深圳大学期末考试试卷

开/闭卷 开卷 A/B 卷 ____

课程编号 15007700 03 课序号 01-09 课程名称 算法设计与分析 学分 3

命题人(签字)____ 审题人(签字)____ ____年____月____日

题号	一	二	三	四	五	六	七	八	九	十	基本题总分	附加题
得分												
评卷人												

基于 RDF 图的语义地点 skyline 查询计算

我们现在手里有一张图，这张图不是地图，而是对欧洲很多国家地理位置以及文化、艺术的抽象描述图。例如图 1 中的图 G 中有 9 个顶点，9 个顶点中的p_1、p_2、p_3、p_4、p_5是带有地理位置的顶点，除了地理位置，这些顶点也还有其他信息，例如顶点p_1的名字是 Florence，它的地理位置是意大利的佛罗伦萨，除了这个地理位置，p_1的其他属性包括：city（城市）, European（欧洲）, Italian（意大利）；顶点p_3的名字是 Florence Cathedral，它的地理位置是意大利的佛罗伦萨大教堂，除了这个地理位置，p_3的其他属性还包括：art（艺术）, myth（神话）, ancient（古老）。类似地，顶点p_2、p_4、p_5分别是 Athens（雅典）、Ancient Greece（古希腊）、Acropolis（卫城），他们也都有各自的属性，见图 2。

除了这些带有地理位置的顶点，还有一些顶点v_1、v_2、v_3、v_4，这些顶点没有地理位置信息，只有一些描述信息，例如v_1的名称是 House of Medici（麦地奇家族），v_1的属性包括：renaissance（文艺复兴）, commerce（商业）, dynasty（王朝），这个顶点描述的是佛罗伦萨 15 世纪至 18 世纪中期在欧洲拥有强大势力的名门望族——麦地奇家族；v_2的名称是 Michelangelo（米开朗基罗），v_2的属性包括：Italian（意大利）, sculptor（雕塑家），painter（画家），architect（建筑），这个顶点描述的是意大利文艺复兴时期著名的艺术家米开朗基罗；v_3的名称是 David（大卫），v_3的属性包括：Sculpture（雕塑），art（艺术），history（历史），这个顶点描述的是米开朗基罗著名的雕塑大卫。类似点，顶点v_4描述的是希腊哲学家 Socrates（苏格拉底）。

图 1 中的顶点如果相互有某种关系，则存在边相互连接，例如佛罗伦萨大教堂（p_2）在佛罗伦萨（p_1），所以p_1和p_2之间有边连接，麦地奇家族（v_2）是佛罗伦萨（p_1）的名门望族，所以和p_1之间有边连接。麦地奇家族在建筑和艺术方面进行了赞助，米开朗基罗（v_2）得到过他们的赞助，所以v_1和v_2之间有边连接，米开朗基罗创作了著名的雕塑大卫，则v_2和v_3之间有边连接，这样就得到了图 1。我

们称图 1 是 RDF 图，图 2 是图 1 中各顶点的属性。

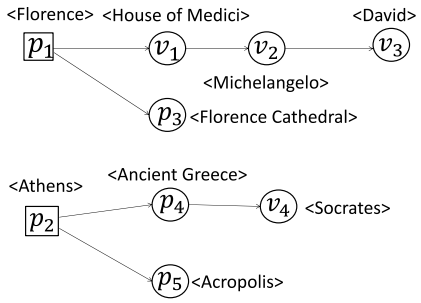


图 1 RDF 图

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}

图 2 顶点的文本属性

下面我们给出 RDF 图的数学定义。令有向图$G = (V, E)$表示一个 RDF 图，其中V是图中顶点的集合，E是图中边的集合。在图G中，带有空间坐标（地理信息）属性的顶点称作地点，例如图 1 中的p_1、p_2等。图G中，顶点$v \in V$的文本属性$v.\psi$是一组词的集合，注意这个符号$v.\psi$表示的是顶点v的文字属性，例如，$v_1.\psi = \{\text{renaissance, commerce, dynasty}\}$。

定义 1（**相关语义地点**）. 给定一组关键词ϕ和一个图$G = (V, E)$，一个与ϕ相关的语义地点是一棵树$T_p = (V', E')$，这颗树以地点p为根节点，并且满足条件$V' \subseteq V, E' \subseteq E$，$\bigcup_{v \in V'} v.\psi \supseteq \phi$，这个符号的意思是所有顶点$v$（顶点$v$属于树$T_p$）的文本属性的并集包含了关键词集合$\phi$。

