

# 44|最短路径：地图软件是如何计算出最优出行路径的？

基础篇的时候，我们学习了图的两种搜索算法，深度优先搜索和广度优先搜索。这两种算法主要是针对无权图的搜索算法。针对有权图，也就是图中的每条边都有一个权重，我们该如何计算两点之间的最短路径（经过的边的权重和最小）呢？今天，我就从地图软件的路线规划问题讲起，带你看看常用的最短路径算法（Shortest Path Algorithm）。

像Google地图、百度地图、高德地图这样的地图软件，我想你应该经常使用吧？如果想从家开车到公司，你只需要输入起始、结束地址，地图就会给你规划一条最优出行路线。这里的最优，有很多种定义，比如最短路线、最少用时路线、最少红绿灯路线等等。作为一名软件开发工程师，你是否思考过，地图软件的最优路线是如何计算出来的吗？底层依赖了什么算法呢？

## 算法解析

我们刚提到的最优问题包含三个：最短路线、最少用时和最少红绿灯。我们先解决最简单的，最短路线。

解决软件开发中的实际问题，最重要的一点就是建模，也就是将复杂的场景抽象成具体的数据结构。针对这个问题，我们该如何抽象成数据结构呢？

我们之前也提到过，图这种数据结构的表达能力很强，显然，把地图抽象成图最合适不过了。我们把每个岔路口看作一个顶点，岔路口与岔路口之间的路看作一条边，路的长度就是边的权重。如果路是单行道，我们就在两个顶点之间画一条有向边；如果路是双行道，我们就在两个顶点之间画两条方向不同的边。这样，整个地图就被抽象成一个有向有权图。

具体的代码实现，我放在下面了。于是，我们要求解的问题就转化为，在一个有向有权图中，求两个顶点间的最短路径。

```
public class Graph { // 有向有权图的邻接表表示
    private LinkedList<Edge> adj[]; // 邻接表
    private int v; // 顶点个数

    public Graph(int v) {
        this.v = v;
        this.adj = new LinkedList[v];
        for (int i = 0; i < v; ++i) {
            this.adj[i] = new LinkedList<>();
        }
    }

    public void addEdge(int s, int t, int w) { // 添加一条边
        this.adj[s].add(new Edge(s, t, w));
    }

    private class Edge {
        public int sid; // 边的起始顶点编号
        public int tid; // 边的终止顶点编号
        public int w; // 权重
        public Edge(int sid, int tid, int w) {
            this.sid = sid;
            this.tid = tid;
            this.w = w;
        }
    }
    // 下面这个类是为了dijkstra实现用的
    private class Vertex {
```

```
public int id; // 顶点编号ID
public int dist; // 从起始顶点到这个顶点的距离
public Vertex(int id, int dist) {
    this.id = id;
    this.dist = dist;
}
}
```

想要解决这个问题，有一个非常经典的算法，最短路径算法，更加准确地说，是单源最短路径算法（一个顶点到一个顶点）。提到最短路径算法，最出名的莫过于Dijkstra算法了。所以，我们现在来看，Dijkstra算法是怎么工作的。

