# 最短路算法

## Introduction



**图论最短路算法核心比较表**

| 特性 | Dijkstra (优先队列优化) | Bellman-Ford | SPFA (队列优化BF) |
|---|---|---|---|
| **解决的问题类型** | **单源最短路 (SSSP)** | **单源最短路 (SSSP)** | **单源最短路 (SSSP)** |
| **适用的图类型** | **非负权** 有向图/无向图 | **任意权** 有向图/无向图 | **任意权** 有向图/无向图 |
| **能否处理负权边?** | ❌ 不能 | ✅ 能 | ✅ 能 |
| **能否检测负权环?** | ❌ 不能 | ✅ 能 (源点可达的环) | ✅ 能 (源点可达的环) |
| **时间复杂度** | $O((V+E)\log V)$ | $O(V \times E)$ | **最坏** $O(V \times E)$，**平均** $O(kE)$ **(k为常数，通常很** |
| **空间复杂度** | $O(V)$或$O(V+E)$ | $O(V)$ | $O(V)$ |
| **核心思想** | 贪心 + BFS，优先队列管理待处理顶点 | 动态规划，$V-1$ 轮松弛所有边 | BF 优化，队列管理松弛成功的 |
| **主要优点** | 非负权图中效率最高 | 能处理负权，可检测负环，实现简单 | 平均性能远优于 BF，处理负权高效 |
| **主要缺点/限制** | 无法处理负权边 | 时间复杂度高，效率较低 | 最坏时间复杂度与 BF 相同，稳定性不如 Dijkstra |

1. SSSP vs APSP:
   - SSSP (Single-Source Shortest Path): 求解从**一个固定起点**到图中所有其他顶点的最短路径。Dijkstra, Bellman-Ford, SPFA 属于此类。
   - APSP (All-Pairs Shortest Path): 求解图中**任意两个顶点之间**的最短路径。Floyd-Warshall 专门解决此问题。也可通过运行 V 次 SSSP 算法解决（如 V 次 Dijkstra 或 V 次 SPFA），但效率取决于图规模和算法选择。
2. **负权边 vs 负权环:**
   - **负权边:** 边权值为负数。Dijkstra **无法处理**，会出错。其他三种算法可以处理。
   - **负权环 (Negative-Weight Cycle):** 图中存在一个环，其总边权之和为负。
     - 在负权环**可达**的路径上，不存在有限的最短路径（可以无限绕环使总权值趋近负无穷）。
     - Bellman-Ford 和 SPFA 在求解 SSSP 时可以 **检测源点可达的负权环**。
     - Floyd-Warshall 在求解 APSP 时可以 **检测图中是否存在任意负权环**（检查 `dist[i][i] < 0`）。
     - 如果图有负权边但 **没有负权环**，则所有最短路径都是有限且良定义的，BF、SPFA、Floyd 能正确求解。
3. **效率与选择指南 (简化版):**
   - SSSP + **所有边权 >= 0:** **优先队列优化 Dijkstra** (最快)。

- SSSP + 含负权边/需检测负环：
  - 不确定是否有负环/需要检测： Bellman-Ford 或 SPFA (SPFA 通常更快)。
  - **确定无负环：**SPFA (平均性能好)。
- APSP (任意两点间)：
  - 图规模小 (V 小)： Floyd-Warshall (简单直接)。
  - 图规模大 + 所有边权 >= 0： 运行 V 次 Dijkstra。
  - 图规模大 + 含负权边但无负环： 运行 V 次 SPFA (需注意最坏时间复杂度风险)。
  - 图规模大 + 含负权边且需检测负环： 考虑 Johnson's Algorithm (结合 BF 和 Dijkstra) 或谨慎使用 Floyd (仅当 V 足够小)。

# Floyd

Floyd 算法基于传递闭包所设计。

> 所谓传递闭包：
> 传递闭包是集合 $X$ 上**二元关系** $R$ 的**最小传递关系**，其定义为包含R的所有传递关系的交集。例如在家族关系中，传递闭包可表达祖先关系，在航空线路中表示经多次航行可达的路径。**该概念在数学中具有严格的构造性定义，可通过有限元素序列实现路径连接。**
>
> 任何关系R的传递闭包必定存在，通过包含R的传递关系交集构造。当两个传递关系取并集时，需通过**传递闭包保持传递性**，这在等价关系合并时尤为重要。该概念在图论中体现为**有向无环图的可达性关系**，并与计算复杂性理论中的NL类问题相关联。
>
> ——摘自百度百科