在我们给出图 1 的例子中，关键词$\phi = \{\text{sculpture, art, history}\}$，图 1 中有两棵树是与$\phi$相关的语义地点，第一棵树以$p_1$为根，树中其它所有顶点$v_1$、$v_2$、$v_3$、$p_3$的文本属性的并集包含了这些关键词；第二棵树以$p_2$为根，树中其它所有顶点$v_4$、$p_4$、$p_5$的文本属性的并集包含了这些关键词。

在一个相关语义地点$T_p = (V', E')$中，根节点p到一个关键词$w_i \in \phi$的距离$d_g(T_p, w_i)$用下面公式计算：

$$d_g(T_p, w_i) = \min_{v \in V'} d(p, v)，\ w_i \in v.\psi$$

其中，$d(p, v)$是相关语义地点T_p中p到v的最短路径的长度.

定义 2（**语义地点支配**）. 给定一组关键词ϕ和一个图$G = (V, E)$，令$\mathbb{T}$表示所有相关语义地点的集合，语义地点支配是$\mathbb{T}$上的一个偏序关系。这个偏序关系定义为：给定$\mathbb{T}$中的两个相关语义地点T_{p_1}和T_{p_2}，如果 T_{p_1}支配T_{p_2}，则T_{p_1}与T_{p_2}之间具有偏序关系，记为$T_{p_1} \prec T_{p_2}$。当且仅当下面两个条件满足，则称T_{p_1}支配T_{p_2}：

$\exists w_i \in \phi$，满足$d_g(T_{p_1}, w_i) < d_g(T_{p_2}, w_i)$

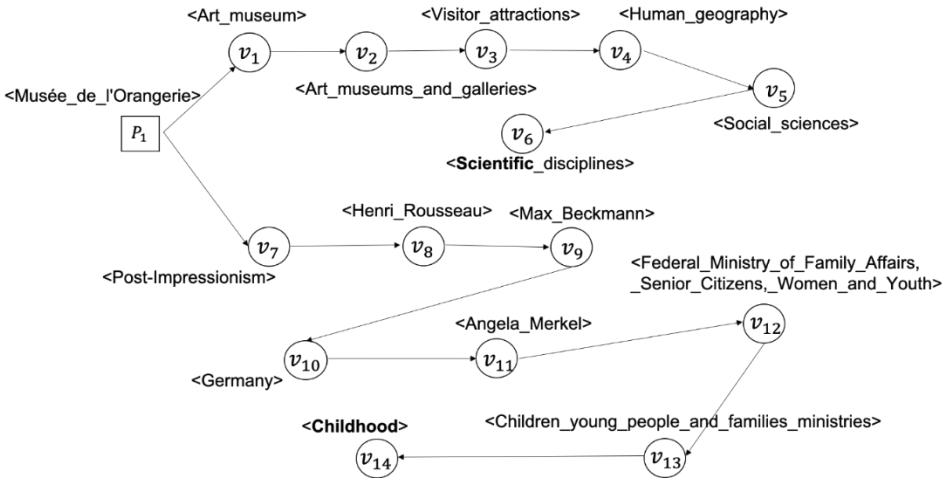
$\forall w_i \in \phi$，满足 $d_g(T_{p_1}, w_i) \leq d_g(T_{p_2}, w_i)$