这个算法的原理稍微有点儿复杂，单纯的文字描述，不是很好懂。所以，我还是结合代码来讲解。

// 因为Java提供的优先级队列，没有暴露更新数据的接口，所以我们需要重新实现一个

```
private class PriorityQueue { // 根据vertex.dist构建小顶堆
```

```
private Vertex[] nodes;
private int count;
public PriorityQueue(int v) {
    this.nodes = new Vertex[v+1];
    this.count = v;
}
public Vertex poll() { // TODO: 留给读者实现... }
public void add(Vertex vertex) { // TODO: 留给读者实现... }
// 更新结点的值，并且从下往上堆化，重新符合堆的定义。时间复杂度O(logn)。
public void update(Vertex vertex) { // TODO: 留给读者实现... }
public boolean isEmpty() { // TODO: 留给读者实现... }
}
```

```
public void dijkstra(int s, int t) { // 从顶点s到顶点t的最短路径
```

```
int[] predecessor = new int[this.v]; // 用来还原最短路径
Vertex[] vertexes = new Vertex[this.v];
for (int i = 0; i < this.v; ++i) {
    vertexes[i] = new Vertex(i, Integer.MAX_VALUE);
}
PriorityQueue queue = new PriorityQueue(this.v); // 小顶堆
boolean[] inqueue = new boolean[this.v]; // 标记是否进入过队列
vertexes[s].dist = 0;
queue.add(vertexes[s]);
inqueue[s] = true;
while (!queue.isEmpty()) {
    Vertex minVertex = queue.poll(); // 取堆顶元素并删除
    if (minVertex.id == t) break; // 最短路径产生了
    for (int i = 0; i < adj[minVertex.id].size(); ++i) {
        Edge e = adj[minVertex.id].get(i); // 取出一条minVertex相连的边
        Vertex nextVertex = vertexes[e.to]; // minVertex-->nextVertex
        if (minVertex.dist + e.w < nextVertex.dist) { // 更新next的dist
            nextVertex.dist = minVertex.dist + e.w;
            predecessor[nextVertex.id] = minVertex.id;
            if (inqueue[nextVertex.id] == true) {
                queue.update(nextVertex); // 更新队列中的dist值
            } else {
                queue.add(nextVertex);
                inqueue[nextVertex.id] = true;
            }
        }
    }
}
}
// 输出最短路径
```

```
System.out.print(s);
print(s, t, predecessor);
}

private void print(int s, int t, int[] predecessor) {
    if (s == t) return;
    print(s, predecessor[t], predecessor);
    System.out.print("->" + t);
}
```

我们用vertexes数组，记录从起始顶点到每个顶点的距离（dist）。起初，我们把所有顶点的dist都初始化为无穷大（也就是代码中的Integer.MAX\_VALUE）。我们把起始顶点的dist值初始化为0，然后将其放到优先级队列中。

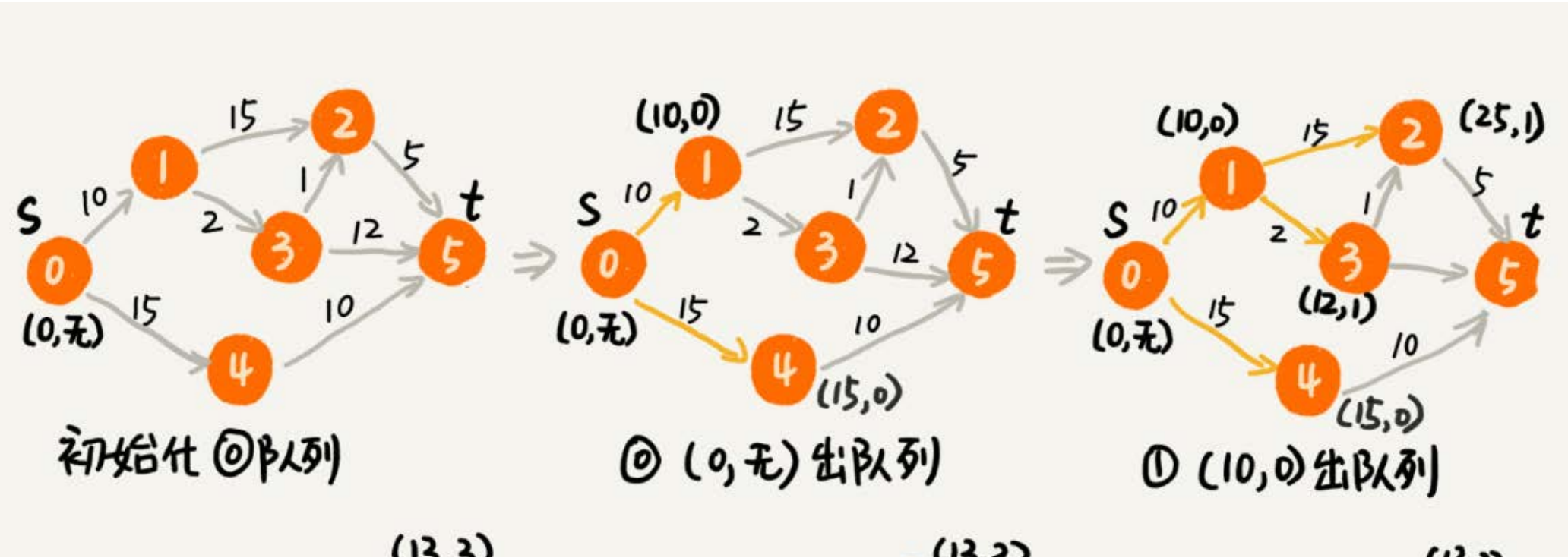
我们从优先级队列中取出dist最小的顶点minVertex，然后考察这个顶点可达的所有顶点（代码中的nextVertex）。如果minVertex的dist值加上minVertex与nextVertex之间边的权重w小于nextVertex当前的dist值，也就是说，存在另一条更短的路径，它经过minVertex到达nextVertex。那我们就把nextVertex的dist更新为minVertex的dist值加上w。然后，我们把nextVertex加入到优先级队列中。重复这个过程，直到找到终止顶点t或者队列为空。

