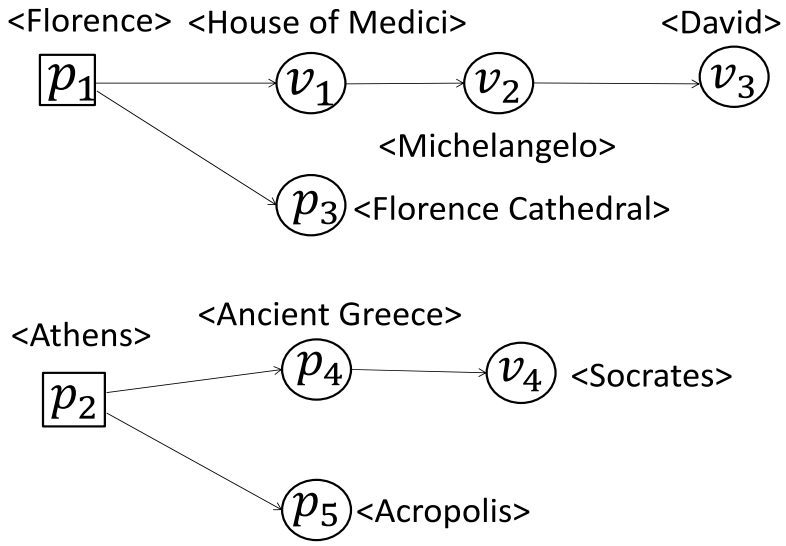
基于RDF图的语义地点skyline查询

李若龙 2018171028

RDF图

有向图，图的每个顶点包含一些词汇信息

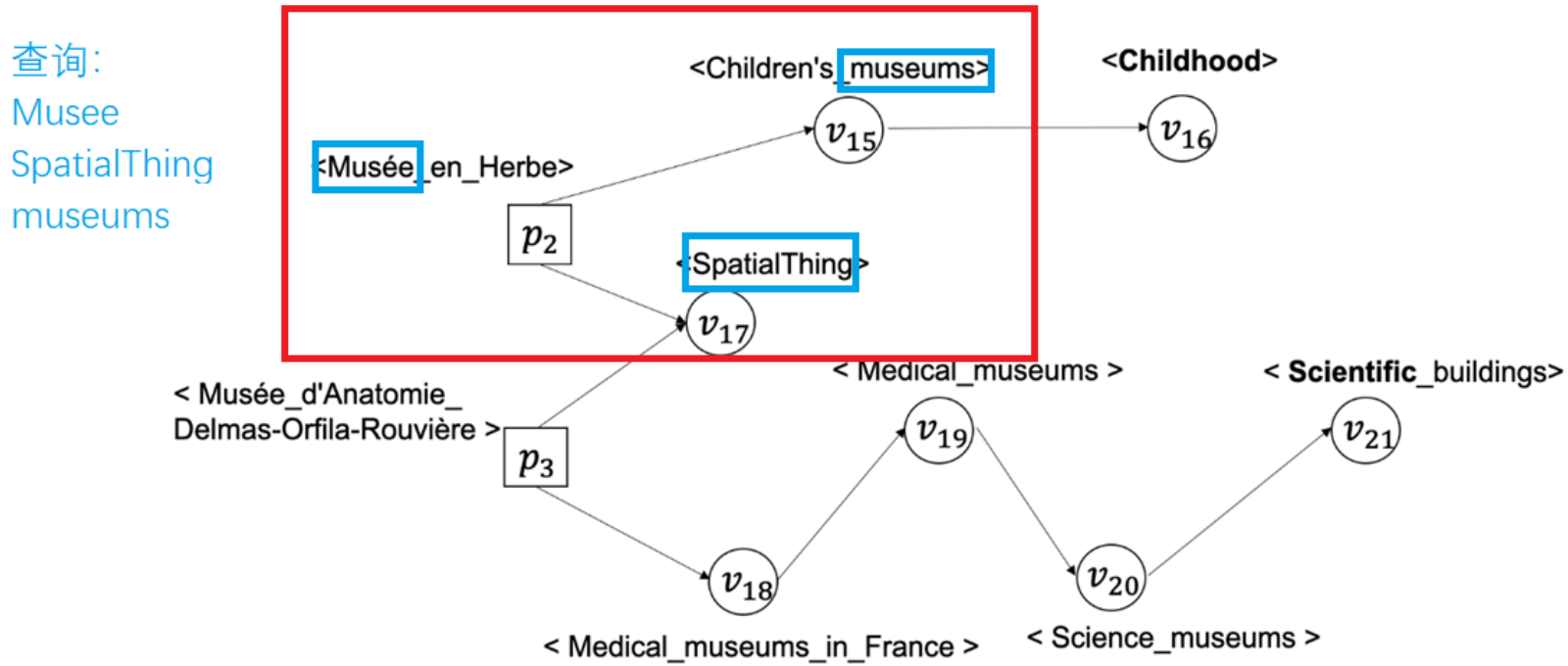


p_1:{ city, European, Italian}
v_1:{ renaissance, commerce, dynasty}
v_2:{ Italian, sculptor, painter, architect}
v_3:{ sculpture, art, history}
p_3:{ art, myth, ancient}
p_2:{ city, European, Greek, capital}
p_4:{ city-state, ancient civilization, war, Olympic}
v_4:{ philosopher, history, dialectic, knowledgeable}
p_5:{ ancient, art, myth, sculpture}

语义地点

假设现在有查询词汇集合w，w的语义地点是RDF图的一个特殊子图，具有树的结构。

W的语义地点是一颗以地点p为根的树，树的所有节点的词汇的并集，包含查询词汇集合w。



语义地点支配

如果对两个查询词汇集合w的语义地点p1和p2

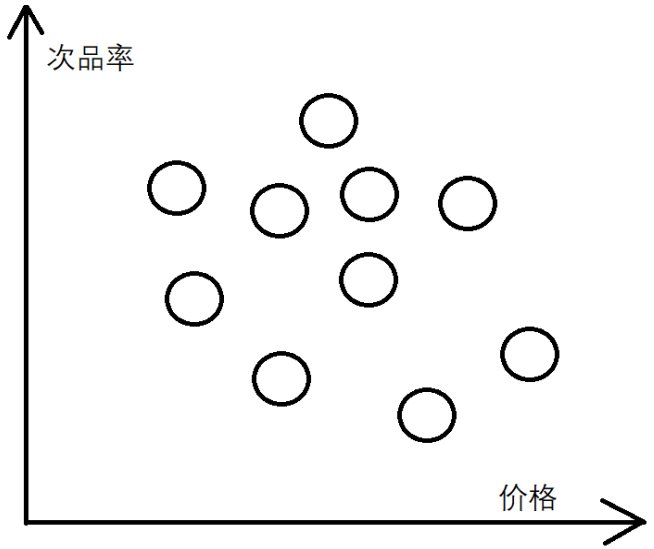
1. p1的根节点到所有词汇的最短距离都小于等于p2的根节点到所有词汇的最短距离
2. 存在某个词汇使得p1的根节点到该词汇的最短距离小于p2的根节点到该词汇的最短距离

那么语义地点p1支配p2。P1支配p2说明p1是比p2更好的选择

skyline

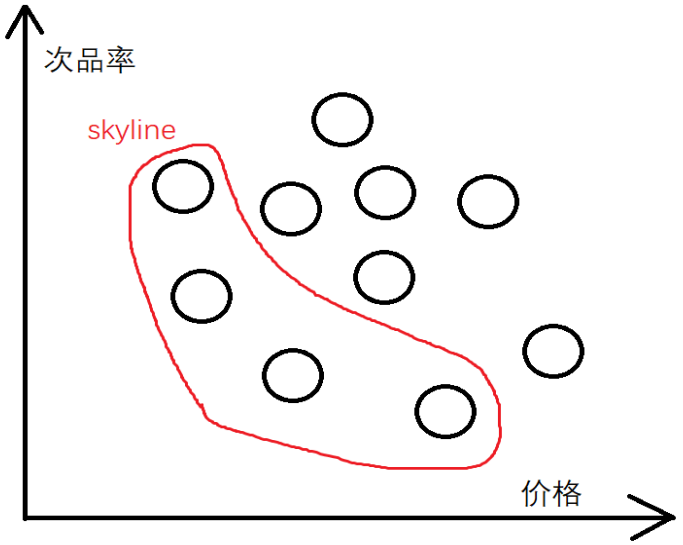
给定一组词汇w，和一个RDF图，在图中所有w的语义地点集合中，找到一个子集使得子集中的语义地点互不支配。