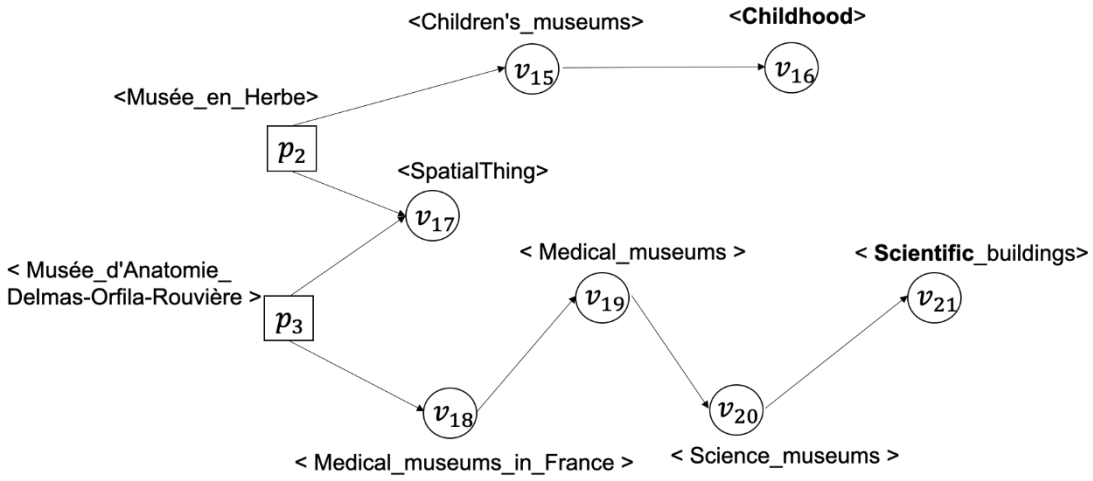
定义 3（**语义地点 skyline 查询**）. 给定一组查询关键词ϕ和一个图$G = (V, E)$,，令$\mathbb{T}$表示所有与ϕ相关的语义地点的集合，$T_p \in \mathbb{T}$是一个语义地点 skyline，当且仅当$\mathbb{T}$中不存在支配T_p的语义地点。在图G上,一个语义地点 skyline 查询ϕ的结果是$\mathbb{T}$中所有语义地点 skyline 的集合，记为$S(G, \phi)$。

举个栗子，在图 1 中，给定一组查询关键词$\phi = \{\text{sculpture, art, history}\}$，图 1 所示的 RDF 图G，其中顶点的文本属性如图 2 所示，其中 $T_{p_1} = \langle p_1, (v_1, v_2, v_3, p_3)\rangle$和$T_{p_2} = \langle p_2, (v_4, p_4, p_5)\rangle$是两个相关语义地点。在这两个语义地点中，关键词到两个根节点的距离分别是$d_g(T_{p_1}, sculpture) = 3$，$d_g(T_{p_1}, art) = 1$，$d_g(T_{p_1}, \text{history}) = 3$，$d_g(T_{p_2}, sculpture) = 1$，$d_g(T_{p_2}, art) = 1$ ，$d_g(T_{p_2}, \text{history}) = 2$ 。根据语义地点支配的定义，可以看出T_{p_2}支配T_{p_1}，因为存在一些关键字$w_i \in \phi$，这些关键词在T_{p_2}中的距离小于在T_{p_1}中的距离，并且所有关键字在T_{p_2}中的距离都小于等于在T_{p_1}中的距离，这样就可以记为$T_{p_2} \prec T_{p_1}$。那么语义地点 skyline 查询$\phi = \{\text{sculpture, art, history}\}$的结果是$T_{p_2}$

大作业要求：

1、 （30 分） （1）写一段文字描述，说明上述语义地点 skyline 查询可以用在什么地方？ （2）设计一个算法计算语义地点 skyline 查询的结果并用伪代码描述； （3）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，例如下图中v_2有四个文本属性。查询关键词$\phi = \{\text{scientific, childhood}\}$，计算下图的语义地点 skyline。





2、（30 分）附件中提供了 Yago_small 是一个小规模图数据，请编写代码，在该数据上完成查询。

3、（40 分）附件中提供的 Yago 数据集包含 8,091,179 顶点和 50,415,307 条边，是一个大规模图数据，请编写代码，在该数据上完成查询，要求高效率完成查询。

数据集格式说明：

- 数据中顶点和关键词都用整数表示。
- 文件 edge.txt 是有向图的邻接链表，每一行代表一个顶点的邻接顶点。
  - 例如，8: 6291031,5330605,6481451,6280292，表示顶点 8 有四个邻接顶点，它们是 6291031,5330605,6481451,6280292。
- 文件 node_keywords.txt 是顶点的文本属性，每一行代表一个顶点的文本属性。
  - 例如，0: 8973992,10029808,8435980，表示顶点 0 有三个关键词，它们是 8973992,10029808,8435980。

评分标准：

1、通过 PPT 讲解算法思想以及数据测试结果，并演示代码；
2、撰写大作业报告，要求论述问题，讲述求解算法思想，并给出伪代码描述，进行数据测试，并进行分析。
3、PPT 讲解和大作业各占 50%成绩。

