

# 最短路：基础

GKxx

## 负权？负环？

对于有向图：

- 如果有一个从起点可达的负环，那么最短路长度就是负无穷。
- 有负权但没负环不要紧。

对于无向图：

- 一般来说，我们都是将无向图的边  $\{u, v\}$  看做两条有向边  $(u, v), (v, u)$  的。
- 如果有一条从起点可达的负权边，那么最短路长度就是负无穷。

## 重边？自环？不连通？

- 重边里只有边权最小的那条有用。
- 自环如果是负的，就是负环；否则没用。
- 不连通没啥特殊的。

以下我们都假定图没有重边、没有自环。

## 松弛 Relax

$Relax(u, v, w)$  指的是尝试利用边  $(u, v)$  来更新  $v$  的最短路信息，具体来说就是

```
if (dist[u] + w(u, v) < dist[v])  
    dist[v] = dist[u] + w(u, v)
```

## 常见算法

算法	类别	时间复杂度	可否有负权	可否判断负环
Bellman-Ford	单源	$O(VE)$	可以	可以
Dijkstra	单源	$O(V^2 + E)$ $O(E \log V)$ , $O(V \log V + E)$	不可以	不可以
Floyd-Warshall	所有点对	$\Theta(V^3)$	可以	可以

**简单情况：**  $w(e) \equiv C \geq 0$

当所有边权都相等时，可以直接 BFS。

时间复杂度  $O(V + E)$ 。

## 简单情况：DAG

在一个有向无环图上，可以直接按拓扑排序扫一遍，时间复杂度  $\Theta(V + E)$ 。

```
topologically sort the vertices of G
for each vertex u, taken in topologically sorted order
    for each vertex v in G.Adj[u]
        Relax(u, v, w)
```

## Bellman-Ford

设  $dist(x, i)$  表示从起点到  $x$  经过**不超过**  $i$  条边的最短路径长度。

考虑“最后一条边”是谁

- 假设是  $(v, x)$ ，那么路径长度是  $dist(v, i - 1) + w(v, x)$
- “**不超过**”意味着  $dist(x, i - 1)$  也要算上

$$dist(x, i) = \min \{ dist(x, i - 1), dist(v, i - 1) + w(v, x) \mid (v, x) \in E \}$$



## Bellman-Ford

$$\text{dist}(x, i) = \min \{ \text{dist}(x, i - 1), \text{dist}(v, i - 1) + w(v, x) \mid (v, x) \in E \}$$

实现：难以枚举  $v$ ，难道需要图转置？

## Bellman-Ford

$$\text{dist}(x, i) = \min \{ \text{dist}(x, i - 1), \text{dist}(v, i - 1) + w(v, x) \mid (v, x) \in E \}$$

实现：难以枚举  $v$ ，难道需要图转置？

直接枚举所有  $v$ ，尝试用  $\text{dist}(v, i - 1) + w(v, x)$  更新  $\text{dist}(x, i)$

```
for (auto i = 1; i <= V - 1; ++i) {  
    for (Vertex x = 0; x != V; ++x)  
        dist[x][i] = dist[x][i - 1];  
    for (Vertex v = 0; v != V; ++v)  
        for (auto [x, w] : G[v])  
            dist[x][i] = std::min(dist[x][i], dist[v][i - 1] + w);  
}
```

## Bellman-Ford

$$\text{dist}(x, i) = \min \{ \text{dist}(x, i - 1), \text{dist}(v, i - 1) + w(v, x) \mid (v, x) \in E \}$$

发现  $\text{dist}(\cdot, i)$  只依赖于  $\text{dist}(\cdot, i - 1)$ ，所以可以滚动数组

```
for (auto i = 1; i <= V - 1; ++i) {  
    for (Vertex x = 0; x != V; ++x)  
        dist[x][i % 2] = dist[x][(i - 1) % 2];  
    for (Vertex v = 0; v != V; ++v)  
        for (auto [x, w] : G[v])  
            dist[x][i % 2] = std::min(dist[x][i % 2], dist[v][(i - 1) % 2] + w);  
}
```

空间复杂度降到了  $O(V)$ 。

## Bellman-Ford

再经过仔细的思考和优化，就有了这样的代码

```
for (auto i = 1; i <= V - 1; ++i)
    for (Vertex v = 0; v != V; ++v)
        for (auto [x, w] : G[v])
            dist[x] = std::min(dist[x], dist[v] + w);
```

# Dijkstra

贪心，建立在“绕路不会有好处”的基础上：

- 假设起点  $S$  有三条出边  $S \rightarrow A, S \rightarrow B, S \rightarrow C$ ，边权分别为 2, 5, 3。
- 我们断言：从  $S$  沿边  $(S, A)$  直接走到  $A$  一定是从  $S$  到  $A$  的最短路径，从  $B$  或  $C$  绕道不可能更短。
- 边权必须非负，否则绕道有可能更短。