我们只用知道它定义一种传递关系即可。这里的传递关系通常指**有向无权图中的可达性关系**。

所谓传递即满足如下图所示的关系：

graph A <--> B B <--> C A <--> I可由左侧关系推出IC
**Problem**可以帮助你更好的理解它的实际意义。

*Code*

既然我们已经可以判定两点之间是否可达，我们尝试更近一步推出两点之间的最短路。

## 核心思想

Floyd-Warshall 是解决**全源最短路径 (APSP)** 问题的经典算法。

基于**动态规划**思想，通过逐步扩展中间点集来更新最短路径。

所谓**松弛**：给定三个顶点 $i, j, k$，总有 $dis_{i,j} = \min(dis_{i,j}, dis_{i,k} + dis_{k,j})$。正确性显然。

初始时，直接连接的两点距离为边权，不连接的点对距离为 $+\infty$，对角线元素为 $0$。

**算法核心：三重循环**

- **第一层循环** $k = 1$ 到 $n$：考虑经过中间点 $k$ 的路径。
- **第二层循环** $i = 1$ 到 $n$：枚举起点。
- **第三层循环** $j = 1$ 到 $n$：枚举终点。

**负权环检测**

在 Floyd 算法完成后，若存在 $dis[i][i] < 0$，则图中存在负权环（经过点 $i$ 的环权之和为负）。

Problem：P1119 灾后重建

Code：Code

# SPFA（BellmanFord + 队列优化）

## 前置算法 —— BellmanFord

基于**松弛**操作检测负环并实现最短路。

每轮**松弛**都要遍历所有边。

所谓**松弛**：给定一条边 $(u, v, w)$，总有 $dis_v = min(dis_v, dis_u + w)$。正确性显然。

初始时源点 $dis_{root}$ 为 $0$，其它点 $dis_i = inf$。

Code

## SPFA

**前言**

SPFA（Shortest Path Faster Algorithem），是基于 BellmanFord 的队列优化算法，但是……

> 关于 SPFA
>
> 它死了……

**不要在能使用 Dijkstra 的时候使用 SPFA 或 BellmanFord。**

---

> 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。

由于只有上一次对 $dis_i$ 产生更新的点才会对下一次更新产生贡献，于是想到利用类似 BFS 的方法（使用队列）对 BellmanFord 进行优化。

```
int n,m,u,v,w,dis[maxn],cnt[maxn];
bool inq[maxn];
vector<pii> g[maxn];
queue<int>q;
void in(int x){inq[x] = true,q.push(x),cnt[x]++;};
int out(){int x = q.front();q.pop();return inq[x] = false,x;};
void spfa(int s){
    in(s);
    dis[s] = 0;
```

```
    while(!q.empty()){
        u = out();
        for(auto [v,w]:g[u]){
            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                if(!inq[v]){
                    in(v);
                    if(cnt[v] == n+1){
                        cout<<"NO SOLUTION";
                        exit(0);
                    }
                }
            }
        }
    }
}
```

*Complete Code*

Problem：P3371 【模板】单源最短路径（弱化版）

# Dijkstra

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（最短的边松弛出最短路的潜力最大）。
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。
3. 直到 T 集合为空，算法结束。

Dijkstra 遵循每次选择一条集合中距离最小的点进行松弛操作。这也意味着 **Dijkstra无法处理负权边**

## 暴力做法

**时间复杂度 $O(V^2)$**

```
namespace Graph {
    struct Edge {
        int v,w;
    };
    constexpr int maxn = 1e4+5;
    int dis[maxn],u,v,w,mind,cnt[maxn];
    vector<Edge> g[maxn];
    bool vis[maxn];
    void add() {
        cin>>u>>v>>w;
        g[u].push_back({v,w});
        return;
```

```
    }
    void Dijkstra(int n,int s){
        memset(dis,0x3f,sizeof(dis));
        dis[s] = 0;
        for(int i=1;i<=n;i++){
            // 1. 找到未访问顶点中距离最小的顶点
            mind = INT_MAX,u = -1;
            for(int j=1;j<=n;j++){
                if(!vis[j] && dis[j]<mind){
                    mind = dis[j];
                    u = j;
                }
            }
            // 如果找不到可达的未访问顶点，结束循环
            if(u == -1)break;
            // 标记该顶点已访问
            vis[u] = true;
            // 2. 松弛操作：更新所有邻接顶点的距离
            for(Edge e:g[u]){
                v = e.v,w = e.w;
                if(!vis[i]){//只考虑未访问的节点
                    dis[v] = min(dis[v],dis[u]+w);
                }
            }
        }
    }
    void print(int n){
        for(int i=1;i<=n;i++)cout<<dis[i]<<' ';
        cout<<endl;
        return;
    }
}
```

## 优先队列优化

**时间复杂度** $O((V+E)\log V)$

每次取出最短路最小的节点，避免枚举查找。

*Code*

注意这里使用了vis数组进行判断，实际上我们可以不使用vis数组，直接判断**存储的**最短路是否和**现存的**最短路相等（判断最短路信息是否已经**过期**）。