我们用一个“采购问题”来描述skyline查询问题：假设工厂希望采购【既便宜又次品率低】的原材料，而采购部列出以下的选择：



skyline

我们认为价格越低且次品率越低越好，但很多时候我们会面临多个不互相支配的“最优解”，即下图中红色部分圈出的四个选择。他们都代表了



1. 在同一价位下，没有任何选择比我次品率更低的了
2. 在同一次品率下，没有任何选择比我价格更低了

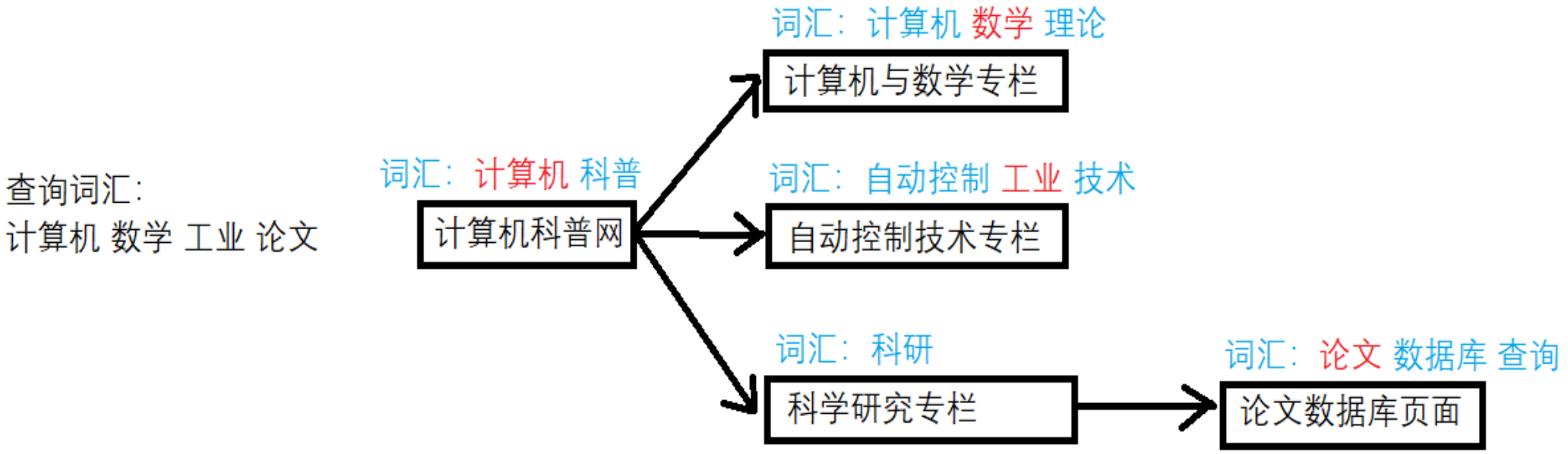
我们称这些选择为【以价格和次品率为评价标准的skyline】

同样的，RDF图中描述一个语义地点的好坏，也是通过他们到词汇的距离来决定的，我们的任务是在RDF图中找到所有以词汇距离为评价标准的语义地点中的skyline。

语义地点查询应用场景

语义地点skyline的查询可以运用在许多多目标优化的领域，比如搜索引擎的相关性查询，网页之间的超链接表示为图的边，而一个语义地点代表一个网页的主页。我们希望得到所有和我们查询的词汇最相关的网页主页。

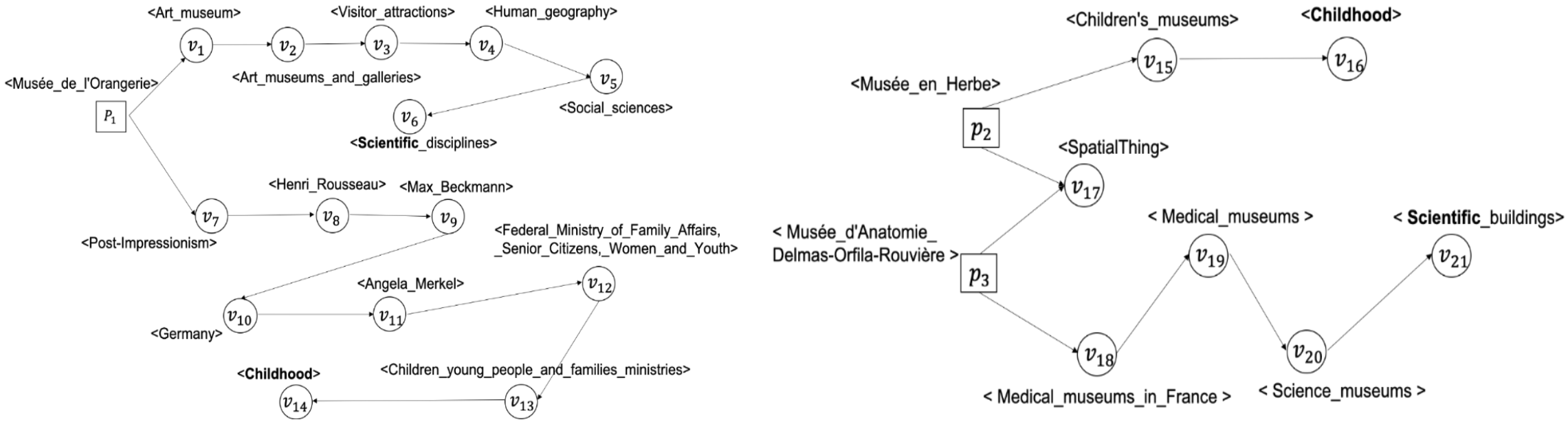
如图，我们查询的词汇是【计算机 数学 工业 论文】



Skyline查询示例

假定下图中每个顶点旁<>里面的内容是该顶点的文本属性，下划线是单词分割符

查询关键词 <childhood, scientific>，计算下图的语义地点skyline。



语义地点skyline查询问题分析

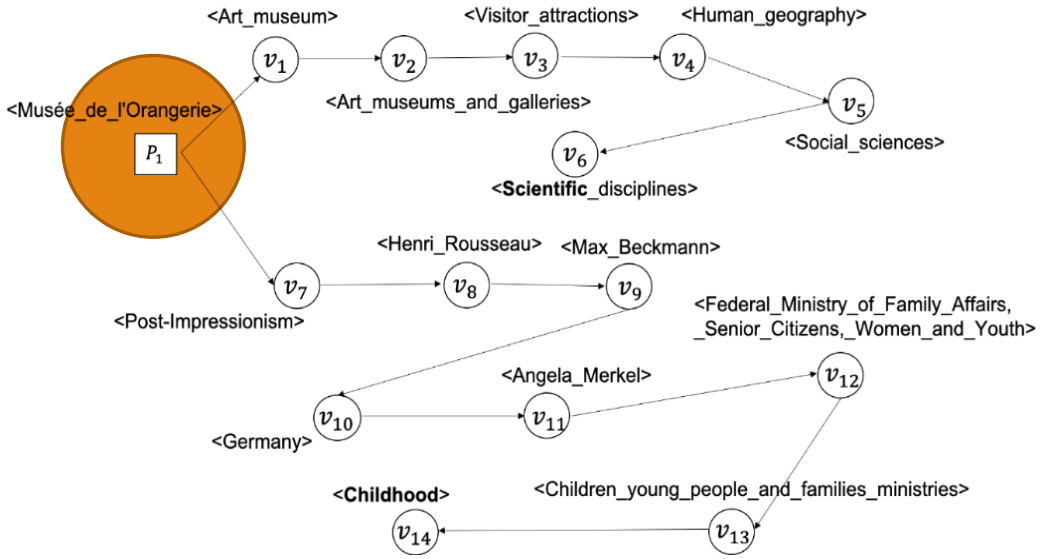
1. 查找所有符合条件的语义地点
2. 在查询出的所有语义地点中找到skyline

