Dijkstra：带权有向图单源最短路径算法

张昊　　黄学文

苏州大学计算机科学与技术学院 2019 级图灵班

2021/12/27

目录

问题描述

最优子结构

贪心策略：Dijkstra 算法

- 贪心策略
- 松弛操作
- 伪代码
- 时间复杂度

贪心选择性质

重构最优解

案例分析

高级编程语言实现

问题描述

在**最短路径问题**中，给定一个带权重的有向图 $G = (V, E)$ 和权重函数 $w : E \to \mathbb{R}$，该权重函数将每条边映射到实数值的权重上。图中一条路径 $p =< v_0, v_1, \cdots, v_k >$ 的权重 $w(p)$ 是构成该路径的所有边的权重之和：

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

定义从结点 u 到结点 v 的**最短路径权重** $\delta(u, v)$ 如下：

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{if exist a path from } u \text{ to } v \\ \infty, & \text{others} \end{cases}$$

结点 u 到结点 v 的**最短路径**定义为任何一条权重为 $w(p) = \delta(u, v)$ 的从结点 u 到结点 v 的路径。而**单源最短路径问题**是给定一个图 $G = (V, E)$，找到从给定源结点 $s \in V$ 到每个结点 $v \in V$ 的最短路径。

最优子结构

最短路径算法通常依赖最短路径的一个重要性质：两个结点之间的一条最短路径包含着其他的最短路径。
具体地，我们使用如下的引理来描述最短路径问题的最优子结构性质。

最优子结构

给定带权重的有向图 $G = (V, E)$ 和权重函数 $w : E \to \mathbb{R}$。
设 $p =< v_0, v_1, \cdots, v_k >$ 是从结点 v_0 到 v_k 结点的一条最短路径，并且对于任意的 i 和 j，$0 \le i \le j \le k$，
设 $p_{ij} =< v_i, v_{i+1}, \cdots, v_j >$ 为路径 p 中从结点 v_i 到结点 v_j 的子路径。
那么 p_{ij} 是从结点 v_i 到结点 v_j 的一条最短路径。

证明

使用“剪切-粘贴”技术证明。
若将路径 p 分解为

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

则有 $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$。
假设存在一条从 v_i 到 v_j 的路径 $p_{ij}^{'}$，且 $w(p_{ij}^{'}) < w(p_{ij})$。则

$$p^{'} : v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}^{'}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

是一条从结点从 v_0 到结点 v_k 的权重为

$$w(p^{'}) = w(p_{0i}) + w(p_{ij}^{'}) + w(p_{jk})$$

的路径。
但是，$w(p^{'}) < w(p)$，这与 p 是从结点 v_0 到 v_k 结点的一条最短路径相矛盾。
证毕。

递归式

进一步我们有：

若 $p_{ij} =< v_i, \cdots, v_r, \cdots, v_j >$ 是从结点 v_i 到 v_j 结点的一条最短路径，
其中 $0 \leq i \leq r < j < V$，
则子路径 $p_{ir} =< v_i, \cdots, v_r >$ 是从结点 v_i 到 v_r 结点的一条最短路径。

用 $d[i,j]$ 表示从结点 v_i 到 v_j 结点的最短路径权重（$0 \leq i \leq j \leq V$）。
根据最优子结构，我们可以写出递归式：

$$d[i,j] = \begin{cases} 0, & \text{if } i = j \\ w(i,j), & \text{if } j = i+1 \\ \min\{d[i,r] + d[r,j]\}, & \text{if } i < j \end{cases}$$

因此我们可以使用动态规划来解决这一问题。
但是，我们可以观察到这一问题有更直观的解法。

贪心策略：Dijkstra 算法

- 一个很直观的想法是，我们每次在选择新的结点时都选择“最近”的那个结点，这是一种贪心的策略。
- Dijkstra 算法的主要思想是这种贪心策略。

Dijkstra 算法解决的是带权重的有向图上的单源最短路径问题。

Dijkstra 算法

算法在运行过程中维护一组结点的集合 S，该集合中保存了从源结点 s 已经找到了最短路径的结点。算法重复地从集合 $V-S$ 中选择最短路径估计（即 d 属性）最小的结点 u，将 u 加入集合 S，然后对所有从 u 出发的边进行松弛。重复这个动作，直至 S 包含了图的所有结点，即 $S=V$。

特别地，该算法要求所有边的权重都为**非负值**，即 $\forall(u,v)\in E$，都有 $\delta(u,v)\geq 0$。

松弛操作

对于每个结点 v 维护一个属性 $v.d$ 来记录从源结点 s 到结点 v 的最短路径权重的上界，称 $v.d$ 为 s 到 v 的**最短路径估计**。
可以使用下面运行时间为 $O(V)$ 的算法来对最短路径估计和前驱结点进行初始化。