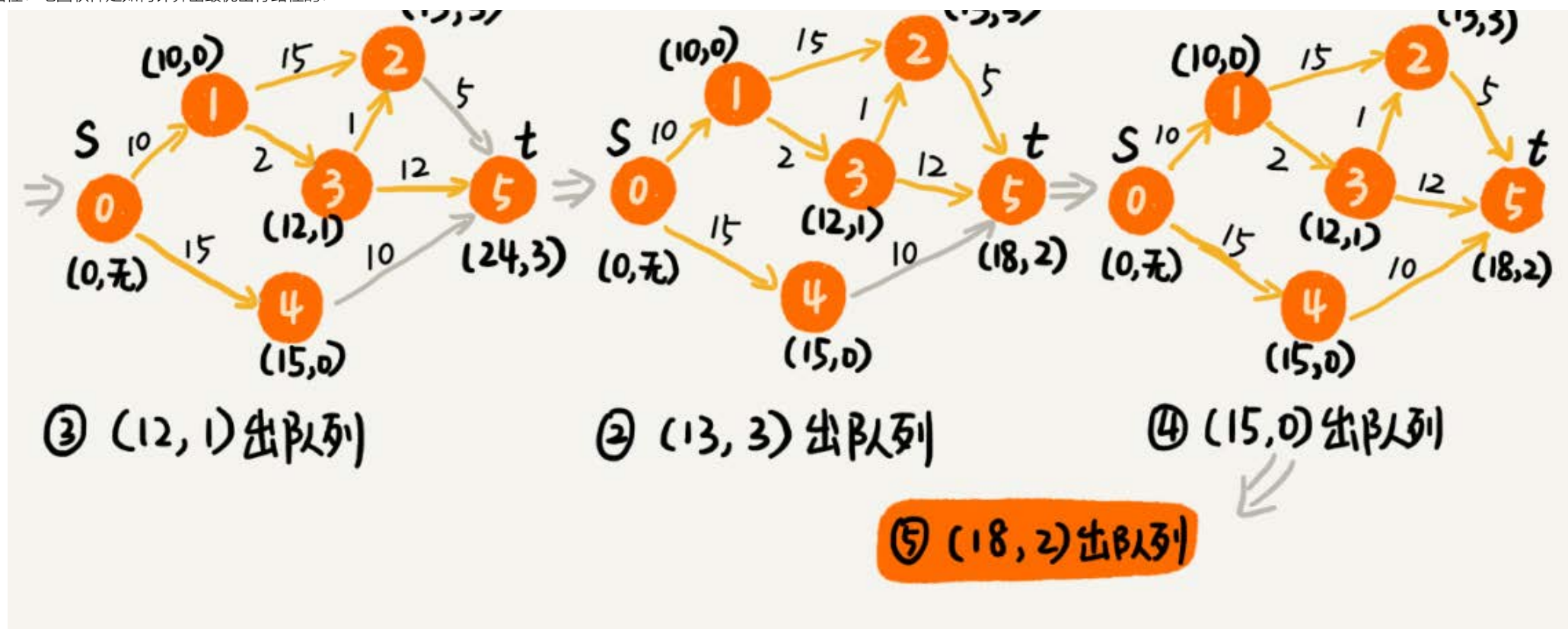
以上就是Dijkstra算法的核心逻辑。除此之外，代码中还有两个额外的变量，predecessor数组和inqueue数组。