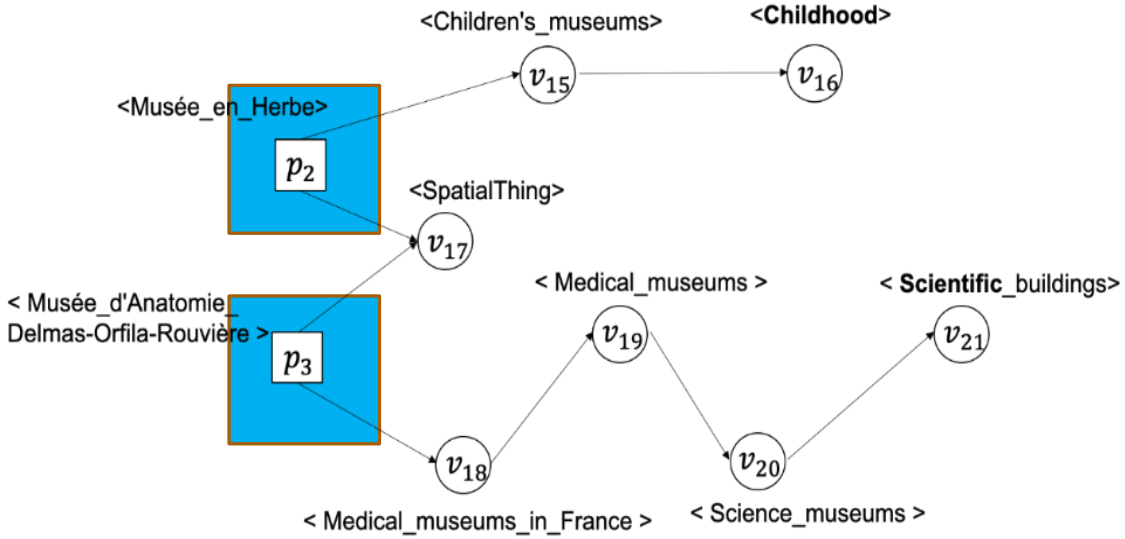
Step 1：找出所有语义地点

从p1出发到词汇scientific的最短距离是6，而到childhood的最短距离是8，所以Tp1是一个和<scientific, childhood>相关的语义地点。

从p2出发到childhood的最短距离是2，到scientific的最短距离是无穷，Tp2不是语义地点

从p3出发到childhood的最短距离是无穷，到scientific的最短距离是4，Tp3不是语义地点

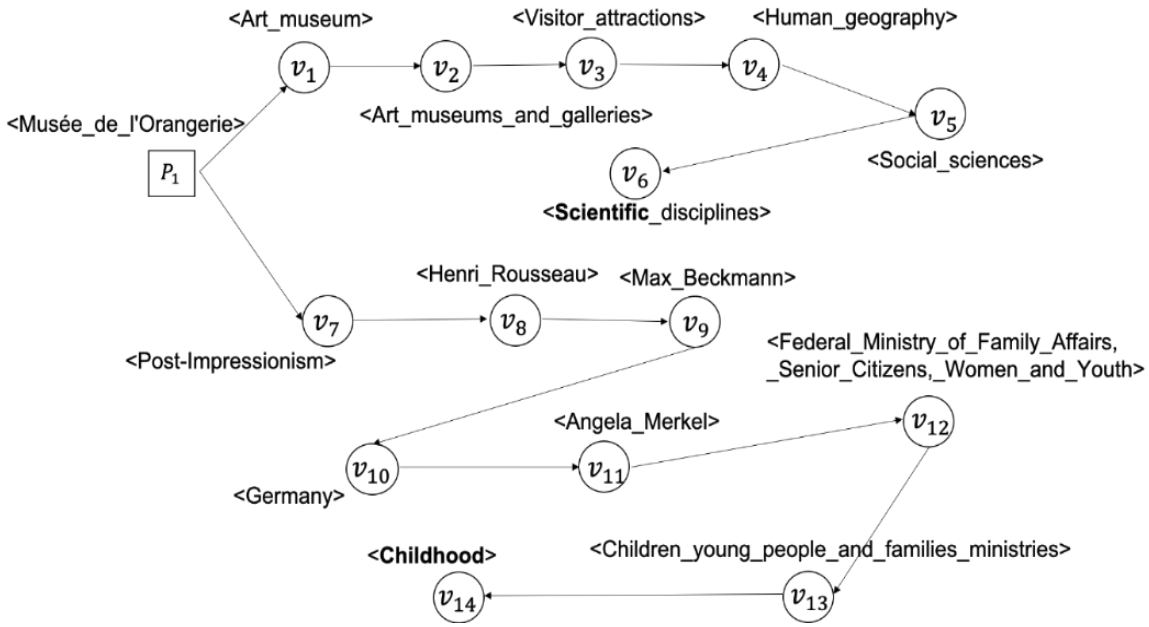




Step 2：查询skyline

因为只有一个语义地点就是Tp1，我们得到语义地点集合{Tp1}

语义地点集合中所有地点不互相支配，那么上图的语义地点skyline查询结果就是集合{Tp1}



skyline查询算法设计

假设我们现在已经拥有所有符合条件的语义地点集合p1，我们希望找出p1集合中所有语义地点skyline。

新建集合p2，枚举p1中的每一个语义地点c，对每个c都遍历p2中的所有语义地点

1.如果c不被p2中的任何语义地点支配，那么c加入p2集合

2.如果c支配任意一个p2中的语义地点，那么用c替换p2中对应的语义地点

3.循环结束，对p2中的语义地点去重，得到所有语义地点skyline集合

Skyline查询伪代码描述

```
x->y 表示x支配y

for c in p1:
    for x in p2:
        if c->x:
            x = c
        if x->c:
            break
unique(p2)  // 去重
```