这种思想十分重要，运用 Dijkstra 求**次短路**正是基于此。

Problem：P4779 【模板】单源最短路径（标准版）

# Johnson全源最短路

**时间复杂度** $O(V\times E\log E)$

# 核心思想

Johnson 算法的精妙之处在于它**巧妙地结合了 Bellman-Ford 算法和 Dijkstra 算法**的优势：

1. **处理负权边：** 利用 Bellman-Ford 算法的能力检测负权环并处理负权边。
2. **高效求解 APSP：** 利用 Dijkstra 算法在非负权重图上的效率，避免了对所有节点对重复运行 Bellman-Ford（$O(V^2E)$）或使用 Floyd-Warshall（$O(V^3)$）在稀疏图上的相对低效。
3. **重赋权（Reweighting）：算法的核心步骤是通过一个精心设计的权重转换函数**，修改原图 $G$ 中所有边的权重，生成一个新图 $G'$。这个转换的关键在于：
   - **保持最短路径：** 图 $G$ 中的最短路径在图 $G'$ 中仍然是相同节点序列的最短路径（反之亦然）。
   - **消除负权重：图 $G'$ 中所有边的权重都变为非负值** $(w' \geq 0)$。

一旦图 $G'$ 的所有边权变为非负，我们就可以对图 $G'$ 中的**每一个节点**安全地运行 **Dijkstra 算法**（单源最短路径算法，时间复杂度 $O(E \log V)$）来求解所有节点对的最短路径。最后，再将 $G'$ 中得到的最短路径距离**转换回**原图 $G$ 的距离。

# 算法步骤详解

记 a，b 之间的最短距离为 $\delta(a,b)$。

1. **添加新节点：**
   - 创建一个新的节点 $s$，该节点**不属于**原图 $G(V,E)$。
   - 从 $s$ 向原图 $G$ 中的**每一个节点** $v \in V$ 添加一条**权重为 0** 的有向边。
   - 这样得到一个新的图 $G''=(V'',E'')$，其中 $V''=V\cup\{s\}$，$E''=E\cup\{(s,v)\mid v\in V,\text{weight}=0\}$。
2. **运行 Bellman-Ford 算法：**
   - 以新添加的节点 $s$ 作为源点，在扩展图 $G''$ 上运行 **Bellman-Ford 算法**。
   - **目的 1：检测负权环。**
     - 如果 Bellman-Ford 算法检测到 $G''$ 中存在从 $s$ 可达的负权环，则意味着**原图 $G$ 中也存在负权环**（因为新加的边权重为 0，环必然在原图中）。
     - **算法终止：** 如果存在负权环，则无法定义最短路径（因为可以无限次绕环降低路径长度），算法报告错误并结束。
   - **目的 2：计算势函数 $h(v)$。**
     - 如果不存在负权环，Bellman-Ford 算法会成功计算出从源点 $s$ 到原图 $G$ 中每个节点 $v \in V$ 的最短路径距离。
     - 我们将这个距离记作 $h(v)$，即 $h(v)=\delta(s,v)$（在 $G''$ 中）。
     - **$h(v)$ 被称为节点 $v$ 的“势能”或“高度”。**
3. **重赋权（计算新权重 $w'$）：**
   - 对于原图 $G$ 中的**每一条边** $(u,v)\in E$，计算其**新的权重** $w'(u,v)$：

$$w'(u,v)=w(u,v)+h(u)-h(v)$$

   - 这个转换是 Johnson 算法的**核心魔法**：
     - **消除负权：** $w'(u,v)\geq 0$。证明如下：
       根据三角不等式（由 Bellman-Ford 保证的最短路径性质），在 $G''$ 中：

$$\delta(s,v)\leq\delta(s,u)+w(u,v)\implies h(v)\leq h(u)+w(u,v)$$

       移项得：

$$w(u,v)+h(u)-h(v)\geq 0\implies w'(u,v)\geq 0$$

     - **保持最短路径：** 对于原图 $G$ 中的任意路径 $p=\langle v_0,v_1,\ldots,v_k\rangle$，设 $w(p)$ 和 $w'(p)$ 分别表示该路径在

$G$ 和 $G'$ 中的权重。则有：

$$w'(p) = w(p) + h(v_0) - h(v_k)$$

证明：

$$\begin{aligned} w'(p) &= \sum_{i=1}^{k} w'(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} [w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)] \\ &= \left[\sum_{i=1}^{k} w(v_{i-1}, v_i)\right] + [h(v_0) - h(v_k)] \quad \text{(裂项相消)} \\ &= w(p) + h(v_0) - h(v_k) \end{aligned}$$

