15讲从树到图：如何让计算机学会看地图

你好，我是黄申。

我们经常使用手机上的地图导航App，查找出行的路线。那计算机是如何在多个选择中找到最优解呢？换句话说，计算机是如何挑选出最佳路线的呢？

前几节，我们讲了数学中非常重要的图论中的概念，图，尤其是树中的广度优先搜索。在广度优先的策略中，因为社交网络中的关系是双向的，所以我们直接用无向边来求解图中任意两点的最短通路。

这里，我们依旧可以用图来解决这个问题，但是，影响到达最终目的地的因素有很多，比如出行的交通工具、行驶的距离、每条道路的交通状况等等，因此，我们需要赋予到达目的地的每条边不同的权重。而我们想求的最佳路线，其实就是各边权重之和最小的通路。

我们前面说了，广度优先搜索只测量通路的长度，而不考虑每条边上的权重。那么广度优先搜索就无法高效地完成这个任务了。那我们能否把它改造或者优化一下呢？

我们需要先把交通地图转为图的模型。图中的每个结点表示一个地点，每条边表示一条道路或者交通工具的路线。其中，边是有向的，表示单行道等情况。其次，边是有权重的。

假设你关心的是路上所花费的时间，那么权重就是从一点到另一点所花费的时间；如果你关心的是距离，那么权重就是两点之间的物理距离。这样，我们就把交通导航转换成图论中的一个问题：在边有权重的图中，如何让计算机查找最优通路？

基于广度优先或深度优先搜索的方法

我们以寻找耗时最短的路线为例来看看。

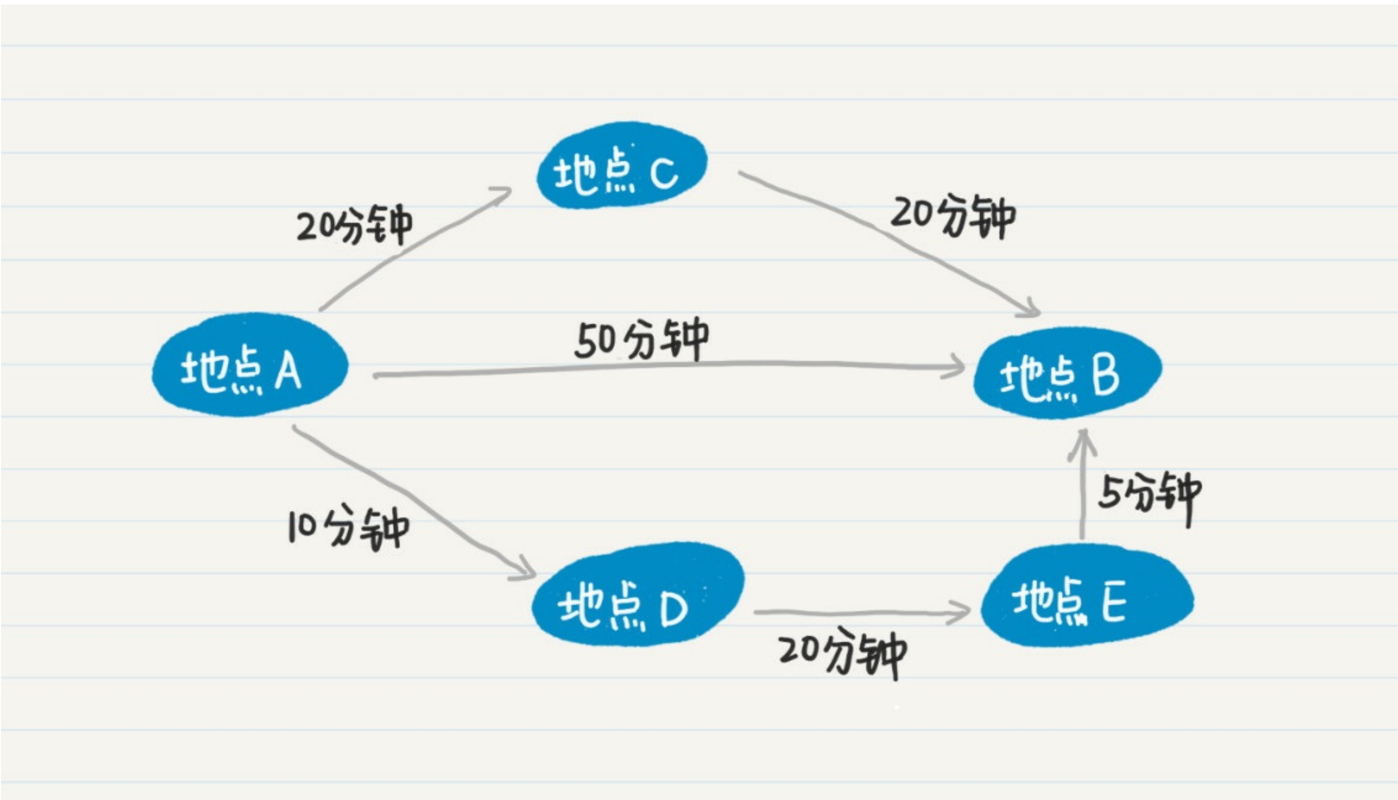
一旦我们把地图转换成了图的模型，就可以运用广度优先搜索，计算从某个出发点，到图中任意一个其他结点的总耗时。基本思路是，从出发点开始，广度优先遍历每个点，当遍历到某个点的时候，如果该点还没有耗时的记录，记下当前这条通路的耗时。如果该点之前已经有耗时记录了，那就比较当前这条通路的耗时是不是比之前少。如果是，那就用当前的替换掉之前的记

录。

实际上，地图导航和之前社交网络最大的不同在于，每个结点被访问了一次还是多次。在之前的社交网络的案例中，使用广度优先策略时，对每个结点的首次访问就能获得最短通路，因此每个结点只需要被访问一次，这也是为什么广度优先比深度优先更有效。