基于 RDF 图的语义地点 skyline 查询问题，问题分为

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

RDF 图：

有向图，图的每个顶点包含一些词汇信息

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：

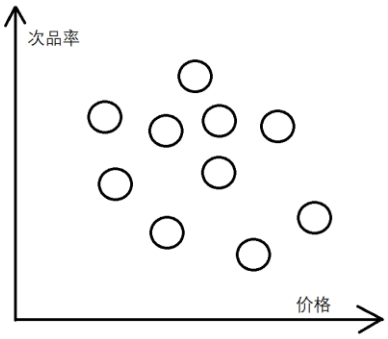
如果对两个查询词汇集合 w 的语义地点 p1 和 p2

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

语义地点 skyline：

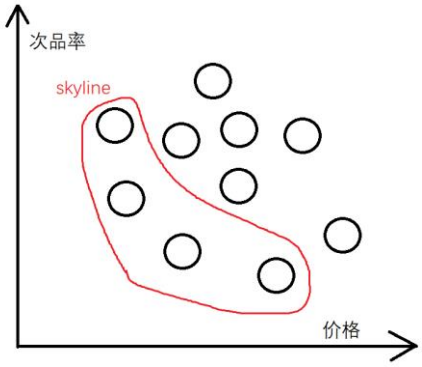
给定一组词汇 w，和一个 RDF 图，在图中所有 w 的语义地点集合中，找到一个子集使得子集中的语义地点互不支配。我们用一个“采购问题”来描述 skyline 查询问题：假设工厂希望采购【既便宜又次品率低】的原材料，而采购部列出以下的选择：



因为存在很多因素描述一个选择的好与坏，比如上图存在两个因素“价格”和“次品率”。

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

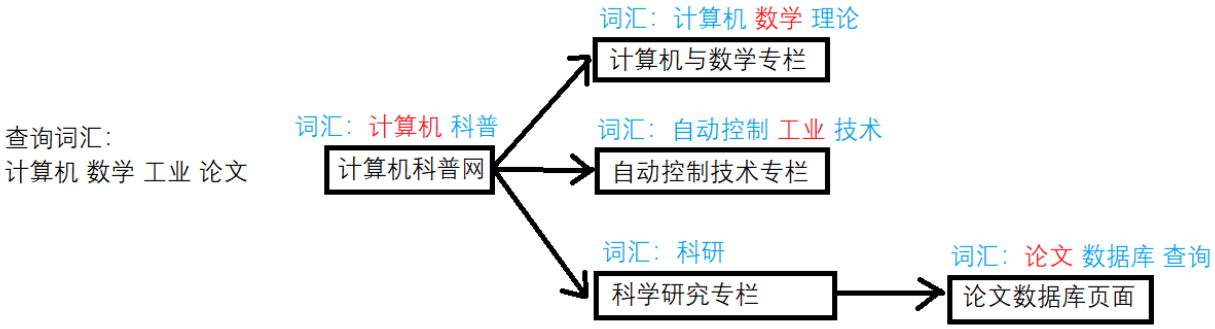


我们称这些选择为【以价格和次品率为评价标准的 skyline】。同样的，RDF 图中描述一个语义地点的好坏，也是通过他们到词汇的距离来决定的，我们的任务是在 RDF 图中找到所有以词汇距离为评价标准的语义地点中的 skyline。