Skyline查询复杂度分析

如果有n个语义地点待筛选，那么最多插入n次

而每次插入都要用O(n)遍历p2集合查看支配关系，故总体复杂度为$O(n^2)$。

因为实际语义地点数目占少数，所以该查询方法虽然是$O(n^2)$的复杂度，但是实测执行速度非常快速。

查找语义地点的算法占大部分时间，而不是查询skyline

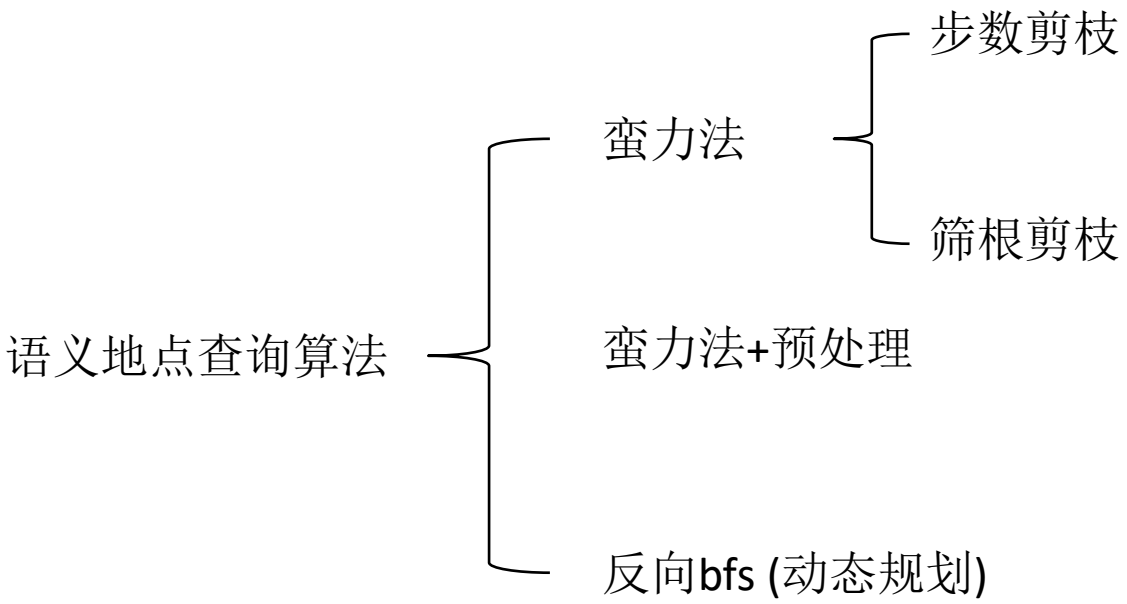
节点词汇查询

通过建立每一个节点的哈希表来作为节点词汇集合

利用哈希的特性可以在O(1)的常数时间内查询该节点是否具有某个词汇

表的建立在读取原始数据时就已经完成。

语义地点查询算法设计



蛮力法

我们根据语义地点的定义，对每个点找离其最近的查询词汇。

通过bfs广度优先搜索可以快速确定符合条件的点到源点的最短距离。

蛮力法通过枚举所有顶点作为bfs的源点，做n次bfs。

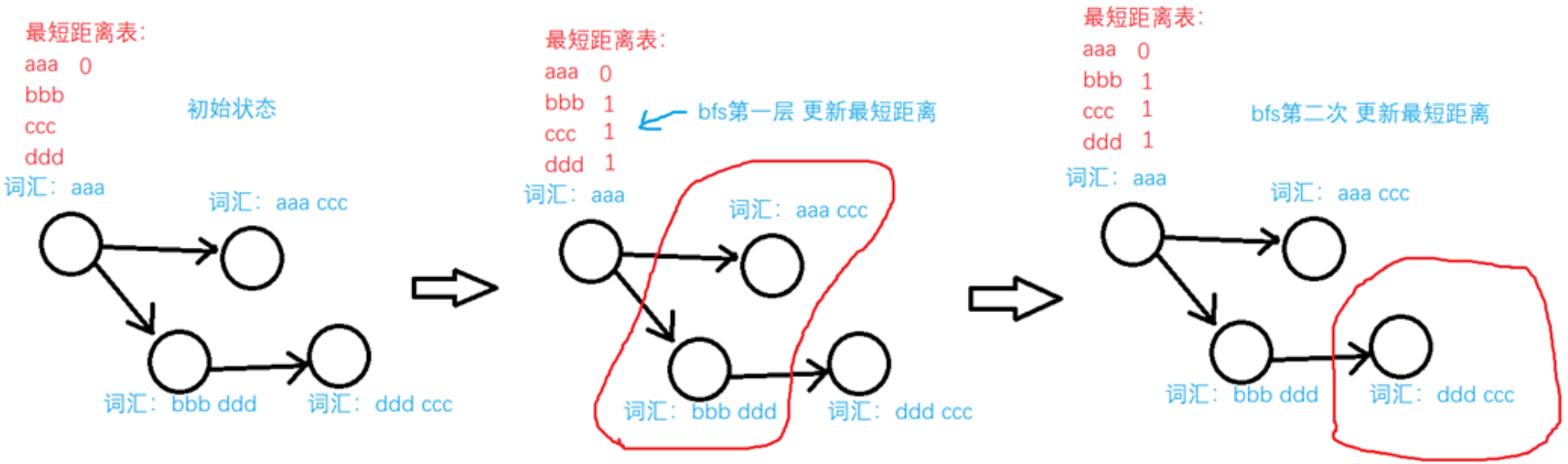
Bfs过程中查看遍历到的点是否包含查询词汇并尝试更新查询词汇到源点的最短距离。

蛮力法

下图描述了一次bfs查询最短距离的情况，以左上角的顶点为源点进行bfs。

每次bfs结束，我们检查各个词汇与源点的距离便可确定源点是否是语义地点，如果源点可达所有查询词汇则是语义地点，则加入集合p1。

事后用上文描述的skyline查询方法，在p1集合中找出skyline。



蛮力法伪代码

```
for i in n:
    c = bfs(i)  // 得到结果c
    if 如果c是语义地点:
        c加入p1集合
skyline(p1)     // skyline查询p1集合
```

```
bfs(src):
    c = {}
    queue = {src}
    step = 0
    while 队列非空:
        for x in queue:
            for w in 查询词汇:
                if x点的词汇包含w:
                    源点到w的距离 = step
            for y in x的邻居:
                queue.push(y)   // bfs下一层
        step++
    return c
```

蛮力法复杂度

枚举n个源点进行bfs共需要n次bfs

每次bfs的复杂度是O(n+e)，取上限为O(e)

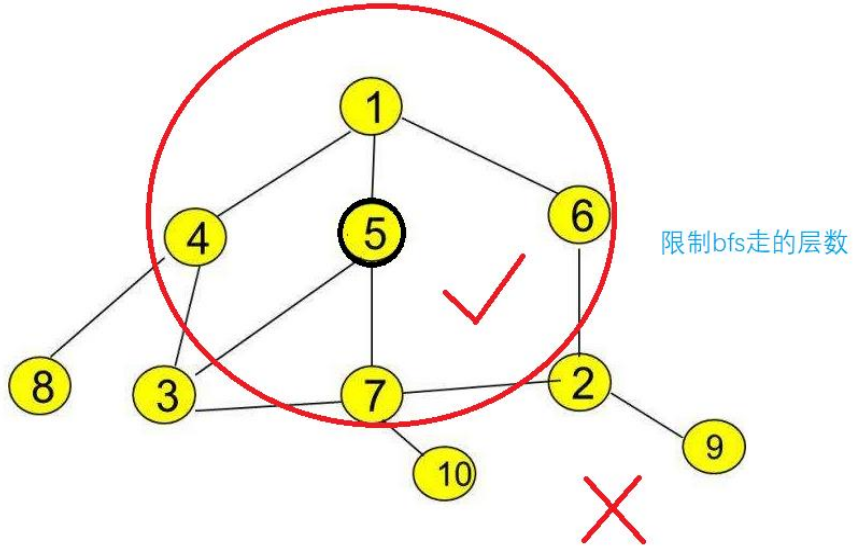
总体复杂度O(e*n)

蛮力法+步数剪枝

我们认为用户查询的词汇总是有很强的相关性，而距离根节点远端的词汇表示其相关性弱，是答案的可能性小，于是把它剪掉。

我们预设一个步数来限制bfs的搜索。bfs搜索的时候，搜索层次达到k以上就放弃搜索

假设以5为起点，步数剪枝可以用下图来表示



蛮力法+步数剪枝伪代码

```
bfs(src, k):
    c = {}
    queue = {src}
    step = 0
    while 队列非空 and step<k:
        for x in queue:
            for w in 查询词汇:
                if x点的词汇包含w:
                    源点到w的距离 = step
            for y in x的邻居:
                queue.push(y)   // bfs下一层
        step++
    return c
```

蛮力法+步数剪枝分析

因为k步剪枝使得bfs的代价变为常数代价，枚举顶点进行n次bfs，总复杂度O(n)

该剪枝策略的优点是可以加快搜索

缺点是搜索结果正确性无法保证，不能搜出所有的答案，甚至不一定有结果

因为存在输入词汇相距很远的情况。

蛮力法+筛根剪枝

我们假设语义地点根节点常常会包含某个查询词汇，没有包含任何查询词汇的节点，通常不会作为语义地点的根节点，这与我们实际生活中的认知相吻合：

我们搜索三个关键词【王老吉 收购 谷歌】，我们发现语义地点的根节点（也就是搜索结果的主页）通常会包含若干个关键词，而那些不包含关键词的节点基本不会作为语义地点的根节点出现。

蛮力法+筛根分析

我们通过对语义地点的根节点做进一步筛选，然后再对选出的根节点施加蛮力法，即只选择包含关键词的节点做源点进行bfs，从而排除部分无效的bfs。

使用蛮力法+筛根能够有效减少无效的bfs，大大加快查询速度，但是这个优化是不稳定的，其上限仍然是蛮力法的复杂度。

除此之外，和步数剪枝相仿，这个剪枝策略找到的可能不是最优解，不是所有的答案，甚至无法找到解，因为输入词汇可以任意组合，必定存在若干个语义地点，其根节点不包含查询词汇。

蛮力法+预处理（离线查询）

既然每次查询都要对所有节点做一次bfs，我们为何不在一次bfs中记录所有词汇到源点的距离，而非只记录查询词汇。

使用预处理的思想，在一次预处理中进行bfs，记录所有词汇到源点的距离并且将结果存储在n张巨大的哈希表中（每个节点都有一张），这意味着我们可以花费常数时间来查询任意节点到任意词汇的最短距离。

注：离线查询的本质还是蛮力法

离线查询bfs伪代码

和之前的bfs类似，只是检查并更新所有词汇到bfs源点的距离，而非只更新待查询词汇。

```
bfs(src):
    queue = {src}
    step = 0
    while 队列非空 and step<k:
        for x in queue:
            for w in x的所有词汇:
                源点到w的距离 = step
            for y in x的邻居:
                queue.push(y)   // bfs下一层
        step++
```

离线查询分析

因为查询任意节点到任意词汇的距离，都是常数时间，我们一次遍历所有顶点，就能够找到所有的语义地点，离线查询总体复杂度为O(n)

离线查询的优点是查询速度快，缺点是预处理的时间开销很大

预处理需要n次bfs，而每次bfs因为要遍历所有词汇，所以一次bfs复杂度为O(n+e+m)，取上界为O(m)，其中m是所有的词汇数目，所以预处理的总复杂度是O(n*m)。

除此之外，还需要巨大的空间来存储距离，空间复杂度高达O(n*m)

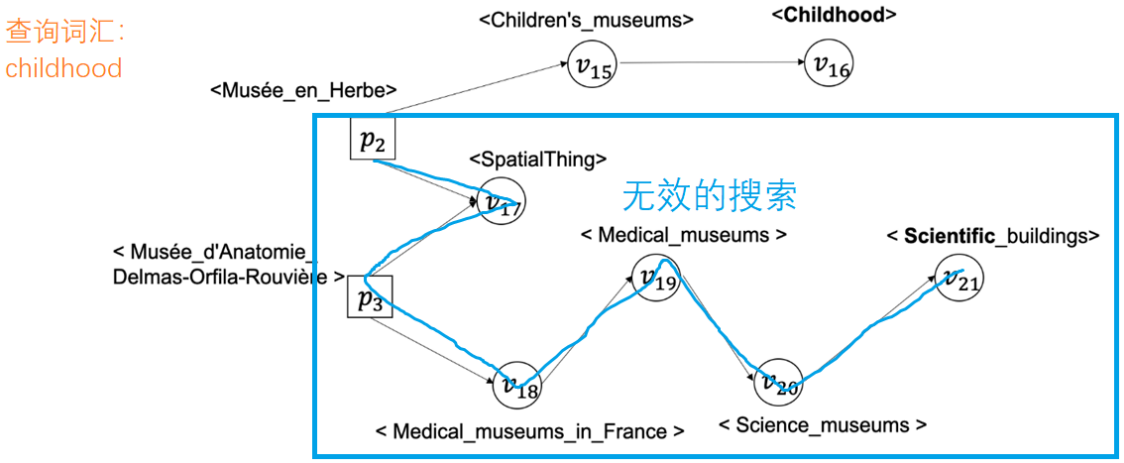
离线查询的另一个缺点是不支持图的更新，每次图的更新都要重新处理整个图。该方法适用于离线且小规模，查询频次高的数据集上。