而在地图导航的案例中，从出发点到某个目的地结点，可能有不同的通路，也就意味着耗时不同。而耗时是通路上每条边的权重决定的，而不是通路的长度。因此，为了获取达到某个点的最短时间，我们必须遍历所有可能的路线，来取得最小值。这也就是说，我们对某些结点的访问可能有多次。

我画了一张图，方便你理解多条通路对最终结果的影响。这张图中有A、B、C、D、E五个结点，分别表示不同的地点。



从这个图中可以看出，从A点出发到到目的地B点，一共有三条路线。如果你直接从A点到B点，度数为1，需要50分钟。从A点到C点再到B点，虽然度数为2，但总共只要40分钟。从A点到D点，到E点，再到最后的B点，虽然度数为3，但是总耗时只有35分钟，比其他所有的路线更优。这种情形之下，使用广度优先找到的最短通路，不一定是最优的路线。所以，对于在地图上查找最优路线的问题，无论是广度优先还是深度优先的策略，都需要遍历所有可能的路线，然后取最优的解。

在遍历所有可能的路线时，有几个问题需要注意。

第一，由于要遍历所有可能的通路，因此一个点可能会被访问多次。当然，这个"多次"是指某个结点出现在不同通路中，而不是多次出现在同一条通路中。因为我们不想让用户总是兜圈子，所以需要避免回路。

第二，如果某个结点x和起始点s之间存在多个通路，每当x到s之间的最优路线被更新之后，我们还需要更新所有和x相邻的结点之最优路线，计算复杂度会很高。

一个优化的版本：Dijkstra算法

无论是广度优先还是深度优先的实现，算法对每个结点的访问都可能多于一次。而访问多次，就意味着要消耗更多的计算机资源。那么，有没有可能在保证最终结果是正确的情况下，尽可能地减少访问结点的次数，来提升算法的效率呢？

首先，我们思考一下，对于某些结点，是不是可以提前获得到达它们的最终的解（例如最短耗时、最短距离、最低价格等等），从而把它们提前移出遍历的清单？如果有，是哪些结点呢？什么时候可以把它们移出呢？Dijkstra算法要登场了！它简直就是为了解决这些问题量身定制的。

Dijkstra算法的核心思想是，对于某个结点，如果我们已经发现了最优的通路，那么就无需在将来的步骤中，再次考虑这个结点。Dijkstra算法很巧妙地找到这种点，而且能确保已经为它找到了最优路径。

1.Dijkstra算法的主要步骤

让我们先来看看Dijkstra算法的主要步骤，然后再来理解，它究竟是如何确定哪些结点已经拥有了最优解。

首先你需要了解几个符号。

第一个是source，我们用它表示图中的起始点，缩写是s。

然后是weight，表示二维数组，保存了任意边的权重，缩写为w。w[m, n]表示从结点m到结点n的有向边之权重，大于等于0。如果m到n有多条边，而且权重各自不同，那么取权重最小的那条边。

接下来是min_weight，表示一维数组，保存了从s到任意结点的最小权重，缩写为mw。假设从s到某个结点m有多条通路，而每条通路的权重是这条通路上所有边的权重之和，那么mw[m]就表示这些通路权重中的最小值。mw[s]=0，表示起始点到自己的最小权重为0。

最后是Finish，表示已经找到最小权重的结点之集合，缩写为F。一旦结点被放入集合F，这个结点就不再参与将来的计算。

初始的时候，Dijkstra算法会做三件事情。第一，把起始点s的最小权重赋为0，也就是mw[s] = 0。第二，往集合F里添加结点s，F包含且仅包含s。第三，假设结点s能直接到达的边集合为M，对于其中的每一条边m，则把mw[m]设为w[s, m]，同时对于所有其他s不能直接到达的结点，将通路的权重设为无穷大。