predecessor数组的作用是为了还原最短路径，它记录每个顶点的前驱顶点。最后，我们通过递归的方式，将这个路径打印出来。打印路径的print递归代码我就不详细讲了，这个跟我们在图的搜索中讲的打印路径方法一样。如果不理解的话，你可以回过头去看下那一节。

inqueue数组是为了避免将一个顶点多次添加到优先级队列中。我们更新了某个顶点的dist值之后，如果这个顶点已经在优先级队列中了，就不要再将它重复添加进去了。

看完了代码和文字解释，你可能还是有点懵，那我就举个例子，再给你解释一下。





理解了Dijkstra的原理和代码实现，我们来看下，**Dijkstra**算法的时间复杂度是多少？

在刚刚的代码实现中，最复杂就是while循环嵌套for循环那部分代码了。while循环最多会执行V次（V表示顶点的个数），而内部的for循环的执行次数不确定，跟每个顶点的相邻边的个数有关，我们分别记作 $E_0, E_1, E_2, \dots, E_{(V-1)}$ 。如果我们把这V个顶点的边都加起来，最大也不会超过图中所有边的个数E（E表示边的个数）。

for循环内部的代码涉及从优先级队列取数据、往优先级队列中添加数据、更新优先级队列中的数据，这样三个主要的操作。我们知道，优先级队列是用堆来实现的，堆中的这几个操作，时间复杂度都是 $O(\log V)$ （堆中的元素个数不会超过顶点的个数V）。

所以，综合这两部分，再利用乘法原则，整个代码的时间复杂度就是 $O(E \cdot \log V)$ 。

弄懂了Dijkstra算法，我们再来回答之前的问题，如何计算最优出行路线？

从理论上讲，用Dijkstra算法可以计算出两点之间的最短路径。但是，你有没有想过，对于一个超级大地图来说，岔路口、道路都非常多，对应到图这种数据结构上来说，就有非常多的顶点和边。如果为了计算两点之间的最短路径，在一个超级大图上动用Dijkstra算法，遍历所有的顶点和边，显然会非常耗时。那我们有没有什么优化的方法呢？

做工程不像做理论，一定要给出个最优解。理论上算法再好，如果执行效率太低，也无法应用到实际的工程中。对于软件开发工程师来说，我们经常要根据问题的实际背景，对解决方案权衡取舍。类似出行路线这种工程上的问题，我们没有必要非得求出个绝对最优解。很多时候，为了兼顾执行效率，我们只需要计算出一个可行的次优解就可以了。

有了这个原则，你能想出刚刚那个问题的优化方案吗？

虽然地图很大，但是两点之间的最短路径或者说较好的出行路径，并不会很“发散”，只会出现在两点之间和两点附近的区块内。所以我们可以整个大地图上，划出一个小的区块，这个小区块恰好可以覆盖住两个点，但又不会很大。我们只需要在这个小区块内部运行Dijkstra算法，这样就可以避免遍历整个大图，也就大大提高了执行效率。

不过你可能会说了，如果两点距离比较远，从北京海淀区某个地点，到上海黄浦区某个地点，那上面的这种处理方法，显然就不工作了，毕竟覆盖北京和上海的区块并不小。

我给你点提示，你可以现在打开地图App，缩小放大一下地图，看下地图上的路线有什么变化，然后再思考，这个问题该怎么解决。

对于这样两点之间距离较远的路线规划，我们可以把北京海淀区或者北京看作一个顶点，把上海黄浦区或者上海看作一个顶点，先规划大的出行路线。比如，如何从北京到上海，必须要经过某几个顶点，或者某几条干道，然后再细化每个阶段的小路线。

这样，最短路径问题就解决了。我们再来看另外两个问题，最少时间和最少红绿灯。

前面讲最短路径的时候，每条边的权重是路的长度。在计算最少时间的时候，算法还是不变，我们只需要把边的权重，从路的长度变成经过这段路所需要的时间。不过，这个时间会根据拥堵情况时刻变化。如何计算车通过一段路的时间呢？这是一个蛮有意思的问题，你可以自己思考下。