- 这个等式表明：
  - 路径 $p$ 在 $G$ 和 $G'$ 中的权重**仅相差一个常数** $h(v_0) - h(v_k)$。
  - 这个常数**只与路径的起点 $v_0$ 和终点 $v_k$ 有关，与路径 $p$ 本身的具体节点序列无关**。
- **结论**：如果在原图 $G$ 中路径 $p$ 是从 $u$ 到 $v$ 的最短路径，那么在重赋权后的图 $G'$ 中，$p$ 也必然是从 $u$ 到 $v$ 的最短路径（因为所有从 $u$ 到 $v$ 的路径增加的权重都是相同的常数 $h(u) - h(v)$，所以权重最小的路径仍然是权重最小的路径）。反之亦然。

4. **在 $G'$ 上运行 Dijkstra**：
   - 现在，我们得到了一个新图 $G' = (V, E)$，其边权函数为 $w'$，且 $w'(u, v) \geq 0$。
   - 对于 $G'$ 中的**每一个节点** $u \in V$：
     - 以 $u$ 为源点，运行 **Dijkstra 算法**（因为所有边权非负）。
     - Dijkstra 算法会计算出从 $u$ 到 $G'$ 中所有其他节点 $v$ 的最短路径距离 $\delta'(u, v)$。
5. **还原原始距离**：
   - 对于每对节点 $(u, v)$，利用步骤 3 中推导出的关系 $w'(p) = w(p) + h(u) - h(v)$，将 $G'$ 中的最短路径距离 $\delta'(u, v)$ 还原为原图 $G$ 中的最短路径距离 $\delta(u, v)$：

$$\delta(u, v) = \delta'(u, v) - h(u) + h(v)$$

     - 这个公式直接来自 $w'(p) = w(p) + h(u) - h(v)$ 的推广。因为 $\delta'(u, v)$ 是 $G'$ 中从 $u$ 到 $v$ 的最小 $w'$ 值，$\delta(u, v)$ 是 $G$ 中从 $u$ 到 $v$ 的最小 $w$ 值，它们之间的关系就是 $\delta'(u, v) = \delta(u, v) + h(u) - h(v)$。移项即得还原公式。

## Code

Problem：P5905 【模板】全源最短路（Johnson）

# 次短路

研究次短路之前，我们要知道次短路的几种类别。

## 概念辨析

1. 有的次短路允许与最短路相等（有多条最短路），称为**非严格次短路**。
2. 有的次短路被要求不能等于最短路，称为**严格次短路**。
3. 有的次短路**允许重复**经过**同一道路或节点**，严格意义上称之为**次短途径（Walk）而不是路径（Path）**，但我们也可以

把他放到最短路研究的范畴。

4. 有的次短路**不允许重复**经过**同一道路或节点**，这是我们通常意义上所说的次短路。

| 场景/需求 | ExtDijkstra | 删边法 |
| --- | --- | --- |
| **简单路径** | ✗ | ✓ |
| **大规模数据** | ✓ | ✗ |
| **k短路扩展** | ✓ | ✗ |

## ExtDijkstra（途径）

在ExtDijkstra算法中，每次对边进行松弛操作时，需要同时检查最短路和次短路的更新可能性。特别要注意，由于次短路**允许包含环**（即同一节点可被多次访问），同一个节点的次短路状态可能被多次更新，这依赖于 ExtDijkstra 要求使用**距离是否有效**来判断能否更新节点。

> **实际上我们可以不使用vis数组，直接判断存储的最短路是否和现存的最短路相等（判断最短路信息是否已经过期）。**

代码实现中要特别关注是否为严格次短路。

Problem:P2865 [USACO06NOV] Roadblocks G - Code:link

## 删边法（路径）

次短路的边至少有一条不与最短路相同，因此可以每次禁用最短路上的一条边来强制 dijkstra 选择第二短的路。

Problem:P1491 集合位置 - Code:link

# Application

## 分层图-动态规划

一般的，我们使用 $dis$ 数组来记录每个节点的最短路。

我们可以借助动态规划的思想，来使用 $dis$ 数组记录下不同状态下到达某个节点的最短路。

每个状态对应一个**层**，这种具有多种状态或是动态变化的图我们称为分层图。

凭借这种思想可以通过许多与最短路有关的变种题目。

同样的，升维是一种很常用的 Trick。

```
//case 1
if(k_u > 0){
        k_v = f1(k_u);
        if(dis[v][k_v] > /*A*/){
                dis[v][k_v] = /*A*/;
                pq.push({dis[v][k_v],v,k_v});
        }
}
//case 2
```

```
k_v = f2(k_u);
if(dis[v][k_v] > /*B*/){
        dis[v][k_v] = /*B*/;
        pq.push({dis[v][k_v],v,k_v});
}
```

用它替换掉 `Dijkstra` 里面的松弛操作即可（依照题意修改 `f1` `f2` `A` `B` 的内容即可）

## 求解不等式组

使用查分约束把不等式组的求解建模为图论最短路的问题。

见 DiffConstraints。