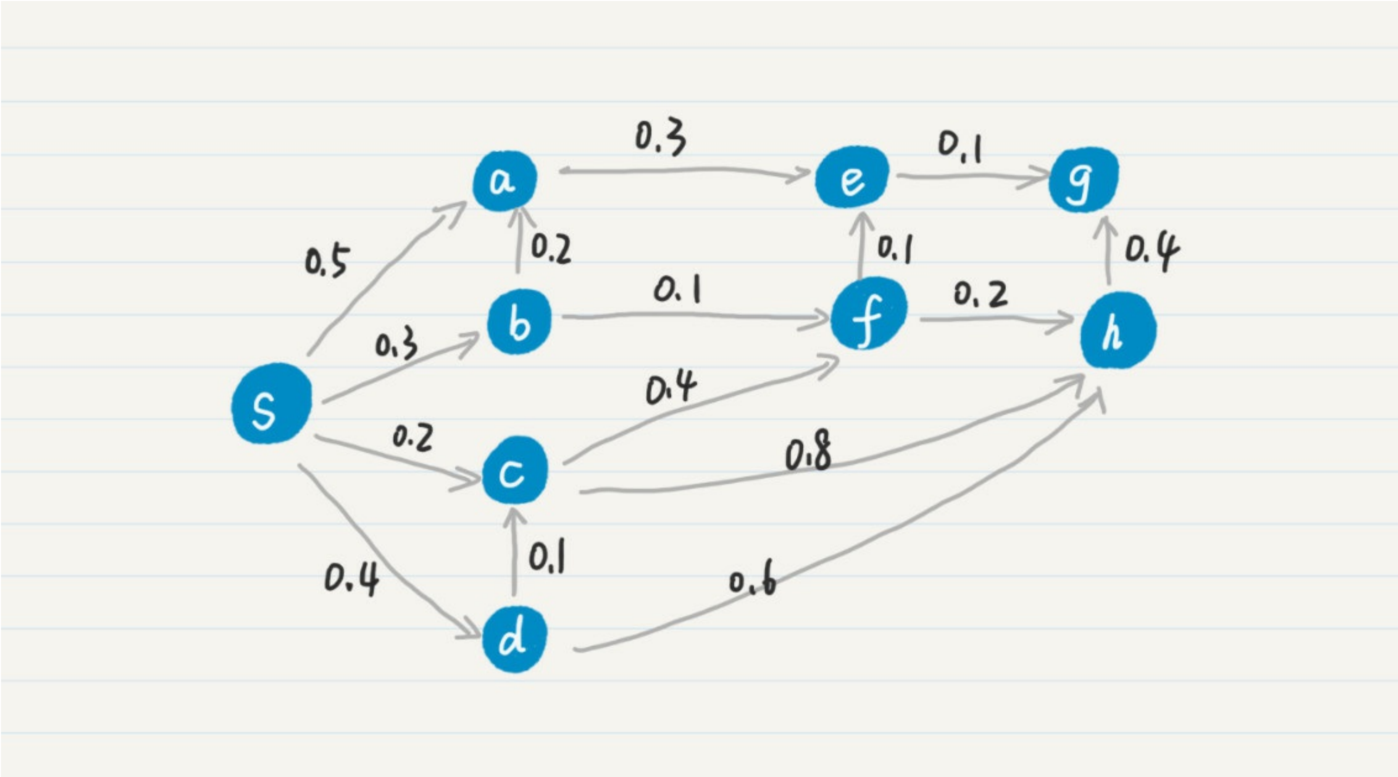
然后，Dijkstra算法会重复下列两个步骤。

第一步，查找最小mw。从mw数组选择最小值，则这个值就是起始点s到所对应的结点的最小权重，并且把这个点加入到F中，针对这个点的计算就算完成了。比如，当前mw中最小的值是mw[x]=10，那么结点s到结点x的最小权重就是10，并且把结点x放入集合F，将来没有必要再考虑点x，mw[x]可能的最小值也就确定为10了。

第二步，更新权重。然后，我们看看，新加入F的结点x，是不是可以直接到达其他结点。如果是，看看通过x到达其他点的通路权重，是否比这些点当前的mw更小，如果是，那么就替换这些点在mw中的值。例如，x可以直接到达y，那么把(mw[x] + w[x, y])和mw[y]比较，如果(mw[x] + w[x, y])的值更小，那么把mw[y]更新为这个更小的值，而我们把x称为y的前驱结点。

然后，重复上述两步，再次从mw中找出最小值，此时要求mw对应的结点不属于F，重复上述动作，直到集合F包含了图的所有结点，也就是说，没有结点需要处理了。

字面描述有些抽象，我用一个具体的例子来解释一下。你可以看我画的这个图。



我们把结点s放入集合F。同s直接相连的结点有a、b、c和d，我把它们的mw更新为w数组中的值，就可以得到如下结果：

步骤	F	mw	已确定的mw
1	s	mw[a]=0.5 mw[b]=0.3 mw[c]=0.2 mw[d]=0.4 其他mw=∞	空集

然后，我们从mw选出最小的值0.2，把对应的结点c加入集合F，并更新和c直接相连的结点f、h的mw值，得到如下结果：

步骤	F	mw	已确定的mw
1	s	mw[a]=0.5 mw[b]=0.3 mw[c]=0.2 mw[d]=0.4 其他mw=∞	空集
2	s, c	mw[a]=0.5 mw[b]=0.3 mw[d]=0.4 mw[f]=0.6 mw[h]=1.0 其他mw=∞	mw[c]=0.2

然后，我们从mw选出最小的值0.3，把对应的结点b加入集合F，并更新和b直接相连的结点a和f的mw值。以此逐步类推，可以得到如下的最终结果：

步骤	F	mw	已确定的mw
1	s	mw[a]=0.5 mw[b]=0.3 mw[c]=0.2 mw[d]=0.4 其他mw=∞	空集
2	s c	mw[a]=0.5 mw[b]=0.3 mw[d]=0.4 mw[f]=0.6 mw[h]=1.0 其他mw=∞	mw[c]=0.2
3	s c b	mw[a]=0.5 mw[d]=0.4 mw[f]=0.4 mw[h]=1.0 其他mw=∞	mw[c]=0.2 mw[b]=0.3
4	s c b d	mw[a]=0.5 mw[f]=0.4 mw[h]=1.0 其他mw=∞	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4
5	s c b d f	mw[a]=0.5 mw[h]=0.6 mw[e]=0.5 其他mw=∞	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4 mw[f]=0.4
6	s c b d f a	mw[h]=0.6 mw[e]=0.5 其他mw=∞	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4 mw[f]=0.4 mw[a]=0.5
7	s c b d f a e	mw[h]=0.6 mw[g]=0.6	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4 mw[f]=0.4 mw[a]=0.5 mw[e]=0.5
8	s c b d f a e h	mw[g]=0.6	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4 mw[f]=0.4 mw[a]=0.5 mw[e]=0.5 mw[h]=0.6
9	s c b d f a e h g	空集	mw[c]=0.2 mw[b]=0.3 mw[d]=0.4 mw[f]=0.4 mw[a]=0.5 mw[e]=0.5 mw[h]=0.6 mw[g]=0.6

