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 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

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。

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 $n - 1$。

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（只有三个 `for`）。

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 $V'=1,2,\ldots,k$ 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 $V'=1,2,\ldots,n$ 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。

C++

```cpp
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
    }
  }
}
```

Python

```python
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            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

对于给定的 `k`，当更新 `f[k][x][y]` 时，涉及的元素总是来自 `f[k-1]` 数组的第 `k` 行和第 `k` 列。然后我们可以发现，对于给定的 `k`，当更新 `f[k][k][y]` 或 `f[k][x][k]`，总是不会发生数值更新，因为按照公式 `f[k][k][y] = min(f[k-1][k][y], f[k-1][k][k]+f[k-1][k][y])`，`f[k-1][k][k]` 为 0，因此这个值总是 `f[k-1][k][y]`，对于 `f[k][x][k]` 的证明类似。

因此，如果省略第一维，在给定的 `k` 下，每个元素的更新中使用到的元素都没有在这次迭代中更新，因此第一维的省略并不会影响结果。

C++

```cpp
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
    }
  }
}
```

Python

```python
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            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 $O(n^3)$。

更多参见 最小环 部分内容。

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 $O(\frac{n^3}{w})$。

```
// std::bitset<SIZE> f[SIZE];
for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
    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。

过程

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 $S \to u \to v$（其中 $S \to u$ 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。

实现

参考实现

C++

```cpp
struct Edge {
  int u, v, w;
};

vector<Edge> edge;

int dis[MAXN], u, v, w;
constexpr int INF = 0x3f3f3f3f;

bool bellmanford(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
  bool flag = false;  // 判断一轮循环过程中是否发生松弛操作
  for (int i = 1; i <= n; i++) {
    flag = false;
    for (int j = 0; j < edge.size(); j++) {
      u = edge[j].u, v = edge[j].v, w = edge[j].w;
      if (dis[u] == INF) continue;
      // 无穷大与常数加减仍然为无穷大
      // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        flag = true;
      }
    }
    // 没有可以松弛的边时就停止算法
    if (!flag) {
      break;
    }
  }
  // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
  return flag;
}
```

Python

```python
class Edge:
    def __init__(self, u=0, v=0, w=0):
        self.u = u
        self.v = v
        self.w = w


INF = 0x3F3F3F3F
edge = []

```

```python
def bellmanford(n, s):
    dis = [INF] * (n + 1)
    dis[s] = 0
    for i in range(1, n + 1):
        flag = False
        for e in edge:
            u, v, w = e.u, e.v, e.w
            if dis[u] == INF:
                continue
            # 无穷大与常数加减仍然为无穷大
            # 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                flag = True
        # 没有可以松弛的边时就停止算法
        if not flag:
            break
    # 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
    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。

实现

C++

```cpp
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], cnt[MAXN], vis[MAXN];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= n) return false;
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return true;
}
```

Python

```python
from collections import deque


class Edge:
    def __init__(self, v=0, w=0):
        self.v = v
        self.w = w


e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
INF = 0x3F3F3F3F


def spfa(n, s):
    dis = [INF] * (n + 1)
```

```python
    cnt = [0] * (n + 1)
    vis = [False] * (n + 1)
    q = deque()

    dis[s] = 0
    vis[s] = True
    q.append(s)
    while q:
        u = q.popleft()
        vis[u] = False
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                cnt[v] = cnt[u] + 1  # 记录最短路经过的边数
                if cnt[v] >= n:
                    return False
                # 在不经过负环的情况下，最短路至多经过 n - 1 条边
                # 因此如果经过了多于 n 条边，一定说明经过了负环
                if not vis[v]:
                    q.append(v)
                    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。

Dijkstra 算法

Dijkstra（/'dikstrɑ/或/'dɛikstrɑ/）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。

过程

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

朴素的实现方法为每次 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 $O(n^2+m)=O(n^2)$。

可以用堆来优化这一过程：每成功松弛一条边 (u,v)，就将 v 插入堆中（如果 v 已经在堆中，直接执行 Decrease-key），1 操作直接取堆顶结点即可。共计 $O(m)$ 次 Decrease-key，$O(n)$ 次 pop，选择不同堆可以取到不同的复杂度，参考 堆 页面。堆优化能做到的最优复杂度为 $O(n\log n+m)$，能做到这一复杂度的有斐波那契堆等。

特别地，可以使用优先队列维护，此时无法执行 Decrease-key 操作，但可以通过每次松弛时重新插入该结点，且弹出时检查该结点是否已被松弛过，若是则跳过，复杂度 $O(m\log n)$，优点是实现较简单。

这里的堆也可以用线段树来实现，复杂度为 $O(m\log n)$，在一些特殊的非递归线段树实现下，该做法常数比堆更小。并且线段树支持的操作更多，在一些特殊图问题上只能用线段树来维护。

在稀疏图中，$m=O(n)$，堆优化的 Dijkstra 算法具有较大的效率优势；而在稠密图中，$m=O(n^2)$，这时候使用朴素实现更优。

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[1]。

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 $D(u) = dis(u)$。

初始时 $S = \varnothing$，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 $D(u) = dis(u)$ 的点。因为 s 点一定满足 $D(u) = dis(u) = 0$，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，$S \neq \varnothing$，如果不存在 s 到 u 的路径，则 $D(u) = dis(u) = +\infty$，与假设矛盾。

于是一定存在路径 $s \to x \to y \to u$，其中 y 为 $s \to u$ 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 $s \to x$ 或 $y \to u$ 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 $s \to x \to y \to u$ 中，因为图上所有边边权非负，因此 $D(y) \leq D(u)$。从而 $dis(y) = D(y) \leq D(u) \leq dis(u)$。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。

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。

朴素实现

C++

