isdanni](https://github.com/isdanni) [https://github.com/isdanni], [Xeonacid](https://github.com/Xeonacid) [https://github.com/Xeonacid]

© 本页面的全部内容按 [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/deed.zh) [https://creativecommons.org/licenses/by-sa/4.0/deed.zh] 和 [SATA](https://github.com/zTrix/sata-license) [https://github.com/zTrix/sata-license] 协议之条款下提供，附加条款亦可能应用

## 评论

[22](https://github.com/OI-wiki/gitment/issues/45) [https://github.com/OI-wiki/gitment/issues/45] comments

Anonymous 



[]

Leave a comment