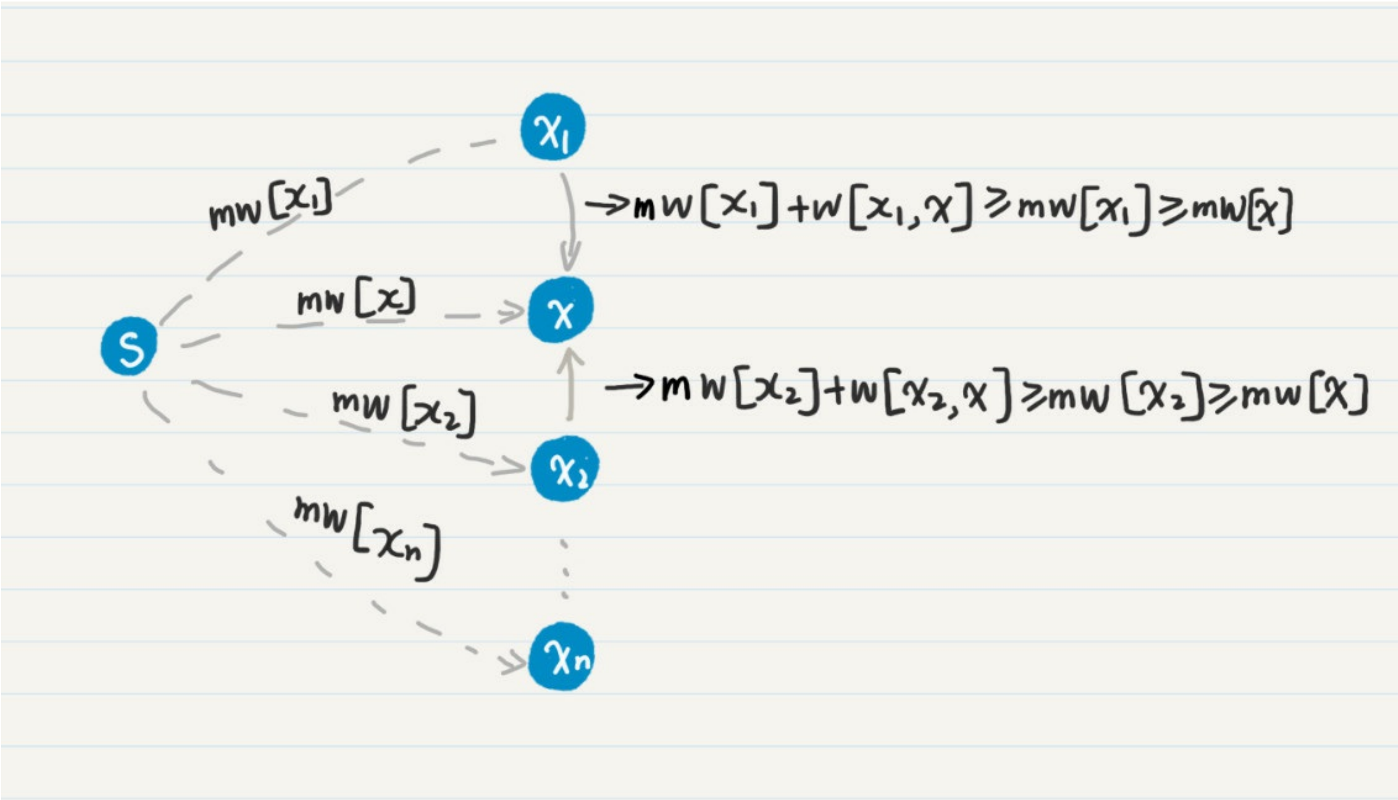
你可以试着自己从头到尾推导一下，看看结果是不是和我的一致。

说到这里，你可能会产生一个疑问：Dijkstra算法提前把一些结点排除在计算之外，而且没有遍历全部可能的路径，那么它是如何确保找到最优路径的呢？下面，我们就来看看这个问题的答案。Dijkstra算法的步骤看上去有点复杂，不过其中最关键的两步是：第一个是每次选择最小的mw；第二个是，假设被选中的最小mw，所对应的结点是x，那么查看和x直接相连的结点，并更新它们的mw。

2.为什么每次都要选择最小的mw?

最小的、非无穷大的mw值，对应的结点是还没有加入F集合的、且和s有通路的那些结点。假设当前mw数组中最小的值是mw[x]，对应的结点是x。如果边的权重都是正值，那么通路上的权重之和是单调递增的，所以其他通路的权重之和一定大于当前的mw[x]，因此即使存在其他的通路，其权重也会比mw[x]大。