```cpp
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];

void dijkstra(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
  for (int i = 1; i <= n; i++) {
    int u = 0, mind = 0x3f3f3f3f;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
    vis[u] = true;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
    }
  }
}
```

Python

```python
class Edge:
    def __init(self, v=0, w=0):
        self.v = v
        self.w = w


e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
INF = 0x3F3F3F3F


def dijkstra(n, s):
    dis = [INF] * (n + 1)
    vis = [0] * (n + 1)

    dis[s] = 0
    for i in range(1, n + 1):
        u = 0
        mind = INF
        for j in range(1, n + 1):
            if not vis[j] and dis[j] < mind:
                u = j
                mind = dis[j]
        vis[u] = True
```

```
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
```

优先队列实现

C++

```cpp
struct edge {
  int v, w;
};

struct node {
  int dis, u;

  bool operator>(const node& a) const { return dis > a.dis; }
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];
priority_queue<node, vector<node>, greater<node>> q;

void dijkstra(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  memset(vis, 0, (n + 1) * sizeof(int));
  dis[s] = 0;
  q.push({0, s});
  while (!q.empty()) {
    int u = q.top().u;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.push({dis[v], v});
      }
    }
  }
}
```

Python

```python
def dijkstra(e, s):
    """
    输入：
    e:邻接表
    s:起点
    返回：
    dis:从s到每个顶点的最短路长度
    """
    dis = defaultdict(lambda: float("inf"))
    dis[s] = 0
    q = [(0, s)]
```

```
    vis = set()
    while q:
        _, u = heapq.heappop(q)
        if u in vis:
            continue
        vis.add(u)
        for v, w in e[u]:
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                heapq.heappush(q, (dis[v], v))
    return dis
```

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。

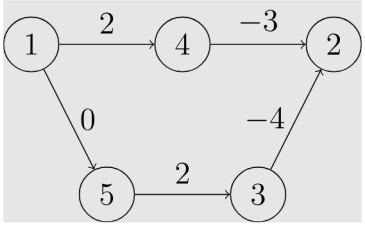
任意两点间的最短路可以通过枚举起点，跑 n 次 Bellman–Ford 算法解决，时间复杂度是 $O(n^2m)$ 的，也可以直接用 Floyd 算法解决，时间复杂度为 $O(n^3)$。

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。

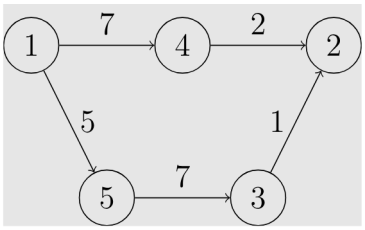
一种容易想到的方法是给所有边的边权同时加上一个正数 x，从而让所有边的边权均非负。如果新图上起点到终点的最短路经过了 k 条边，则将最短路减去 kx 即可得到实际最短路。

但这样的方法是错误的。考虑下图：



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

但假如我们把每条边的边权加上 5 呢？



新图上 $1 \to 2$ 的最短路为 $1 \to 4 \to 2$，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 $s \to p_1 \to p_2 \to \cdots \to p_k \to t$ 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 $s \to t$ 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 $s \to t$ 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 $s \to t$ 的最短路与新图上 $s \to t$ 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
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`。

一些特殊情形

- 边权只由 0 和 1 组成的图上最短路：0-1 BFS；
- 允许至多 k 次改变路径成本等操作的最短路问题：分层图最短路。

参考资料与注释

1. 《算法导论（第 3 版中译本）》，机械工业出版社，2013 年，第 384 - 385 页。 ↩