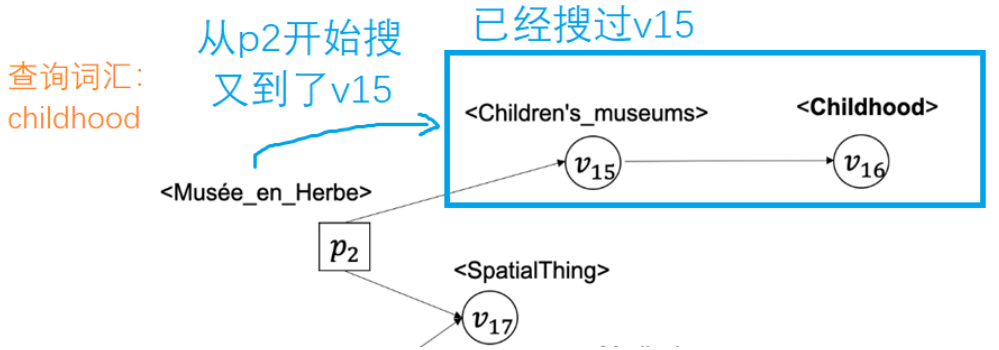
蛮力法的问题：重复与无效

在蛮力法求解语义地点的时候，我们只关注任意节点到任意查询词汇的最短距离

蛮力法枚举所有节点，计算最短距离，因为枚举大量起点，主要的时间都花在bfs上

存在很多重复或无效的bfs，所以蛮力法的时间开销相当大。

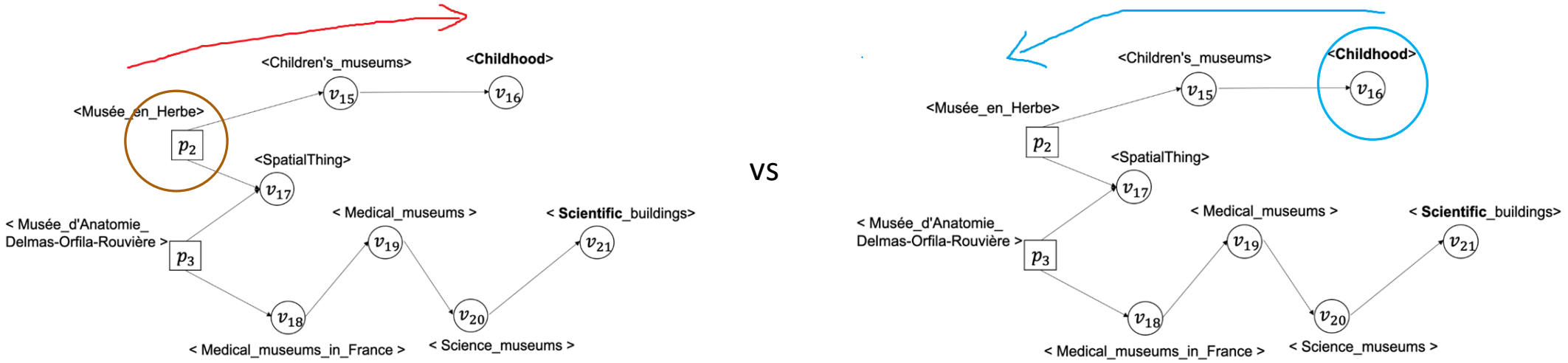




反向bfs求解语义地点

反向bfs采用逆向思维。即从拥有查询词汇的节点开始，沿着有向图的反向边进行bfs，沿路更新父节点们到词汇的距离。

避免重复或者无效的搜索，即只做有用的bfs。

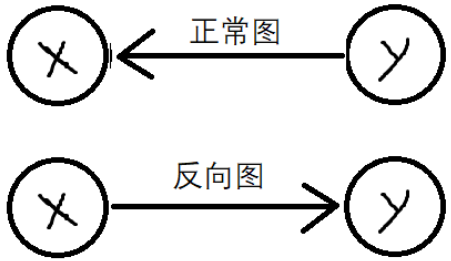


反向bfs递推式

我们定义一个数组dis，dis[x][i]描述了x点到第i个查询词汇的最短距离。

我们沿着有向图的反向边进行bfs，沿途按照如下的递推公式更新节点到词汇的距离：

$$dis[y][i] = \min(dis[y][i],\ dis[x][i] + 1)$$



其中x是反向边的起点，y是反向边的终点，即反向边 x->y。

因为在原图中y到x，需要走1步，x到词汇i需要走dis[x][i]步，故有上述递推式。

自底向上的动态规划，包含关键词的点为最小子问题（距离为0）

反向bfs伪代码描述

和蛮力法主动搜寻词汇不同，反向bfs自己主动更新其他节点到查询词汇的距离，而不是等待别人来搜索自己。

这么做大大减少了重复的bfs。这种更新方式和自底向上的动态规划异曲同工。

```
// dis[x][w] 表示x点到词汇w的最短距离
bfs_rev(src):
    queue{src}
    while 队列非空:
        x = queue.front
        queue.pop
        for y in x的逆邻接表:
            flag = false
            for w in 查询词汇:
                if dis[x][w]+1 < dis[y][w]:
                    dis[y][w] = dis[x][w]+1
                    flag = true
            if flag==true:  // 更新了就继续bfs
                q.push(y)
```

```
for i in n:
    if i节点包含某个查询词汇:
        bfs_rev(src)

p1 = {}     // 合法语义地点集合p1
for i in n:
    if i是一个语义地点:
        i插入p1集合
skyline(p1) // 在p1中找skyline
```