你可以结合这个图，来理解我刚才这段话。



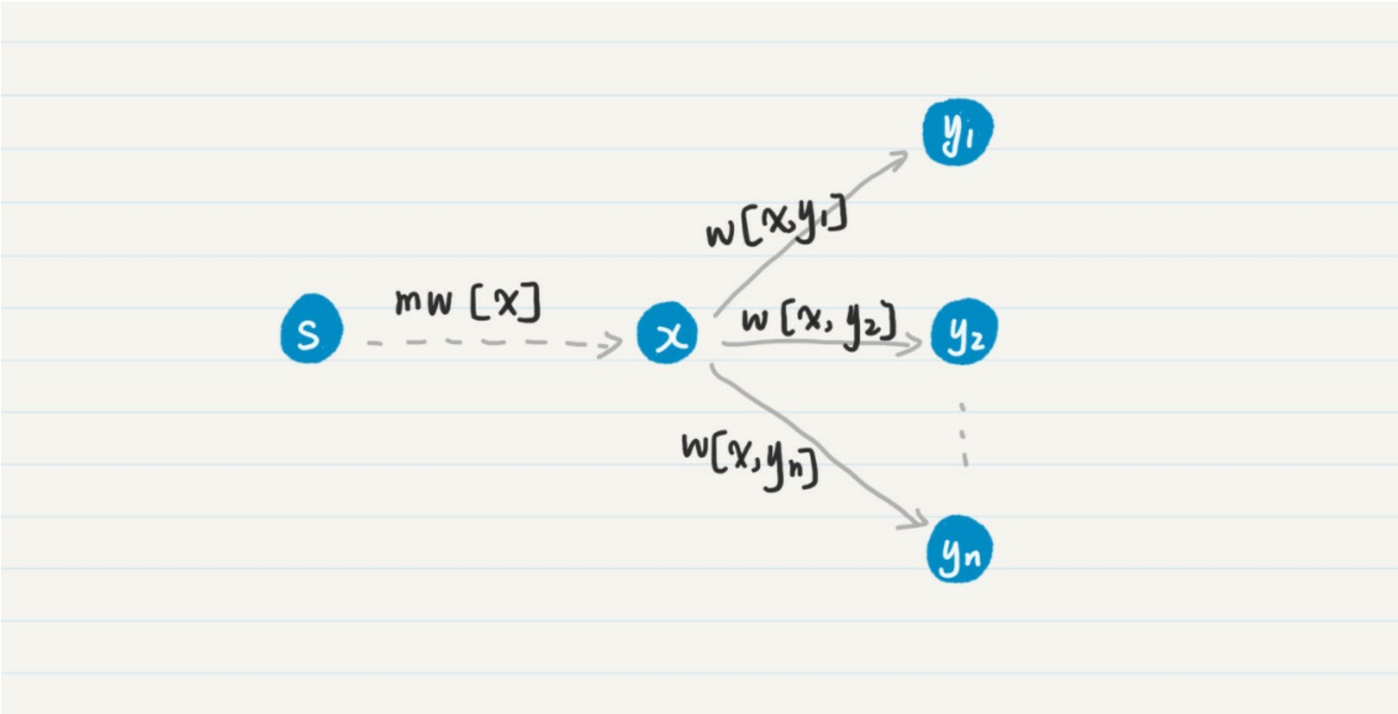
图中的虚线表示省去了通路中间的若干结点。mw[x]是当前mw数组中的最小值，所以它小于等于任何一个mw[xn]，其中xn不等于x。

我们假设存在另一个通路，通过x_{n}达到x，那么通路的权重总和为mw[x_{n}] + w[x_{n}, x] ≥ mw[x_{n}] ≥ mw[x]。所以我们可以得到一个结论：拥有最小mw值的结点x不可能再找到更小的mw值，可以把它放入“已完成“的集合F。

这就是为什么每次都要选择最小的mw值，并认为对应的结点已经完成了计算。和广度优先或者深度优先的搜索相比，Dijkstra算法可以避免对某些结点，重复而且无效的访问。因此，每次选择最小的mw，就可以提升了搜索的效率。

3.为什么每次都要看x直接相连的结点?

我们已经确定mw[x]是从点s到点x的最小权重，那么就可以把这个确定的值传播到和x直接相连、而且不在F中的结点。通过这一步，我们就可以获得从点s到这些点、而且经过x的通路中最小的那个权重。我画了张图帮助你理解。



在这个图中，x直接相连y_{1}，y_{2}，…，y_{n}。从点s到点x的mw[x]已经确定了，那么对于从s到yn的所有通路，只有两种可能，经过x和不经过x。如果这条通路经过x，那么其权重的最小值就是mw'[y_{i}] = mw[x] + w[x, y_{i}]中的一个（1≤i≤n），我们只需要把这个值和其他未经过x结点的通路之权重对比就足够了。这就是为什么每次要更新和x直接相连的结点之mw。

这一步和广度优先策略中的查找某个结点的所有相邻结点类似。但是，之后，Dijkstra算法重复挑选最小权重的步骤，既没有遵从广度优先，也没有遵从深度优先。即便如此，它仍然保证了不会遗漏任意一点和起始点s之间、拥有最小权重的通路，从而保证了搜索的覆盖率。你可能会奇怪，这是如何得到保证的？我使用数学归纳法，来证明一下。