语义地点查询应用场景：

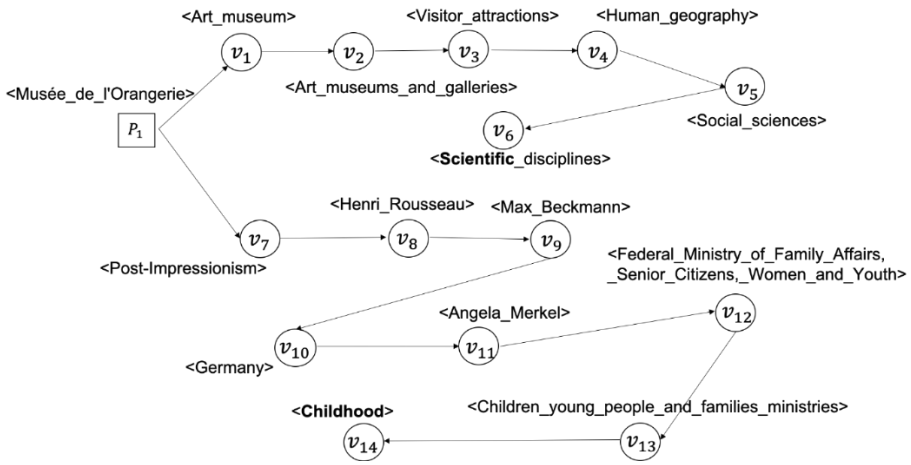
语义地点 skyline 的查询可以运用在许多多目标优化的领域，比如搜索引擎的相关性查询，网页之间的超链接表示为图的边，而一个语义地点代表一个网页的主页。我们希望得到所有和我们查询的词汇最相关的网页主页。

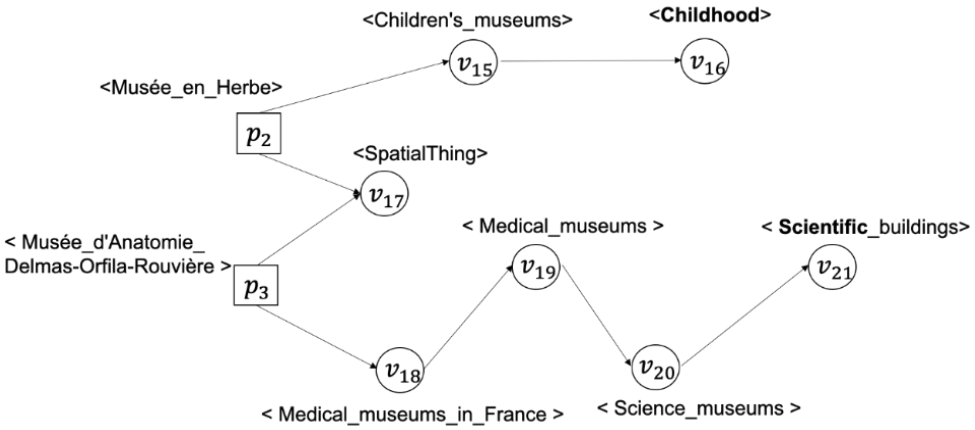
如图，我们查询的词汇是【计算机 数学 工业 论文】，下图展示了一个语义地点。每个节点代表一个页面，边则代表页面之间的超链接，该语义地点包含所有查询词汇。



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，例如下图中 v2 有四个文本属性。查询关键词 ，计算下图的语义地点 skyline。





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

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。

算法：

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，但是执行速度非常快速。

节点词汇查询：

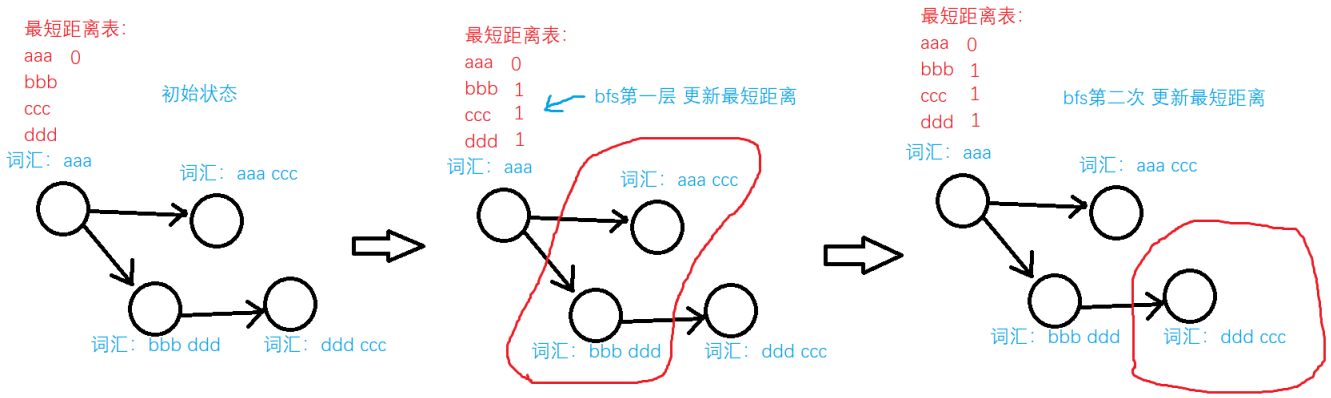
通过建立每一个节点的哈希表来作为节点词汇集合，利用哈希的特性可以在 O(1)的常数时间内查询该节点是否具有某个词汇，表的建立在读取原始数据时就已经完成。