算法 3 INITIALIZE-SINGLE-SOURCE(G, s)

```
for ∀v ∈ G.V do
    v.d = ∞
    v.π = NIL
end for
s.d = 0
```

- 在初始化操作结束后，对于所有的结点 $v \in V$，有 $v.\pi = \text{NIL}$，$s.d = 0$
- 对于所有的结点 $v \in V - \{s\}$，我们有 $v.d = \infty$

松弛操作

对一条边 (u, v) 的**松弛**过程为：

- 首先测试是否可以对从 s 到 v 的最短路径进行改善
- 测试的方法是，比较从结点 s 到结点 u 之间的最短路径距离加上结点 u 到结点 v 之间的边权重，以及当前的结点 s 到结点 v 的最短路径估计
- 如果前者更小，则对 $v.d$ 和 $v.\pi$ 进行更新。

可以使用算法对边 (u, v) 在 $O(1)$ 时间内进行松弛操作。

算法 4 RELAX(u, v, w)

```
if u.d + w(u, v) < v.d then
    v.d = u.d + w(u, v)
    v.π = u
end if
```

贪心策略：Dijkstra 算法

Dijkstra 算法的核心动作就是从集合 $V - S$ 选择“最近”的结点加入到 S 中，并检查新加入的结点是否可以到达其他结点，并且从源结点 s 通过该结点到达其他结点的路径权重估计是否比之前的估计要小，若是则更新。

算法 1 DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$ **do**
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for $v \in G.Adj[u]$ **do** // (u, v) 所有从 u 出发的边
        RELAX(u, v, w)
    end for
end while