每经过一条边，就要经过一个红绿灯。关于最少红绿灯的出行方案，实际上，我们只需要把每条边的权值改为1即可，算法还是不变，可以继续使用前面讲的Dijkstra算法。不过，边的权值为1，也就相当于无权图了，我们还可以使用之前讲过的广度优先搜索算法。因为我们前面讲过，广度优先搜索算法计算出来的两点之间的路径，就是两点的最短路径。

不过，这里给出的所有方案都非常粗糙，只是为了给你展示，如何结合实际场景，灵活地应用算法，让算法为我们所用，真实的地图软件的路径规划，要比这个复杂很多。而且，比起Dijkstra算法，地图软件用的更多的是类似A\*的启发式搜索算法，不过也是在Dijkstra算法上的优化罢了，我们后面会讲到，这里暂且不展开。

## 总结引申

今天，我们学习了一种非常重要的图算法，Dijkstra最短路径算法。实际上，最短路径算法还有很多，比如Bellford算法、Floyd算法等等。如果感兴趣，你可以自己去研究。

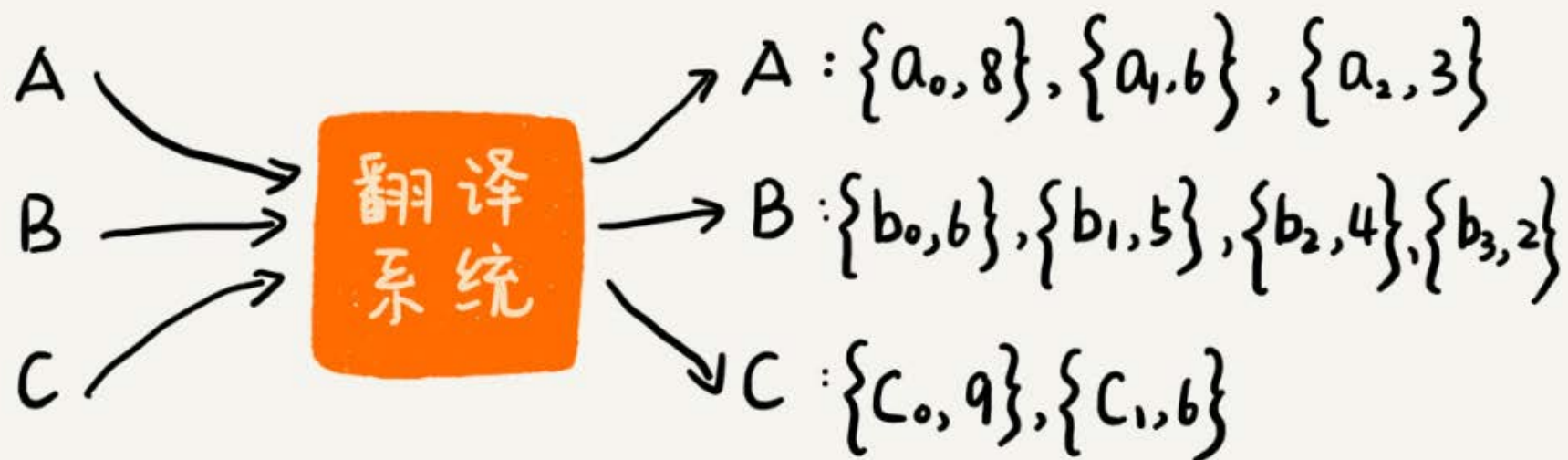
关于Dijkstra算法，我只讲了原理和代码实现。对于正确性，我没有去证明。之所以这么做，是因为证明过程会涉及比较复杂的数学推导。这个并不是我们的重点，你只要掌握这个算法的思路就可以了。

这些算法实现思路非常经典，掌握了这些思路，我们可以拿来指导、解决其他问题。比如Dijkstra这个算法的核心思想，就可以拿来解决下面这个看似完全不相关的问题。这个问题是我之前工作中遇到的真实的问题，为了在较短的篇幅里把问题介绍清楚，我对背景做了一些简化。

我们有一个翻译系统，只能针对单个词来做翻译。如果要翻译一整个句子，我们需要将句子拆成一个一个的单词，再丢给翻译系统。针对每个单词，翻译系统会返回一组可选的翻译列表，并且针对每个翻译打一个分，表示这个翻译的可信程度。