# Dijkstra

```
S = empty set
Q = G.V
while Q is not empty
    u = ExtractMin(Q)
    S = S + {u}
    for each vertex v in G.Adj[u]
        Relax(u, v, w)
```

$S$  : 目前为止，距离起点的最短路径已经确定的结点的集合

$Q$  : 作为集合来说就是  $V \setminus S$ ，但是它通常会藉由一个特别的数据结构实现。

# Dijkstra

```
S = empty set
Q = G.V
while Q is not empty
    u = ExtractMin(Q)
    S = S + {u}
    for each vertex v in G.Adj[u]
        Relax(u, v, w)
```

最外层的 `while` 循环共执行了  $|V|$  次。

每执行一次，都有一个新的结点的距离起点的最短路径长度被确定。

- 这个结点就是在此时仍不在  $S$  中的结点中，目前已知的距起点最近的那个。
- 然后，利用它的出边进行松弛。

时间复杂度：取决于图怎么存，以及 `ExtractMin` 和 `Relax` 能做多快。

## Dijkstra 时间复杂度

```
S = empty set
Q = G.V
while Q is not empty
    u = ExtractMin(Q)
    S = S + {u}
    for each vertex v in G.Adj[u]
        Relax(u, v, w)
```

邻接矩阵：慢就慢在枚举  $u$  的所有出边需要  $O(V)$  时间，所以必然  $\Theta(V^2)$ ，

`ExtractMin` 和 `Relax` 做多快都白扯。



## Dijkstra 时间复杂度

邻接表无优化： `ExtractMin(Q)` 就花  $O(V)$  的时间找， `Relax(u, v, w)` 只需  $O(1)$ 。

```
S = empty set
Q = G.V
while Q is not empty           //  $O(V)$ 
    u = ExtractMin(Q)          //  $O(V)$ 
    S = S + {u}
    for each vertex v in G.Adj[u] //  $O(\deg(u))$ 
        Relax(u, v, w)         //  $O(1)$ 
```

时间复杂度为  $O(V^2 + \sum_{u \in V} \deg(u)) = O(V^2 + E) = O(V^2)$ 。

- 注意这里对于两重循环的分析，并不是看到两重循环就平方。

# Dijkstra 时间复杂度

邻接表 + 二叉堆：

```
S = empty set
Q = G.V
while Q is not empty
    u = ExtractMin(Q)
    S = S + {u}
    for each vertex v in G.Adj[u]
        if (d[u] + w(u, v) < d[v])
            d[v] = d[u] + w(u, v)
```

```
S = empty set
Q = MakeHeap(G.V)
while Q is not empty
    u = Q.Top(); Q.Pop()
    S = S + {u}
    for each vertex v in G.Adj[u]
        if (d[u] + w(u, v) < d[v])
            d[v] = d[u] + w(u, v)
            Q.DecreaseKey(v, d[v])
```

Q 用一个二叉堆实现，可以将 `ExtractMin` 和 `Relax` 都做到  $O(\log V)$ 。

总时间复杂度为  $O(V \log V + E \log V) = O(E \log V)$

- 真的是“优化”吗？

# Dijkstra 时间复杂度

邻接表 + 斐波那契堆：

```
S = empty set
Q = G.V
while Q is not empty
    u = ExtractMin(Q)
    S = S + {u}
    for each vertex v in G.Adj[u]
        if (d[u] + w(u, v) < d[v])
            d[v] = d[u] + w(u, v)
```

```
S = empty set
Q = MakeHeap(G.V)
while Q is not empty
    u = Q.Top(); Q.Pop()
    S = S + {u}
    for each vertex v in G.Adj[u]
        if (d[u] + w(u, v) < d[v])
            d[v] = d[u] + w(u, v)
            Q.DecreaseKey(v, d[v])
```

和二叉堆相比，斐波那契堆可以将 `DecreaseKey` 做到  $O(1)$ ，所以总时间复杂度为  $O(V \log V + E)$ 。

- 真的是“优化”吗？

## Dijkstra 时间复杂度

- 邻接矩阵： $O(V^2)$
- 邻接表朴素： $O(V^2 + E)$
- 邻接表+二叉堆： $O(V \log V + E \log V) = O(E \log V)$
- 邻接表+斐波那契堆： $O(V \log V + E)$