伪代码

算法 1 DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for v ∈ G.Adj[u] do   // (u, v) 所有从 u 出发的边
        RELAX(u, v, w)
    end for
end while
```

- 上面的算法给出了一种实现方式，使用一个最小优先队列来维护结点集，每个结点的关键值为最短路径估计。
- 这样我们就可以使用优先队列的 EXTRACT-MIN 方法来以 $O(V)$ 的代价取出最短路径估计最小的结点。

时间复杂度

算法使用一个最小优先队列来维护结点集，使用到了三种优先队列操作来维持最小优先队列：

- INSERT (算法第 3 行构造优先队列隐含的操作)
- EXTRACT-MIN (算法第 5 行)
- DECREASE-KEY (算法第 8 行调用的 RELAX 操作中调整 d 属性时)

- 该算法对每个结点调用一次 INSERT 和 EXTRACT-MIN 操作。
- 因为每个结点仅被加入到集合 S 一次，邻接链表 Adj$[u]$ 中的每条边也在算法第 7-9 行的 **for** 循环里只被检查一次，且只会遍历这一次。
- 由于所有邻接链表中的边的总数为 E，**for** 循环的执行次数一共为 E 次。
- 因此，该算法调用 DECREASE-KEY 最多 E 次。

时间复杂度

Djkstra 算法的总运行时间依赖于最小优先队列的实现，我们考虑三种实现方式，其各关键操作与算法的执行时间如表所示。

实现方法	数组	二叉堆	斐波那契堆[1]
INSERT	$O(1)$	$O(\lg V)$	$O(1)$
EXTRACT-MIN	$O(V)$	$O(\lg V)$	$O(\lg V)$
DECREASE-KEY	$O(1)$	$O(\lg V)$	$O(1)$
总执行时间	$O(V^2+E)$	$(V+E)O(\lg V)$	$VO(\lg V)+E$

表: 不同最小优先队列实现的各关键操作与算法的执行时间。特别的，构建最小二叉堆的成本为 $O(V)$。

[1] 有关斐波那契堆的介绍请参阅《算法导论 (第三版)》第 19 章。

时间复杂度

- 如果我们讨论的是稠密图，使用数组实现的最小优先队列可以使得算法总执行时间为 $O(|V|^2 + |E|) = O(|V|^2)$；
- 如果我们讨论的是稀疏图，特别地，如果 $|E| = o(|V|/\lg|V|)$，则可以使用二叉堆来实现最小优先队列。
  算法的总执行时间为 $(|V| + |E|)O(\lg|V|)$。
  若所有结点都可以从源结点到达，则该时间为 $O(|E|\lg|V|)$。
  若 $|E| = o(|V|/\lg|V|)$，则该时间成本相对于直接实现的 $O(|V|^2)$ 成本有所改善。
- 事实上，可以将 Dijkstra 算法的运行时间改善到 $|V|O(\lg|V|) + |E|$，方法是使用斐波那契堆[2]来实现最小优先队列。

[2] 实际上，任何能够将 DECREASE-KEY 操作的摊还代价降低到 $o(\lg|V|)$ 而又不增加 EXTRACT-MIN 操作的摊还代价的方法都将产生比二叉堆的渐近性能更优的实现。

贪心选择性质

- 虽然贪心策略并不总是能够获得最优结果，但是使用贪心策略的 Dijkstra 算法确实能够计算得到最短路径。
- 这正因为我们可以找出下面的贪心选择性质，并且可以证明局部最优解可以作为全局的最优解，从而得到最优子结构。

根据最短路径估计的定义，为了保证贪心策略能够最优，这里贪心选择性质可以表述如下：

贪心选择性质

对于将加入集合 S 的结点 $u \in V - S$，有 $u.d = \delta(s, u)$。

即从集合 $V - S$ 中选择最短路径估计（即 d 属性）最小的结点 u，该结点的最短路径估计等于从源结点 s 到结点 u 的最短路径权重。

贪心选择性质证明

下面我们使用反证法来证明利用这一性质是可以达到局部最优的。

假设结点 u 是第一个加入集合 S 后使得结论不成立的结点，即 $u.d \neq \delta(s,u)$。

当 $u = s$ 时，即 $u = s$ 是第一个加入到集合 S 中的结点。
这样就有 $s.d = \delta(s,s) = 0$，这与我们的假设 $u.d \neq \delta(s,u)$ 矛盾。

故下面讨论 $u \neq s$ ，此时有 $S \neq \emptyset$。

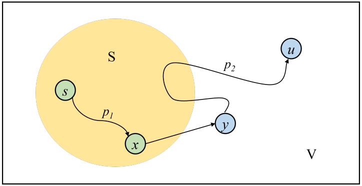
若不存在一条从 s 到 u 的路径，根据非路径性质有 $u.d = \delta(s,u) = \infty$，这与我们的假设 $u.d \neq \delta(s,u)$ 矛盾。

非路径性质

如果从结点 s 到结点 v 之间不存在路径，则总是有 $v.d = \delta(s,v) = \infty$。

贪心选择性质证明 (续)

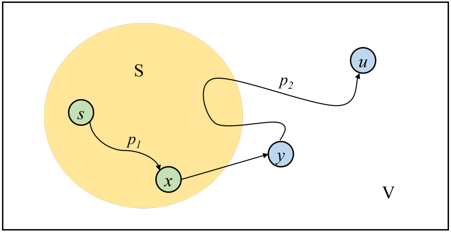
因此，一定存在一条从 s 到 u 的路径;
同样一定存在一条从 s 到 u 的最短路径 p。
考虑路径 p 的第一个满足 $y \in V - S$ 的结点，结点 x 是结点 y 的直接前驱结点，显然 $x \in S$。



如图，至此路径 p 就可以分解为:

$$s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$$

贪心选择性质证明 (续)



$$s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$$

- 结点 y 在结点 u 之前或就是结点 u;
- 结点 x 在结点 s 之前或就是结点 s。
- 路径 p_1, p_2 可能不包含任和边,
- 路径 p_2 可能重新进入 S 也可能不进入 S。

贪心选择性质证明 (续)

接下来我们将推导矛盾来证明结论。

由于我们的假设是结点 u 是第一个不满足结论 $u.d = \delta(s, u)$ 的结点，因此 $x \in S$ 且 x 在 u 加入 S 之前就已经加入 S。
故 x 在加入 S 时，$x.d = \delta(s, x)$，且边 (x, y) 将松弛，由收敛性质有：

$$y.d = \delta(s, y) \tag{1}$$

收敛性质

对于某些结点 $v \in V$，如果 $s \rightsquigarrow u \rightarrow v$ 是图 G 中的一条最短路径，并且在对边 (u, v) 进行松弛前的任意时间有 $u.d = \delta(s, u)$，则在之后的所有时间有 $v.d = \delta(s, v)$。

贪心选择性质证明 (续)

由于 y 是最短路径 $s \overset{p}{\leadsto} u$ 上在结点 u 之前的节点，且权重值非负 [3]，故有 $\delta(s,y) \leq \delta(s,u)$，因此由式1有：

$$y.d = \delta(s,y) \leq \delta(s,u)$$

由上界性质，有 $u.d \geq \delta(s,u)$，因此：

$$y.d = \delta(s,y) \leq \delta(s,u) \leq u.d \tag{2}$$

上界性质

对于所有结点 $v \in V$，总是有 $v.d \geq \delta(s,v)$。一旦 $v.d$ 的取值达到 $\delta(s,v)$，其值将不再发生变化。

[3] 如果带权有向图的权重存在负值，那么这一结论就不会成立，因此 Dijkstra 算法要求所有边的权重都为非负。

贪心选择性质证明 (续)

因为算法每次选择的结点 u 都是最短路径估计最小的结点，
且选择结点 u 时， u 和 y 都在集合 $V-S$ 中，故有

$$u.d \leq y.d \tag{3}$$

由式2和式3可得：

$$y.d = \delta(s,y) = \delta(s,u) = u.d$$

这与我们的假设 $u.d \neq \delta(s,u)$ 矛盾，故结论成立。
证毕。

Dijkstra 算法的正确性

至此，我们已经证明了贪心选择在局部范围内是最优的。
接下来我们证明使用这一贪心选择性质完成的算法是全局最优的，我们将引出如下定理并证明。

Dijkstra 算法的正确性

Dijkstra 算法运行在带权有向图 $G = (V, E)$ 时，如果所有边的权重均为非负值，那么算法终止时，对 $\forall u \in V$，有 $u.d = \delta(s, u)$。

我们可以发现，算法中 4-10 行的 **while** 循环每次开始之前都有

$$\forall v \in S, v.d = \delta(s, v)$$

这是一个循环不变式。我们将利用这一循环不变式来证明这一定理。

证明

1. **初始化** 开始时，$S=\emptyset$，循环不变式直接成立。
2. **保持** 我们注意到，上面对贪心选择性质证明了：
   对于每个节点 $u \in V$，当加入到集合 S 时都有 $u.d=\delta(s,u)$。
   因此根据上界性质，一旦 $u.d=\delta(s,u)$，其值将不再发生变化。
   故该等式在后续所有时间内都将保持不变。
3. **终止** 算法终止时，$Q=V-S=\emptyset$，因此有 $S=V$，
   所以 $\forall v \in S, v.d=\delta(s,v)$。

证毕。

至此，我们就证明了贪心选择是全局最优的。
可以保证，作出贪心选择后，原问题的最优解总是存在，
且贪心选择的局部最优能生成全局最优解，从而保证 Dijkstra 这一贪心算法是正确的。

重构最优解

在算法中，我们除了保存最短路径权重外，还维护了前驱节点 π 值，并可在算法结束后导出前驱子图 G_π。
我们使用下面的推论来重构最优解。

推论

如果在带权有向图 $G = (V, E)$ 上运行 Dijkstra 算法，其中的权重均为非负值，源结点为 s，那么算法终止时，前驱子图 G_π 是一颗根节点为 s 的最短路径树。

证明：根据之前定理的证明，算法运行结束时，对于所有的结点 $v \in V$，$v.d = \delta(s, v)$。因此根据前驱子图性质，前驱子图 G_π 是一棵根结点为 s 的最短路径树。

前驱子图性质

对于所有的结点 $v \in V$，一旦 $v.d = \delta(s, v)$，则前驱子图是一棵根结点为 s 的最短路径树。

重构最优解

推论

如果在带权有向图 $G = (V, E)$ 上运行 Dijkstra 算法，其中的权重均为非负值，源结点为 s，那么算法终止时，前驱子图 G_π 是一颗根节点为 s 的最短路径树。

最短路径树包括了从源结点 s 到每一个可以从 s 到达的结点的一条最短路径。将从结点 v 开始的前驱结点链反转过来，就是从 s 到 v 的一条最短路径。因此，可以使用下面的算法打印出从 s 到 u 的一条最短路径。

算法 2 PRINT-PATH(G, s, v)