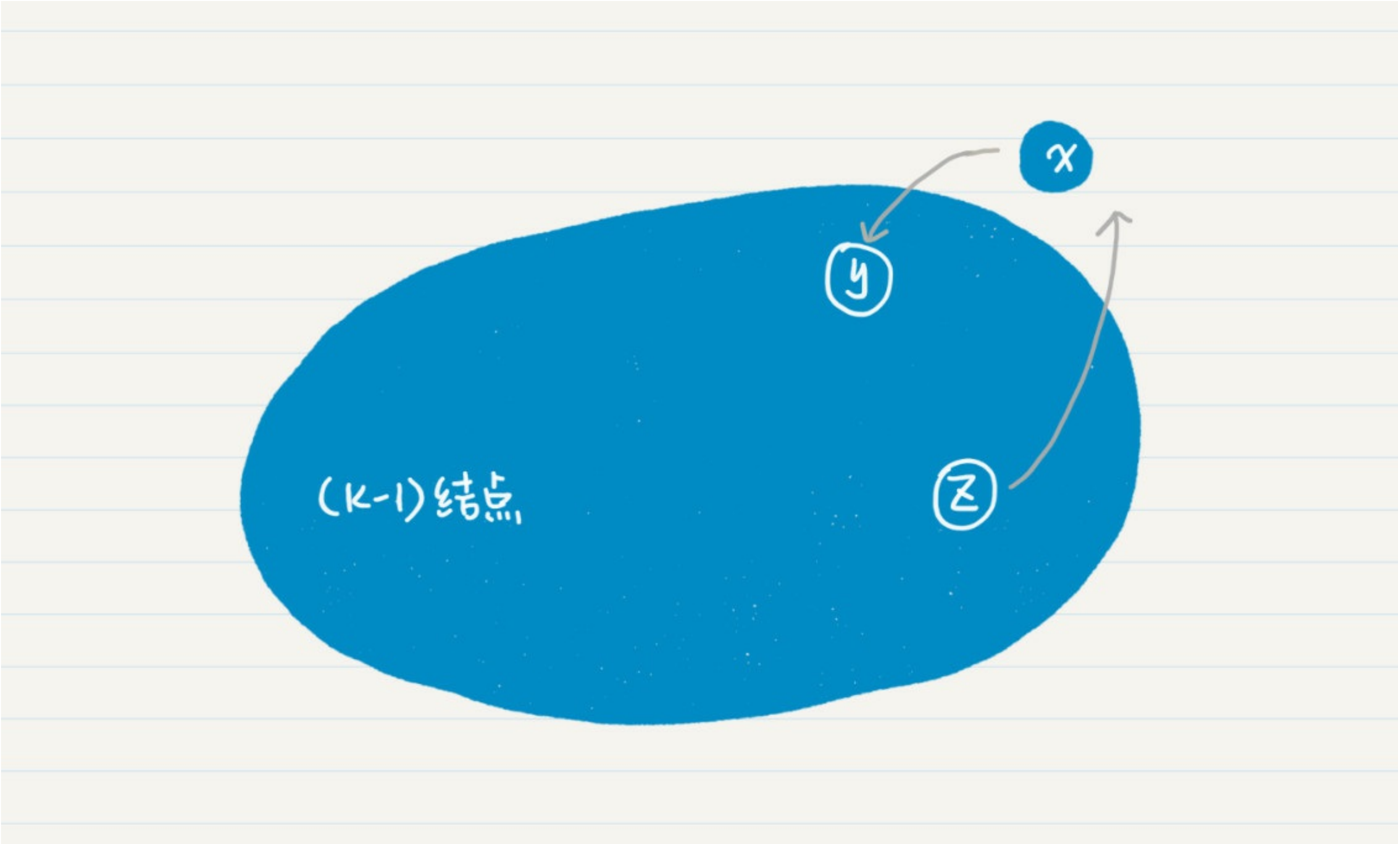
你还记得数学归纳法的一般步骤吗？刚好借由这个例子我们也来复习一下。

我们的命题是，对于任意一个点，Dijkstra算法都可以找到它和起始点s之间拥有最小权重的通路。

首先，当n=1的时候，也就是只有起始点s和另一个终止点的时候，Dijkstra算法的初始化阶段的第3步，保证了命题的成立。

然后，我们假设n=k-1的时候命题成立，同时需要证明n=k的时候命题也成立。命题在n=k-1时成立，表明从点s到k-1个终点的任何一个时，Dijkstra算法都能找到拥有最小权重的通路。那么再增加一个结点x，Dijkstra算法同样可以为包含x的k个终点找到最小权重通路。

这里我们只需要考虑x和这k-1个点连通的情况。因为如果不连通，就没有必要考虑x了。既然连通，x可能会指向之前k-1个结点，也有可能被这k-1个结点所指向。假设x指向了y，而z指向了x，y和z都是之前k-1个结点中的一员。



我们先来看x对y的影响。如果x不在从s到y的最小权重通路上，那么x的加入并不影响mw[y]的最终结果。如果x在从s到y的最小权重通路上，那么就意味着mw[x] + w[x, y]≤mw'[y]，mw'表示没有引入结点x的时候，mw的值。所以有mw[x]≤mw'[y]，这就意味着Dijkstra算法在查找最小mw的步骤中，会在mw'[y]之前挑出mw[x]，也就是找到了从s到y，且经过x的最小权重通路。

我们再来看z对x的影响。假设有多个z指向x，分别是z_{1}, z_{2}, …,z_{m}，从s到x的通路必定会经过这m个z结点中的一个。Dijkstra算法中找最小mw的步骤，一定会遍历mw[z_{i}]（1<=i<=m），而更新权重的步骤，可以并保证从(mw[z_{i}] + w[z_{i}, x])中找出最小值，最终找到从s到x的最优通路。

有了详细的推导，想要写出代码就不难了。我这里只给你说几点需要注意的地方。

在自动生成图的函数中，你需要把广度优先搜索的相应代码做两处修改。第一，现在边是有向的了，所以生成的边只需要添加一次；第二，要给边赋予一个权重值，例如可以把边的权重设置为[0,1.0)之间的float型数值。

为了更好地模块化，你可以实现两个函数：findGeoWithMinWeight和updateWeight。它们分别对应于我之前提到的最重要的两步：每次选择最小的mw；更新和x直接相连的结点之mw。

每次查找最小mw的时候，我们需要跳过已经完成的结点，只考虑那些不在F集合中的点。这也是Dijkstra算法比较高效的原因。此外，如果你想输出最优路径上的每个结点，那么在updateWeight函数中就要记录每个结点的前驱结点。

如果你能跟着我进行一步步的推导，并且手写代码进行练习，相信你对Dijkstra算法会有更深刻的印象。

小结

我们使用Dijkstra算法来查找地图中两点之间的最短路径，而今天我所介绍的Dijkstra使用了更为抽象的“权重”。如果我们把结点作为地理位置，边的权重设置为路上所花费的时间，那么Dijkstra算法就能帮助我们找到，任意两个点之间耗时最短的路线。

除了时间之外，你也可以对图的边设置其他类型的权重，比如距离、价格，这样Dijkstra算法可以让用户找到地图任意两点之

间的最短路线，或者出行的最低价格等等。有的时候，边的权重越大越好，比如观光车开过某条路线的车票收入。对于这种情况，Dijkstra算法就需要调整一下，每次找到最大的mw，更新邻近结点时也要找更大的值。所以，你只要掌握核心的思路就可以了，具体的实现可以根据情况去灵活调整。

思考题

今天的思考题和地图数据的特殊情况有关。

1. 如果边的权重是负数，我们还能用今天讲的Dijkstra算法吗?
2. 如果地图中存在多条最优路径，也就是说多条路径的权重和都是相等的，那么我刚刚介绍的Dijkstra算法应该如何修改呢?