## Floyd-Warshall

所有点对之间的最短路径，动态规划算法。

考虑  $f_k(i, j)$  表示从  $i$  到  $j$  只经过  $\{1, \dots, k\}$  中的结点的最短路径长度。

# Floyd-Warshall

所有点对之间的最短路径，动态规划算法。

- 考虑  $f_k(i, j)$  表示从  $i$  到  $j$  只经过  $\{1, \dots, k\}$  中的结点的最短路径长度。
- 在所有只经过  $\{1, \dots, k-1\}$  中的结点的路径里尝试加入结点  $k$
- 如果从  $i$  先走到  $k$  再走到  $j$  更短，就更新
- $$f_k(i, j) = \min \{f_{k-1}(i, j), f_{k-1}(i, k) + f_{k-1}(k, j)\}$$

## Floyd-Warshall

$$f_k(i, j) = \min \{f_{k-1}(i, j), f_{k-1}(i, k) + f_{k-1}(k, j)\}$$

类似于Bellman-Ford的实现， $k$  这一维不需要存，可以直接滚动。

```
for (Vertex k = 0; k != V; ++k)
    for (Vertex i = 0; i != V; ++i)
        for (Vertex j = 0; j != V; ++j)
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
```

# 记路径

对于单源最短路径，在松弛的时候记录前驱结点

```
for (auto [v, w] : G[x])  
    if (dist[x] + w < dist[v]) {  
        dist[v] = dist[x] + w;  
        pre[v] = x;  
    }
```

之后顺着 `pre` 数组往前走，就相当于是在最短路径上倒着走，直到走到起点 `s`。

```
std::vector<int> reversed_path;  
while (x != s) {  
    reversed_path.push_back(x);  
    x = pre[x];  
}  
reversed_path.push_back(s);
```



## 记路径

对于Floyd-Warshall, 为每一对结点记录最后一次更新用的中间结点 `p[i][j]`。

```
if (dist[i][k] + dist[k][j] < dist[i][j]) {  
    dist[i][j] = dist[i][k] + dist[k][j];  
    p[i][j] = k;  
}
```

递归地找出路径：从 `i` 到 `j` 的最短路就是从 `i` 到 `p[i][j]` 的最短路 + 从 `p[i][j]` 到 `j` 的最短路。

# 负环

Bellman-Ford 和 Floyd-Warshall 都可以判负环：

- Bellman-Ford 运行完之后，枚举每一条边，如果还能松弛，说明有能从起点到达的负环。
- Floyd-Warshall 运行完之后，如果出现 `dist[i][i] < 0`，说明有能从 `i` 到达的负环。

# 计数

最短路径可能有多条，统计最短路径的数量。

对于单源最短路径，记录 `cnt[x]` 表示从起点到 `x` 的最短路径数量。松弛时

- 如果 `dist[x] + w < dist[v]`，说明找到了一条从起点到 `v` 的更短的路径，则更新 `cnt[v] = cnt[x]`。
- 如果 `dist[x] + w == dist[v]`，说明找到了一条和已知的最短路径一样短的路径，则累加 `cnt[v] += cnt[x]`。

## 计数

Floyd-Warshall同理，记录 `cnt[i][j]` 表示从 `i` 到 `j` 的最短路径数量，松弛时更新。

乘法原理：`i` 到 `k` 的最短路径数量乘以 `k` 到 `j` 的最短路径数量。

```
if (dist[i][k] + dist[k][j] < dist[i][j]) {  
    dist[i][j] = dist[i][k] + dist[k][j];  
    cnt[i][j] = cnt[i][k] * cnt[k][j];  
} else if (dist[i][k] + dist[k][j] == dist[i][j])  
    cnt[i][j] += cnt[i][k] * cnt[k][j];
```

## A\*

本质是搜索算法，而 CS101 从最短路的角度讲 A\* 。

思想：设计一个 heuristic function  $h(v)$ ，猜测从  $v$  到终点的距离。

每一次拿出  $dist(v) + h(v)$  最小的结点，用它的出边进行松弛（通常称为 expand）

- 如果  $h(v) = 0$ ，它 (graph search) 就是 Dijkstra。

# Tree search 和 Graph search

**function** TREE-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

**loop do**

**if** the frontier is empty **then return** failure

    choose a leaf node and remove it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

    expand the chosen node, adding the resulting nodes to the frontier

---

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

*initialize the explored set to be empty*

**loop do**

**if** the frontier is empty **then return** failure

    choose a leaf node and remove it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

*add the node to the explored set*

    expand the chosen node, adding the resulting nodes to the frontier

*only if not in the frontier or explored set*

## Tree search 和 Graph search

Tree search : **自以为**自己在 tree 上 search

- 你在 tree 上遍历的时候, **从不担心**重复访问已走过的结点。

Graph search : **自以为**自己在 graph 上 search

- 你在 graph 上遍历的时候需要记录已经走过的结点, 不能反复走。

但实际问题是一个 graph , 所以 tree search **比** graph search **更有可能找到 optimal solution**。

- 想要让 graph search 找到 optimal solution , 就需要更好的 heuristic function 。