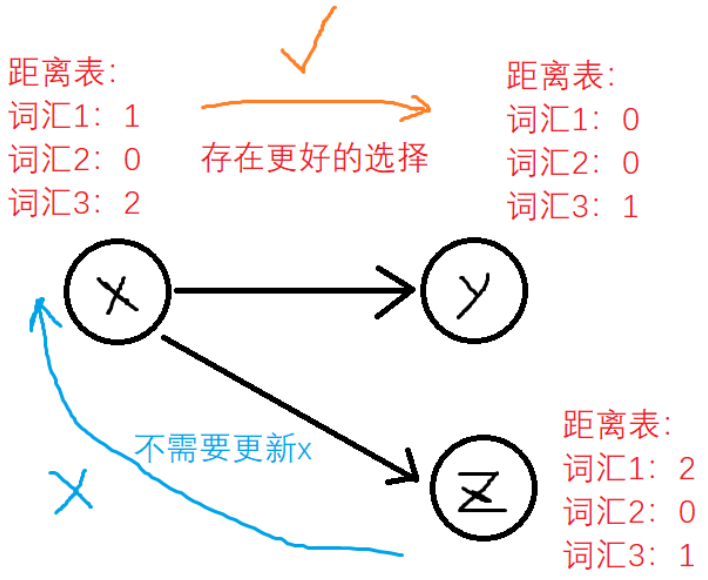
提前结束策略

如果一个节点到所有词汇的距离都不能被更新，那么不再对其bfs

因为该点及其之后的节点最短距离的计算，都不会依赖这一次的bfs带来的最短距离信息

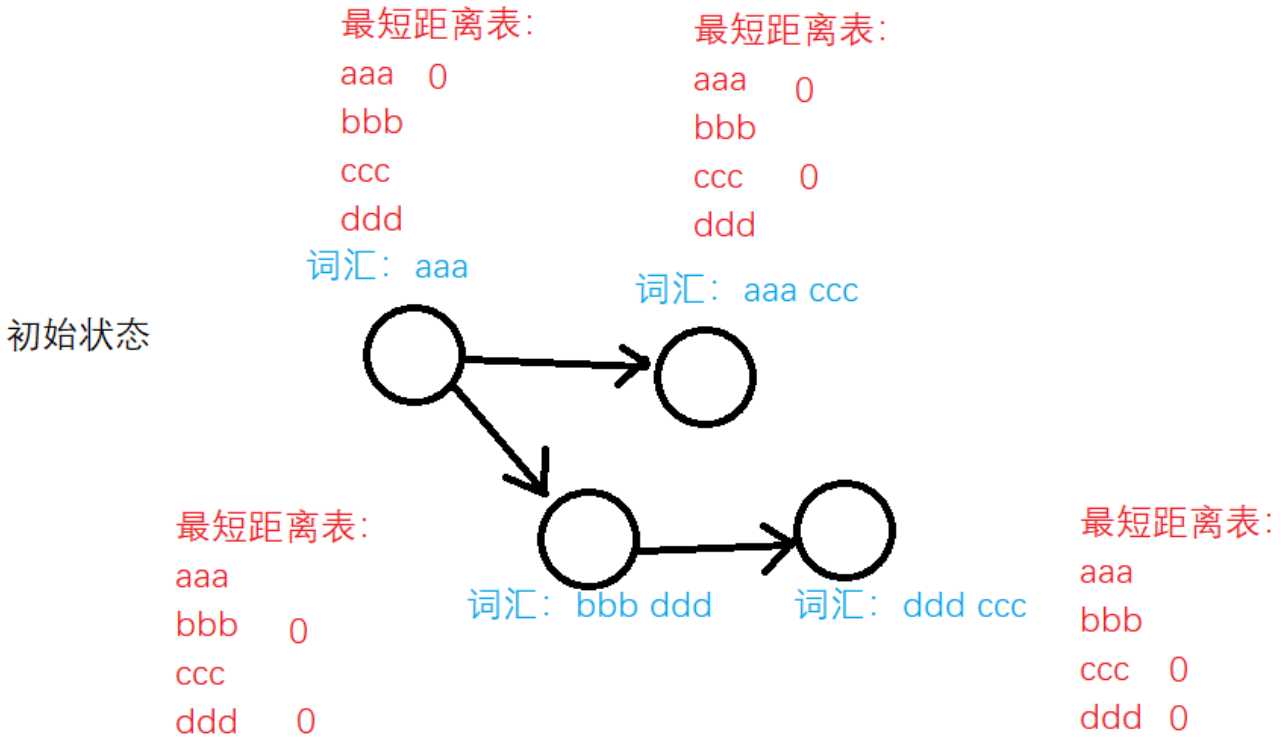
对他们来说这是一次无效的bfs，故可以提前结束

存在不劣于当前更新的选择，没用必要继续更新了



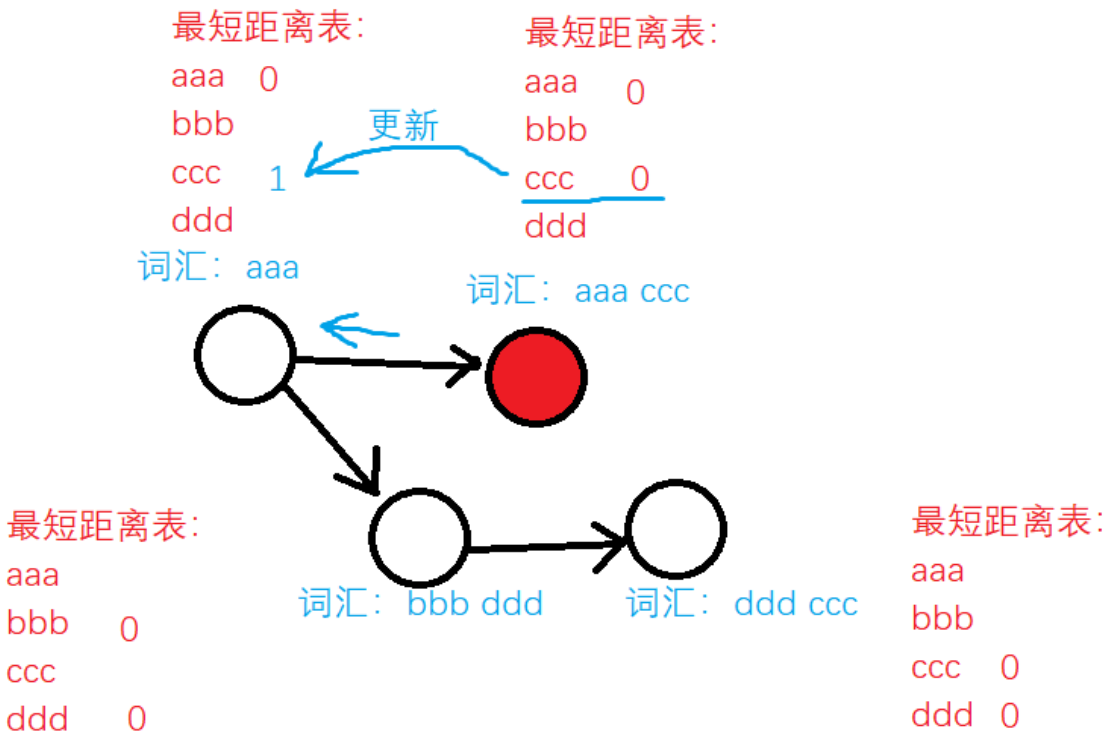
图解

我们给每个节点都分配距离表dis，并且初始化这些距离表，下面是初始状态



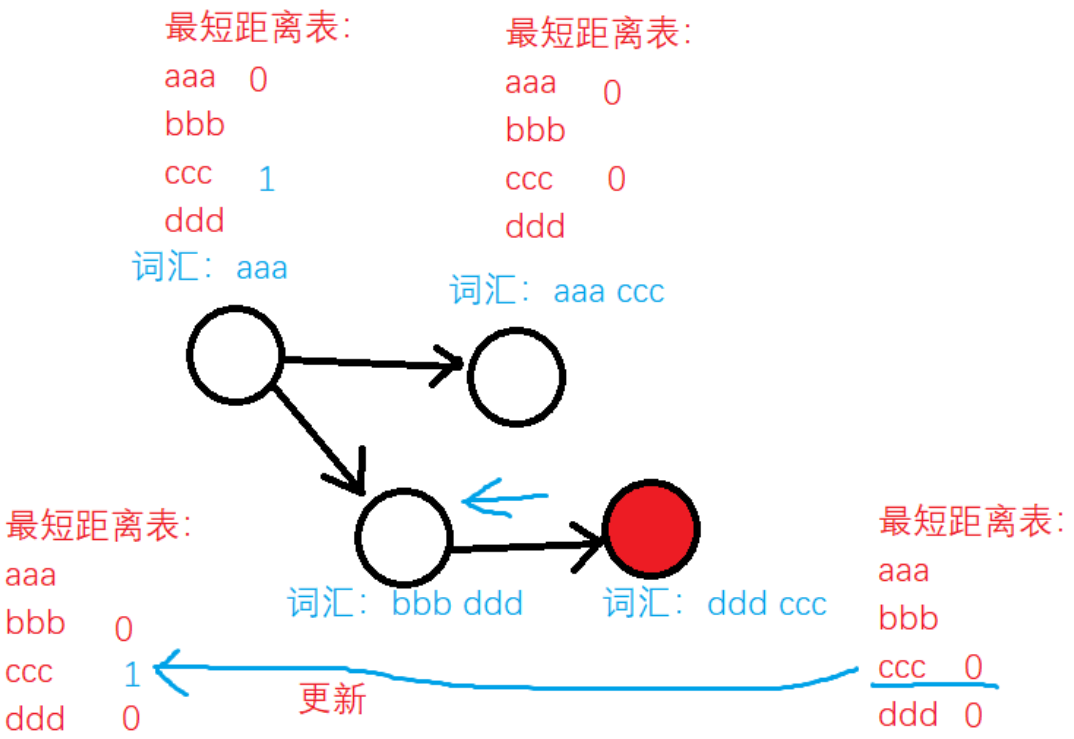
图解

假设查询词汇是{ddd,ccc,bbb}我们开始从包含查询词汇的节点（下图标红）开始bfs