欢迎在留言区交作业，并写下你今天的学习笔记。你可以点击“请朋友读”，把今天的内容分享给你的好友，和他一起精进。

精选留言

Being

思考1: 如果边权值为负数就不能使用Dijkstra了，因为该算法是贪心算法，即每步都找最优解，在当前步的最优基础上找下一步最优，一定是单调递增的，而出现负权边，这样的前提就不满足了。
而且也不能有带负权值的环，这个样就会一直找当前最优，而且总是满足。
思考2:就在找最小值时返回最小值集合，更新集合内所有点的直连边权值的最小值，且把集合点都加入F。
（by the way这张封面图挺好看的 ）

2019-01-17 13:43

作者回复

回答思路很清晰，封面要感谢编辑帮忙 :)

2019-01-18 02:05

null

老师好，Dijkstra算法讲解第二步似乎有问题，你在判定C为到S最近的点后，没有更新到D的距离，D不应该是0.4而是0.3
另外思考题第一题我认为不能使用这个算法，因为最小权重并不是绝对的，有可能后面一个负数的权重，直接改变所有F集合中的值了
第二题，我认为可以同时执行判断，将两个点都加入F，同时更新两点所有直连点的w，如果有点同时链接这2个点，在做判断不知是否正确

2019-01-17 00:01

作者回复

首先，两道思考题回答得很棒，思路都是正确的。

然后，关于你说的推导问题，我又看了看原文的图，边是有向的，不过是从d到c，而不是从c到d。如果从c到d，那么就如你所说的那样。

2019-01-17 07:52

失火的夏天

Dijkstra好像是基于贪心算法的思想，因为老师用数学归纳法证明了贪心选择可以得到最优，但是出现了负数，就不满足贪心选择了，算法思路应该就变成了动态规划

2019-01-30 12:32

作者回复

Dijkstra是贪心，还是动态规划，确实有些不同意见。我个人觉得Dijkstra不算是贪心，因为贪心算法往往无法得到最优解，胜在简单和效率。而Dijkstra是可以找到最优的。我觉得Dijkstra更接近动态规划

2019-01-31 01:07

会飞的猪

```
a=node('a',{'b':0.2,'c':0.3})
b=node('b',{'d':0.2,'f':0.3})
c=node('c',{'d':0.4,'e':0.1})
d=node('d',{'e':0.3})
e=node('e',{'f':0.2})
f=node('f',{})
mw={}
lastMw={}
nolist={'a':a,'b':b,'c':c,'d':d,'e':e,'f':f}
def getLastNode(mw,lastMw):
last=min(mw, key=mw.get)
print('获取到mw最小值',last)
lastMw[last]=mw[last]

for k,v in nolist[last].son.items():
newMw=v+mw[last]
if k in mw:
if newMw<mw[k]:
mw[k]=newMw
else:
mw[k] = newMw
mw.pop(last)

if mw:
getLastNode(mw,lastMw)
return lastMw
acc=getLastNode(a.son,lastMw)
print(acc)
```

结果输出：
获取到mw最小值 b
获取到mw最小值 c
获取到mw最小值 d
获取到mw最小值 e
获取到mw最小值 f
{'b': 0.2, 'c': 0.3, 'd': 0.4, 'e': 0.4, 'f': 0.5}

2019-01-25 13:40

caohuan

黄老师 说的老长了，如果给我们讲个故事 听得会更有趣。

记得 《大话数据结构》里面有说到 广度优先和深度优先算法里，作者 用找东西的例子，广度优先是 到各个地方 比如每个房价

扫一眼，如果没有 再慢慢深入到角落，深度优先 就是 因为有个大概记忆，然后 跟随 记忆 从一个房间 比如抽屉开始 寻找，没有 再去最有可能的角落 找寻，所以 广度优先是 把所有的地方快速扫一眼，没有再慢慢进入小范围 区域，深度优先 就是去指定位置 寻找。
本篇的 Dikstra 大概可以理解：把计算好的节点放入黑箱里，有新的节点加入 只需要与 箱子 的节点连接，然后把新节点与箱子中临近的节点连接起来，计算新节点与临近节点的距离，更新最值，已有的节点间的距离不需要重复计算，总之 Dikstra算法 是没有重复的计算，所以效率会很高，总的计算量会少很多，不像深度优先算法 有大量重复的计算，广度优先算法在添加新节点 时 也会更新已有的计算。
所以Dijstra模块化的思想很节能，它包括 1.寻找MW的最小值或者最大值;2.update更新新节点时，再计算MW的最值。

回答老师的问题：问题一，权限值可以为正为负，例子：跑车游戏中，获胜方为规定时间内奔跑的路程最多，规则为 路线中有不同 奖励，其中有多增加时间的道路，也有减少跑车时间的路线，就是权限值 有正有负。
问题二：多条优先路线，照样可以运用Dijkstra算法，把 多条路线 同时与接入 新节点，然后计算距离，算出MW的值。