```
if v = s then
    print s
else if v.π = NIL then
    print no path from s to v exists
else
    PRINT-PATH(G, s, v.π)
    print v
end if
```

案例分析

我们以下图所示的带权有向图作为例子。结点 v_0 为源节点。

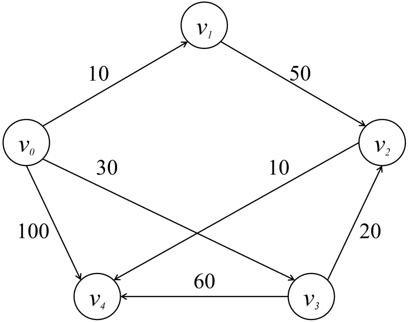


图: 带权重的有向图，边上的数字代表权重。

该例由 5 个结点和 7 条有向边组成，边权均为非负数。

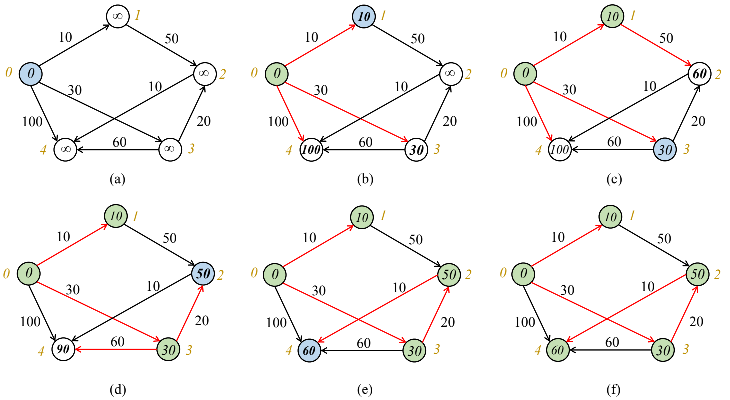
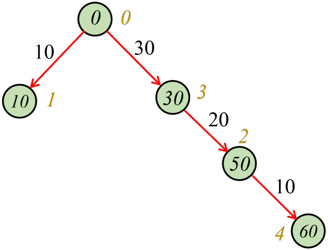


图: Dijkstra 算法的执行过程。结点的下标以黄色字体标注在结点旁。结点 v_0 为源节点。每个结点中的数值为该结点的最短路径估计值（d 值），加粗的数字表示每轮 **while** 循环结束后更新的 d 值。边上的数字代表权重，红色的边表示前驱。绿色的结点属于结合 S，白色的结点属于最小优先队列 $Q = V - S$，蓝色的结点为选定的最短路径估计最小的结点 u。图 (a) 为 **while** 循环首次执行前的场景；图 (b)-(f) 为每轮 **while** 循环成功执行后的场景。最后在图 (f) 中的 d 值和前驱均为最终值。

案例分析：重构最优解



- $v_0 \rightsquigarrow v_1 :< v_0, v_1 >, \ \delta(v_0, v_1) = 10$
- $v_0 \rightsquigarrow v_2 :< v_0, v_3, v_2 >, \ \delta(v_0, v_2) = 50$
- $v_0 \rightsquigarrow v_3 :< v_0, v_3 >, \ \delta(v_0, v_3) = 30$
- $v_0 \rightsquigarrow v_4 :< v_0, v_3, v_2, v_4 >, \ \delta(v_0, v_4) = 60$

高级编程语言实现

使用 Python 3 实现了上述算法。（代码略）
选取了案例分析以及《算法导论 (第三版)》24.3 节图 24-6 的实例进行测试。