句子：A B C 包含三个单词 A, B, C



针对每个单词，我们从可选列表中，选择其中一个翻译，组合起来就是整个句子的翻译。每个单词的翻译的得分之和，就是整个句子的翻译得分。随意搭配单词的翻译，会得到一个句子的不同翻译。针对整个句子，我们希望计算出得分最高的前 $k$ 个翻译结果，你会怎么编程来实现呢？

得分最高 Top 3:

$$a_0 b_0 c_0 : 8 + 6 + 9 = 23 \text{ 分}$$

$$a_0 b_1 c_0 : 8 + 5 + 9 = 22 \text{ 分}$$

$$a_1 b_0 c_0 : 6 + 6 + 9 = 21 \text{ 分}$$

当然，最简单的办法还是借助回溯算法，穷举所有的排列组合情况，然后选出得分最高的前 $k$ 个翻译结果。但是，这样做的时间复杂度会比较高，是 $O(m^n)$ ，其中， $m$ 表示平均每个单词的可选翻译个数， $n$ 表示一个句子中包含多少个单词。这个解决方案，你可以当作回溯算法的练习题，自己编程实现一下，我就不多说了。

实际上，这个问题可以借助Dijkstra算法的核心思想，非常高效地解决。每个单词的可选翻译是按照分数从大到小排列的，所以 $a_0 b_0 c_0$ 肯定是得分最高组合结果。我们把 $a_0 b_0 c_0$ 及得分作为一个对象，放入到优先级队列中。

我们每次从优先级队列中取出一个得分最高的组合，并基于这个组合进行扩展。扩展的策略是每个单词的翻译分别替换成下一个单词的翻译。比如 $a_0 b_0 c_0$ 扩展后，会得到三个组合， $a_1 b_0 c_0$ 、 $a_0 b_1 c_0$ 、 $a_0 b_0 c_1$ 。我们把扩展之后的组合，加到优先级队列中。重复这个过程，直到获取到 $k$ 个翻译组合或者队列为空。

初始化优先级队列

$a_0b_0c_0$  23

$a_0b_0c_0$  出队列

$a_1b_0c_0$  21

$a_0b_1c_0$  22

$a_0b_0c_1$  20

$a_0b_1c_0$  出队列

$a_1b_0c_0$  21

$a_0b_0c_1$  20

$a_1b_1c_0$  20

$a_0b_2c_0$  21

$a_0b_1c_1$  19

$a_1b_0c_0$  出队列

$a_0b_0c_1$  20

$a_1b_1c_0$  20

$a_0b_2c_0$  21

$a_0b_1c_1$  19

$a_2b_0c_0$  18

$a_1b_1c_0$  重复去掉

$a_1b_0c_1$  18



我们来看，这种实现思路的时间复杂度是多少？

假设句子包含 $n$ 个单词，每个单词平均有 $m$ 个可选的翻译，我们求得分最高的前 $k$ 个组合结果。每次一个组合出队列，就对应着一个组合结果，我们希望得到 $k$ 个，那就对应着 $k$ 次出队操作。每次有一个组合出队列，就有 $n$ 个组合入队列。优先级队列中出队和入队操作的时间复杂度都是 $O(\log X)$ ， $X$ 表示队列中的组合个数。所以，总的时间复杂度就是 $O(k*n*\log X)$ 。那 $X$ 到底是多少呢？

$k$ 次出入队列，队列中的总数据不会超过 $k*n$ ，也就是说，出队、入队操作的时间复杂度是 $O(\log(k*n))$ 。所以，总的时间复杂度就是 $O(k*n*\log(k*n))$ ，比之前的指数级时间复杂度降低了很多。

## 课后思考

1. 在计算最短时间的出行路线中，如何获得通过某条路的时间呢？这个题目很有意思，我之前面试的时候也被问到过，你可以思考看看。
2. 今天讲的出行路线问题，我假设的是开车出行，那如果是公交出行呢？如果混合地铁、公交、步行，又该如何规划路线呢？