有个问题 请教老师：一般地图搜索场景使用Dijkstra多一点还是 动态规划多一点，还是其他算法，地图可以用 百度地图、Google地图 举例。

老师 在专栏里 会谈到 机器学习算法 在生活和产品中的运用吗?
2019-01-22 19:00

作者回复

你的建议很好，我后面会注意用更形象的方式来讲解。

至于边的权重，至少在目前的Dijkstra算法中，权重必须是正的。因为只有正的，我们才可以不去考虑已经进入F集合的结点。这个在证明过程中也提到了为什么。

一般地图搜索还是用Dijkstra偏多，当然也有一些优化的算法。

最后，我在后面两大模块讲解时，也会使用工作中实际的案例，加强学习的体验。
2019-01-23 05:35

菩提

```
// 执行测试
public static void main(String[] args) {
Node tree = init();
Map<String, Double> mw = new HashMap<>();
Map<String, Double> result_mw = new HashMap<>();

List<Node> children = tree.children;
Map<String, Double> weights = tree.weights;
for (Map.Entry<String, Double> entry : weights.entrySet()) {
mw.put(entry.getKey(), entry.getValue());
}

while (mw.size() != 0) {
String label = findGeoWithMinWeight(mw);
updateWeight(label, mw.get(label), result_mw);
Node min = getMinNode(children, label);
System.out.println("获取最小值：" + label);
List<Node> nodes = min.children;
if (nodes != null && nodes.size() > 0) {
children.addAll(nodes);
for (Node node : nodes) {
```

```
mw.put(node.label, BigDecimal.valueOf(result_mw.get(label))
.add(BigDecimal.valueOf(min.weights.get(node.label))).doubleValue());
}
}
mw.remove(label);
}

System.out.println(result_mw);
}
}
```

运行结果如下：
获取最小值：c
获取最小值：b
获取最小值：d
获取最小值：f
获取最小值：a
获取最小值：c
获取最小值：f
获取最小值：e
获取最小值：g
获取最小值：h
获取最小值：g
{a=0.5, b=0.3, c=0.2, d=0.4, e=0.5, f=0.4, g=0.6, h=0.6}
2019-01-20 23:21

作者回复

细节注意的很好，点赞
2019-01-21 02:14

菩提

```
children = new ArrayList<>();
children.add(e);
children.add(h);
weights = new HashMap<>();
weights.put("e", 0.1);
weights.put("h", 0.2);
f.children = children;
f.weights = weights;

children = new ArrayList<>();
children.add(g);
weights = new HashMap<>();
weights.put("g", 0.4);
h.children = children;
h.weights = weights;

return start;
}

children = new ArrayList<>();
children.add(e);
children.add(h);
```

```
weights = new HashMap<>();
weights.put("e", 0.1);
weights.put("h", 0.2);
f.children = children;
f.weights = weights;

children = new ArrayList<>();
children.add(g);
weights = new HashMap<>();
weights.put("g", 0.4);
h.children = children;
h.weights = weights;

return start;
}

// 获取最小权重
public static String findGeoWithMinWeight(Map<String, Double> mw) {
double min = Double.MAX_VALUE;
String label = "";
for (Map.Entry<String, Double> entry : mw.entrySet()) {
if (entry.getValue() < min) {
min = entry.getValue();
label = entry.getKey();
}
}
return label;
}

// 更新权重
public static void updateWeight(String key, Double value, Map<String, Double> result_mw) {
if (result_mw.containsKey(key)) {
if (value < result_mw.get(key)) {
result_mw.put(key, value);
}
} else {
result_mw.put(key, value);
}
}

// 获取最小节点
public static Node getMinNode(List<Node> l, String label) {
for (Node node : l) {
if (label.equals(node.label)) {
return node;
}
}
return null;
}
```

2019-01-20 23:20

菩提

1.思考题，如果权重为负数，Dijkstra算法的方式就不能用了。您在文中也提到了，每次取到最小的mw，如果后面出现负数，那前面的权重就不能保证最小了。如果存在多条最优路径，则应该加一个字段记录节点从开始到结束的轨迹。如果权重有多个最优解，则运行轨迹才是需要求解的结果，而不是权重。

2.我将您讲解的推导过程用代码实现了，为了避免小数位数计算导致的精度问题，先转为BigDecimal，再转成了double.由于留言区字数限制，我分开进行提交。

```
public class Lesson15 {

// 定义节点
static class Node {
public String label;
public List<Node> children;
public Map<String, Double> weights;

public Node(String label) {
this.label = label;
}
}

// 初始化
public static Node init() {
Node start = new Node("s");
Node a = new Node("a");
Node b = new Node("b");
Node c = new Node("c");
Node d = new Node("d");
Node e = new Node("e");
Node f = new Node("f");
Node g = new Node("g");
Node h = new Node("h");

List<Node> children = new ArrayList<>();
children.add(a);
children.add(b);
children.add(c);
children.add(d);
Map<String, Double> weights = new HashMap<>();
weights.put("a", 0.5);
weights.put("b", 0.3);
weights.put("c", 0.2);
weights.put("d", 0.4);
start.children = children;
start.weights = weights;

children = new ArrayList<>();
children.add(e);
weights = new HashMap<>();
weights.put("e", 0.3);
a.children = children;
a.weights = weights;
```

```
children = new ArrayList<>();
children.add(a);
children.add(f);
weights = new HashMap<>();
weights.put("a", 0.2);
weights.put("f", 0.1);
b.children = children;
b.weights = weights;

children = new ArrayList<>();
children.add(f);
children.add(h);
weights = new HashMap<>();
weights.put("f", 0.4);
weights.put("h", 0.8);
c.children = children;
c.weights = weights;

children = new ArrayList<>();
children.add(c);
children.add(h);
weights = new HashMap<>();
weights.put("c", 0.1);
weights.put("h", 0.6);
d.children = children;
d.weights = weights;

children = new ArrayList<>();
children.add(g);
weights = new HashMap<>();
weights.put("g", 0.1);
e.children = children;
e.weights = weights;
```

2019-01-20 23:19

strentchRise

第二张图，也就是基于距离的有向有权重的图，难道不可以用递归的分而治之来做么?
每次找出距离我最近的前方节点，这样似乎不用缓存到某节点的最小距离了吧?

2019-01-17 14:48

作者回复

我理解你说的递归分治是深度优先搜索？如果是这样，也是可以的，但是某个结点会被访问多次，效率不高。另外，当前结点的最小距离还是要缓存的，因为最终需要知道起始点到某个结点的最小距离

2019-01-18 03:22