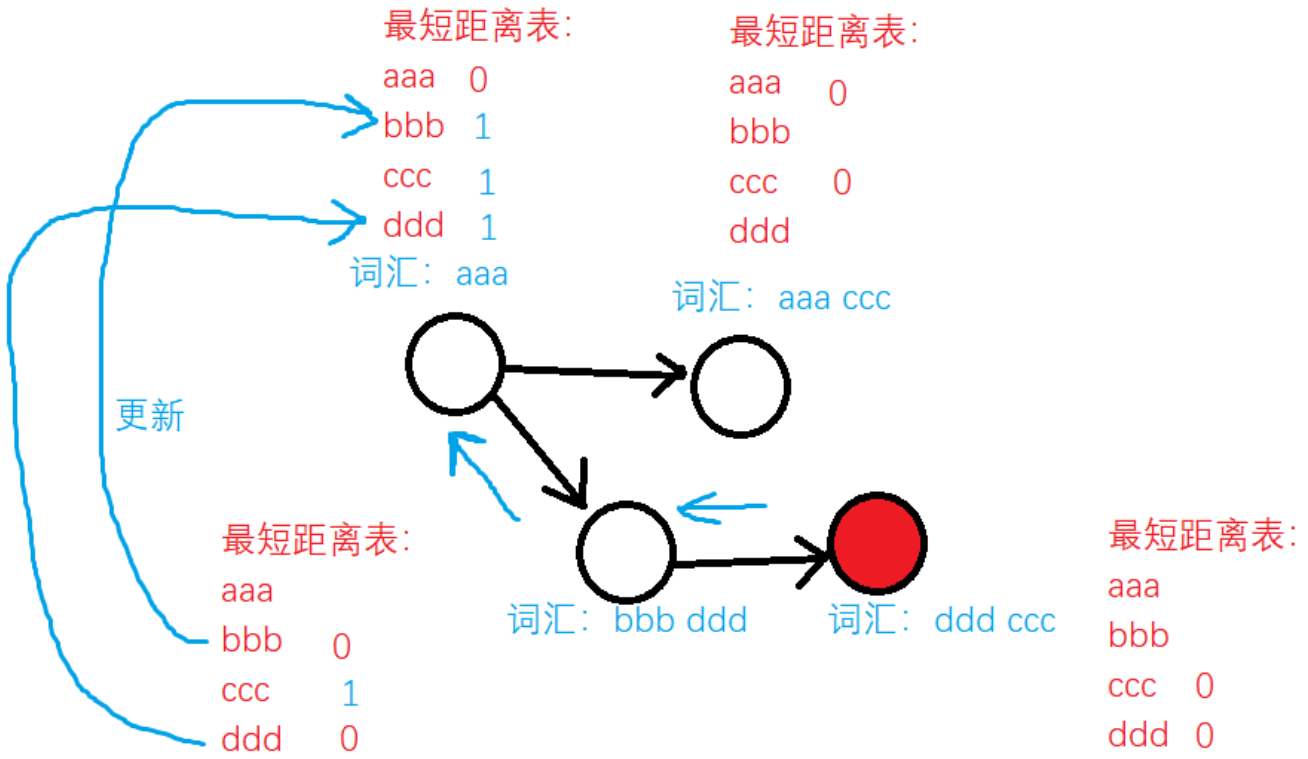
图解

已经无路可走（没有反向边了），我们开始从下一个包含ccc的顶点（下图标红）做bfs



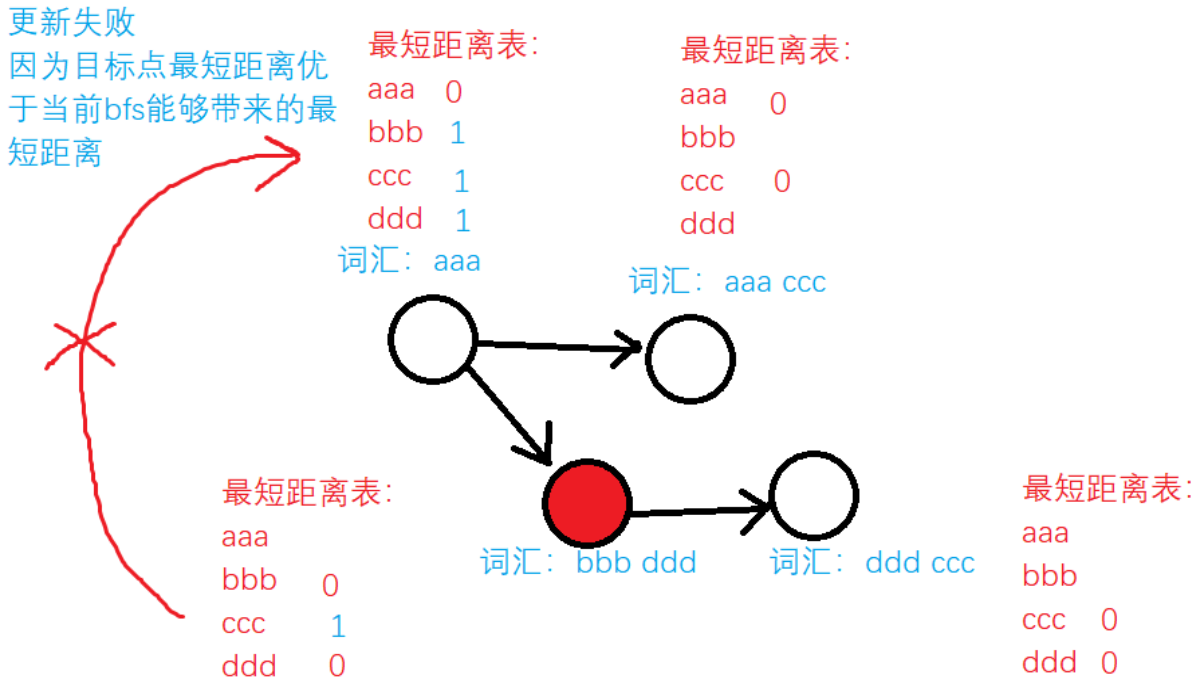
图解

继续沿着反向边bfs并试图更新最短距离（如下图蓝色箭头所示）



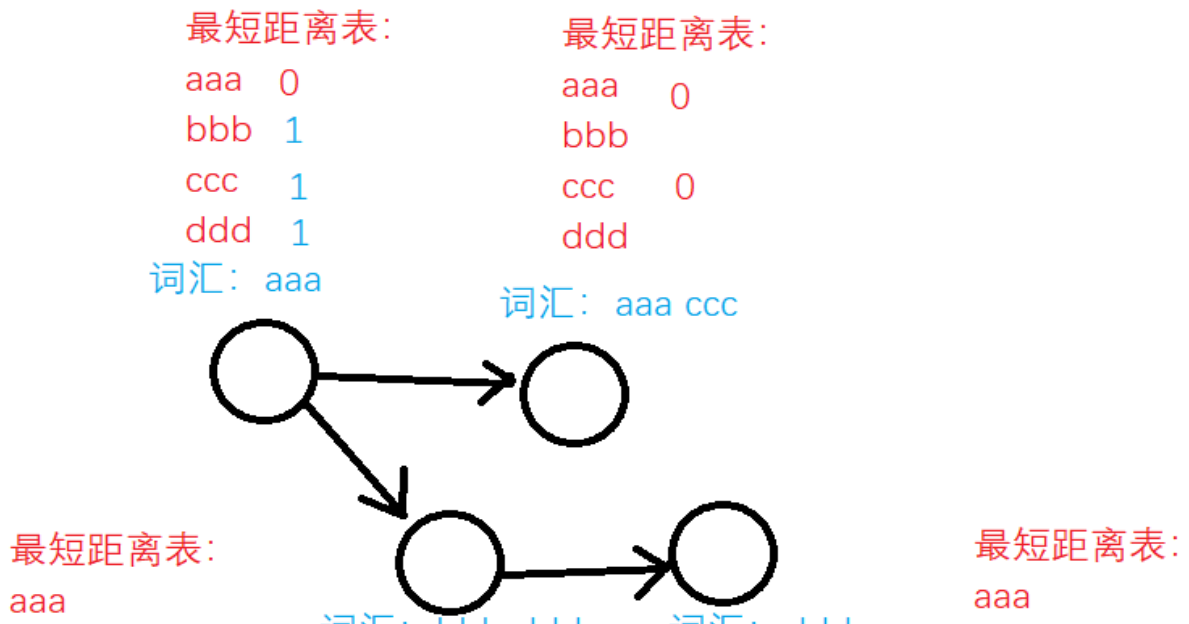
图解

bfs结束。此时还有包含关键词的节点（下图标红），我们尝试再次从红点出发bfs，发现无法更新任何点的距离。这说明这一次bfs带来的最短距离的更新，不能够让其他点的最短距离进一步缩短，就可以舍弃这一次的bfs，因为存在更优解。



图解

全部包含目标词汇的点都bfs之后，得到距离表即是每个点的答案



反向bfs复杂度分析

最坏情况仍然要对n个点做bfs，每次bfs的复杂度为O(n+e)，总体复杂度O(n*e)