欢迎留言和我分享，也欢迎点击“请朋友读”，把今天的内容分享给你的好友，和他一起讨论、学习。



# 数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言：

- 五岳寻仙 2019-01-07 07:46:04  
课后思考题，自己能想到的。

1.获取通过某条路的时间：通过某条路的时间与①路长度②路况(是否平坦等)③拥堵情况④红绿灯个数等因素相关。获取这些因素后就可以建立一个回归模

型(比如线性回归)来估算时间。其中①②④因素比较固定，容易获得。③是动态的，但也可以通过a.与交通管理部门合作获得路段拥堵情况；b.联合其他导航软件获得在该路段的在线人数；c.通过现在时间段正好在次路段的其他用户的真实情况等方式估算。

2.混合公交、地铁和步行时：地铁时刻表是固定的，容易估算。公交虽然没那么准时，大致时间是可以估计的，步行时间受路拥堵状况小，基本与道路长度成正比，也容易估算。总之，感觉公交、地铁、步行，时间估算会比开车更容易，也更准确些。[11赞]

- 徐凯 2019-01-07 08:59:11

@五岳寻仙的答案太棒了 我感觉每条道路应该还有限速，这个因素也要考察。[4赞]

- Liam 2019-01-08 08:50:39

有2个疑问：

1 Dijkstra就是贪心算法吧？

2 它的解可能不是最优解 [2赞]

作者回复2019-01-17 15:22:34

Dijkstra实际上可以看做动态规划：)

- 許敲敲 2019-01-07 08:10:29

类似的python代码也会更新嘛，还不熟悉java的 [2赞]

- yongxiang 2019-01-12 18:19:57

王老师，我输入代码运行后，实际出队列的顺序跟图中的不一样，实际（15，0）出队列 在（13，3）出队列前面。我看了代码，应该是修改（25，1）为（13，3）的时候，小顶堆不会自动更新顺序。需要对22行进行如下修改，更新已经在队列中，又改了dist的Vertex的优先级：

```
if (inQueue[nextVertex.id] == false){
```

```
queue.add(nextVertex);
```

```
inQueue[nextVertex.id] = true;
```

```
}
```

```
else { // 更新已经在队列中，又改了dist的Vertex的优先级
```

```
queue.remove(nextVertex);
```

```
queue.add(nextVertex);
```

```
} [1赞]
```

作者回复2019-01-14 10:03:30

嗯嗯 我更新下代码

44|最短路径：地图软件是如何计算出最优出行路径的？

Geek\_dddebb 2019-01-07 14:30:04

亲测更新vertex后对象在队列中的位置不变 [1赞]

作者回复2019-01-17 15:07:34

代码已经改正，你再看下？；)

- hughieyu 2019-01-07 08:41:58

更新vertex后是否要更新一下对象在优先级队列中的位置，否则会预期更晚弹出优先级队列，会影响查找的速度，除此之外还没有可能出现其他的问题 [1赞]

作者回复2019-01-07 09:39:49

会自动更新位置的 相当于堆中更新一个节点的值

- Alexis何春光 2019-01-13 11:53:05

Dijkstra算一种动态规划算法吗

作者回复2019-01-17 15:32:11

也可以这么说：)

- 李东勇 2019-01-13 11:38:19

有兴趣的可以看下LeetCode 上这道题：<https://leetcode.com/problems/network-delay-time/>

用到的就是Dijkstra 算法

- zixuan 2019-01-11 16:27:35

想到个口水证明，不过首先需要说明下dijkstra不能用于负权图或正负混合权重的图，例如由a、b、c三个顶点构成的图：{Edge(a, b, -1), Edge(a, c, -2), Edge(b, c, -10)}, 按dijkstra算法会得出a到c的最短距离是-2，但实际是-11. 换句话说，通过把正权重取负来求最长路径是行不通的，最长路径(or 负权图的最短路径)似乎并没有高效的解法. 所以大的前提是所有权重 $\geq 0$ .