```
运行:  main ×
开始初始化图
请输入结点: v0 v1 v2 v3 v4
请输入边数: 7
接下来7行，请输入边的信息，格式为: 起点 终点 权重，每条边一行
(7 left) >>> v0 v1 10
(6 left) >>> v0 v3 30
(5 left) >>> v0 v4 100
(4 left) >>> v1 v2 50
(3 left) >>> v2 v4 10
(2 left) >>> v3 v2 20
(1 left) >>> v3 v4 60
请输入开始结点: v0
图初始化完毕
运行结果:
p(v0, v3): v0 v3 , 路径长度: 30.0
p(v0, v1): v0 v1 , 路径长度: 10.0
p(v0, v4): v0 v3 v2 v4 , 路径长度: 60.0
p(v0, v2): v0 v3 v2 , 路径长度: 50.0
```

图: 运行结果。图为本文案例分析中案例的运行结果。

高级编程语言实现

```
运行:  main ×
开始初始化图
请输入结点: s x y z t
请输入边数: 10
接下来10行，请输入边的信息，格式为: 起点 终点 权重，每条边一行
(10 left) >>> s t 10
(9 left) >>> t x 1
(8 left) >>> s y 5
(7 left) >>> t y 2
(6 left) >>> x z 4
(5 left) >>> y t 3
(4 left) >>> y x 9
(3 left) >>> y z 2
(2 left) >>> z s 7
(1 left) >>> z x 6
请输入开始结点: s
图初始化完毕
运行结果:
p(s, x): s y t x , 路径长度: 9.0
p(s, y): s y , 路径长度: 5.0
p(s, t): s y t , 路径长度: 8.0
p(s, z): s y z , 路径长度: 7.0
```

图: 运行结果。图为《算法导论 (第三版)》*24.3* 节图 *24-6* 实例的运行结果。

谢谢！