但是因为实际上数据集中包含查询词汇的点始终占少数，而且bfs还有提前结束的策略，实际执行速度非常快，可以通过接下来的时间测试来验证这一点。

时间测试

随机生成不同数目的随机查询词汇集合

查询词汇的个数从1到5个不等，查询词汇随机选择集合中的词汇。

步数剪枝使用k=3步进行搜索。

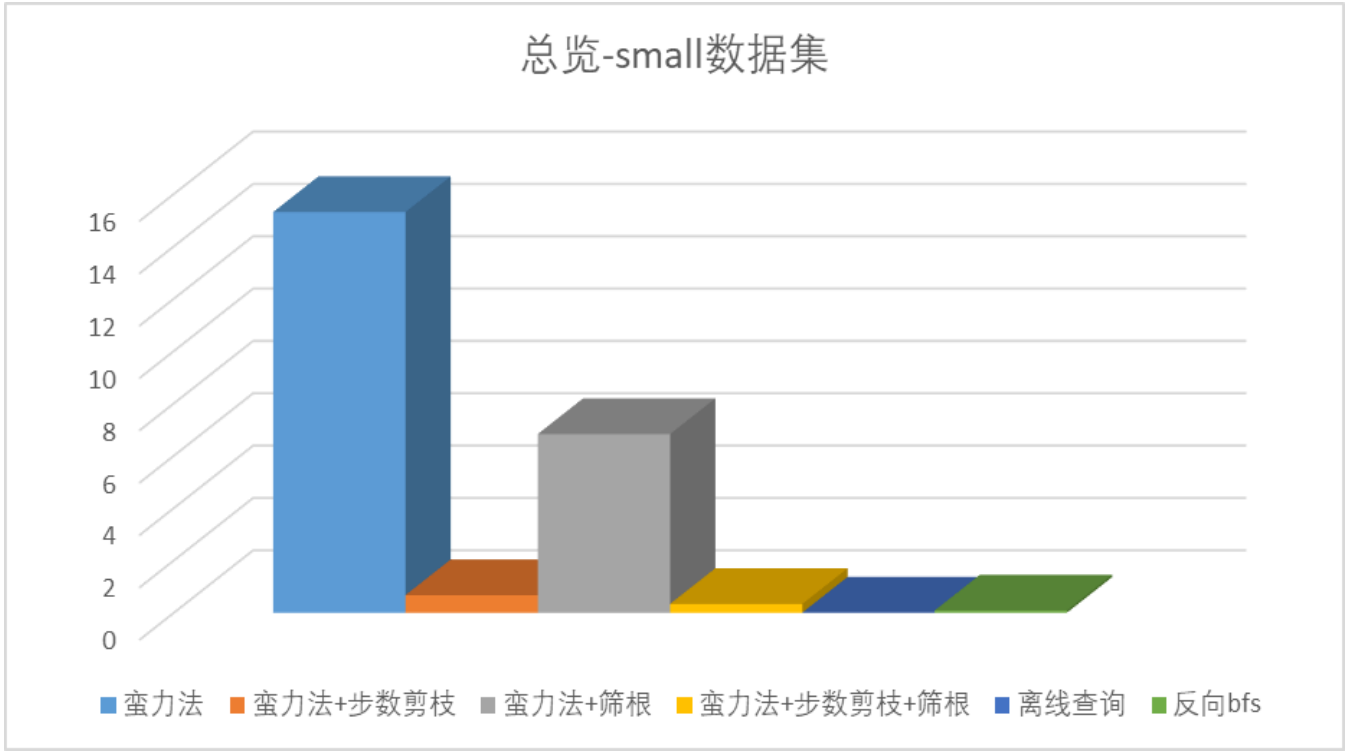
用相同的查询词汇集合，不同的查询方法查询skyline

记录各个方法平均运行时间时间

小规模

任何方法均可以在小规模数据上实现

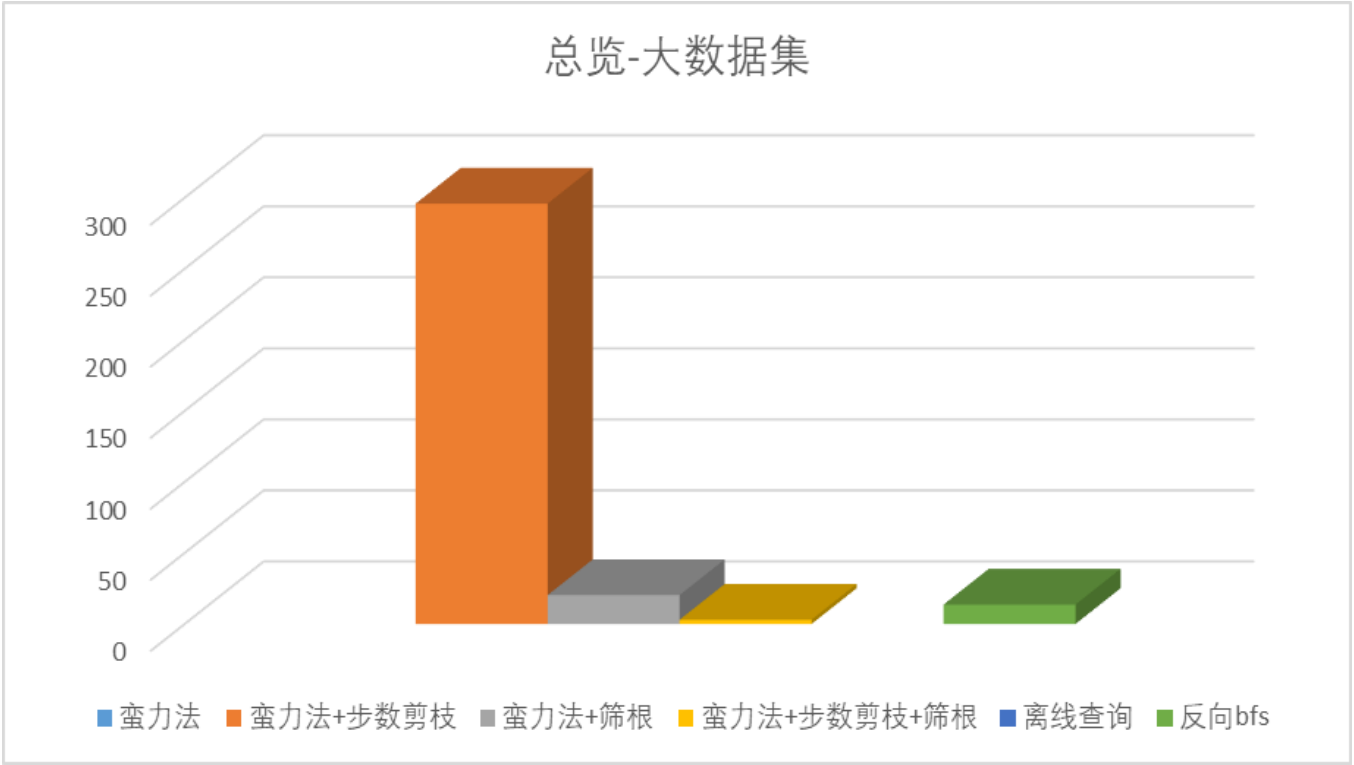
方法	平均时间
蛮力法	15.308
蛮力法+步数剪枝	0.6714
蛮力法+筛根	6.825
蛮力法+步数剪枝+筛根	0.3348
离线查询	0.0064
反向bfs	0.0858



大规模

可以看到蛮力法和离线查询因为开销过大而无法使用

方法	平均时间
蛮力法	
蛮力法+步数剪枝	295.839
蛮力法+筛根	20.2963
蛮力法+步数剪枝+筛根	2.978
离线查询	
反向bfs	13.6176



结论

蛮力法速度最慢，而蛮力法的两种剪枝策略，虽然速度快，但是正确性无法保证。

离线查询因为预处理的代价非常高，本质还是蛮力法，不适用于大规模图数据。

反向bfs是最有效的方法，既能保证速度也能保证正确性。

方法	速度	正确性	适用于大规模数据
蛮力法	慢	√	×
蛮力法+步数剪枝	中等	×	√
蛮力法+筛根	中等	×	√
蛮力法+步数剪枝+筛根	快	×	√
离线查询	快	√	×
反向bfs	快	√	√

总结

剪枝策略虽然能够高效提升效率，但是无法保证结果的正确性，需要在时间和正确性上，根据不同问题，做出取舍。

尽量使用高效容器：通过哈希表可以快速（常数时间）查询一个节点是否包含某个词汇，经过测试，实验环境选取c++11，未开启任何编译优化的情况下，使用unordered_set作为哈希容器，在30-50个元素的时候，查找效率和顺序查找难分伯仲，但是50个以上的元素，哈希容器的查找效率远远高于顺序查找。

要善于发现问题的重叠性：比如蛮力法bfs搜索每一个节点，很多bfs是无用且重复的，我们完全可以利用邻顶到词汇的距离来更新自己到词汇的距离。而反向bfs策略很好的利用这一点，减少重叠且无效的bfs搜索，进而提升运行效率。这也是动态规划思想的体现。

谢谢

谢谢