接下来证明: PriorityQueue出队列的顶点顺序是所有顶点到单源点的最短路径按路径总权重从小到大排列的顺序. Base Case: 第1个出队列的是源点本身, 最短路径总权重=0为最小（这里就要求所有权重 $\geq 0$ , 否则base都不成立）. 假设第1, 2, ...n次出队列的顶点分别对应最短路径总权重最小的前n个顶点，那么第n+1个最短路径顶点自然就是在前n个顶点的所有邻接顶点集合中取更新后总距离离源点最小的那个，即堆顶元素. 假如排第n+1的顶点t不是堆顶元素的话，其最短路径的前驱顶点s必然不在已出队列顶点集合中(否则通过s找到的t就是对顶元素)，但又由于权重 $(s, t) > 0$ , 源点到s的最短距离必然是排前n+1的，则t必须是其前驱节点s，矛盾. 老师最后举的打分的例子初看上去和最短距离没太大关系，可能是指在这个基于历史状态下的单步迭代策略(DP?)使得全局最优上有点类似吧.

- 想当上帝的司机 2019-01-10 00:45:37

if (inqueue[nextVertex.id]==false)//加了这个判断的话，就不会走2了，因为在走1的时候2已经进入inqueue了，我在本地试的是去掉这个条件结果是对的，不知道是不是语言的原因，我是用gonlang写的，优先级队列是网上找的一个插件，老师你本地跑是成功的吗

作者回复 2019-01-10 09:59:15

我测试过的 我再多找个数据测试一下

- 五岳寻仙 2019-01-09 07:23:58

@Liam 有问到 Dijkstra 算法是否是贪心算法，求得的解是否是全局最优解。

答案是：它不是贪心算法，事实上它是动态规划算法，求得的解全局最优解。

这个算法很有名，网上有很多帖子，具体可以百度或谷歌。

- 卡罗 2019-01-08 18:27:42

翻译那个例子没看懂。。。

- 纯洁的憎恶 2019-01-07 23:20:46

思考题好发散啊…我也开开脑洞吧。

1.可考虑的因素有很多，在此补充一点：如果使用我的导航app用户很多，那么就有可能掌握每条道路通行车辆的实时速度、汽车数量，于是就可以结合道路长度比较精确的计算出当前通行时间。

2.公交和地铁都有固定线路，与开车、步行情境下的路线有很大不同。如果只考虑公交，可以简化问题为只保留公交车站为顶点，站与站之间为边的有向图即可。但如果把公交和步行混合起来就复杂了，也行可以考虑公交线路覆盖区域，以车站为顶点、站与站连线为边、边的通勤时间为权重，覆盖不到的区域以岔路为顶点、道路为边、步行通勤时间为权重，构建有向图。

- 纯洁的憎恶 2019-01-07 18:33:57

回顾了一下<sup>31</sup>讲，广度优先算法的过程不难理解，但是广度优先算法遍历无权图中一点s到另一点t的路径，就是它的最短路径，是如何证明的呢？

- 你有资格吗？ 2019-01-07 11:44:58

打卡

- 传说中的成大大 2019-01-07 11:06:46

1. 代码中inqueue声明是 inQueue

2. 感觉有点类似 暴力搜索判断,起始点和截止点的之间的所有结点都考察一遍

- slvher 2019-01-07 09:56:43

机器翻译的例子，用于解码的启发式剪枝是 beam search 算法吧？在 NLP 领域序列解码场合有广泛应用，不保证最优解，但通过调整 beam width 参数能得到工程上可接受的结果



44|最短路径：地图软件是如何计算出最优出行路径的？

作者回复2019-01-17 15:08:38

beam search算法不怎么懂，我抽空研究下：)

不过，我的那个算法是可以得到最优解的。

- P@tricK 2019-01-07 09:25:14

课后思考：

1. 想到一个简单粗暴的，用道路长度/限速作为权重。（实际上算上交通拥堵的话感觉复杂好多，需要去根据当前交通状况判断……）

2. 分别取离起点和终点最近的地铁站为s和t，地铁每个站权重为1（也就是无权），类似Dijkstra算法，从s开始搜索，维护一个优先级队列，遇到换乘点则添加分支。（公交车的如果也用这个想法，每个站都会是换乘点，而且每个换乘点有n个线路…… 会想用上分区的办法）

- 魏全运 2019-01-07 09:00:26

vertex compareTo有问题吧，怎么没有相等的分支呀？

作者回复2019-01-07 09:39:14

有也可以 没有也可以的