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。



上图描述了一次 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
```

蛮力法复杂度分析：

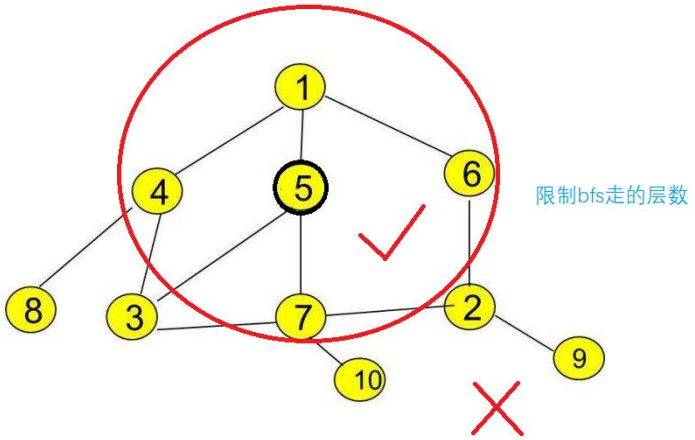
枚举 n 个源点进行 bfs 共需要 n 次 bfs，每次 bfs 的复杂度是 O(n+e)，取上限为 O(e)，所以总体复杂度 O(e*n)

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：

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。

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。

蛮力法+筛根分析：

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。

离线查询（蛮力法 2）：

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。

反向 bfs 求解语义地点：

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，而很多起点根本到不了目标词汇，存在很多无效的 bfs，所以蛮力法的时间开销相当大。

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。即只做有用的 bfs。

反向 bfs 求解距离分析：

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：

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

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。

提前结束策略：

除此之外，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

反向 bfs 伪代码描述：

Bfs 伪代码：

```
// 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)
```

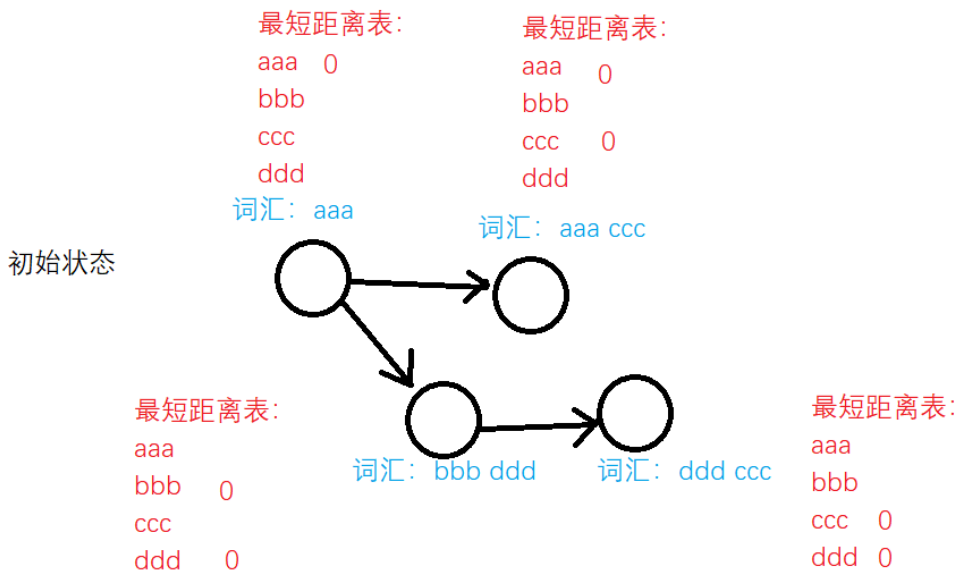
和蛮力法搜寻词汇不同，反向 bfs 自己主动更新其他节点到查询词汇的距离，而不是等待别人来搜索自己。这么做大大减少了无效的 bfs。这种更新方式和自底向上的动态规划异曲同工。

主程序伪代码：

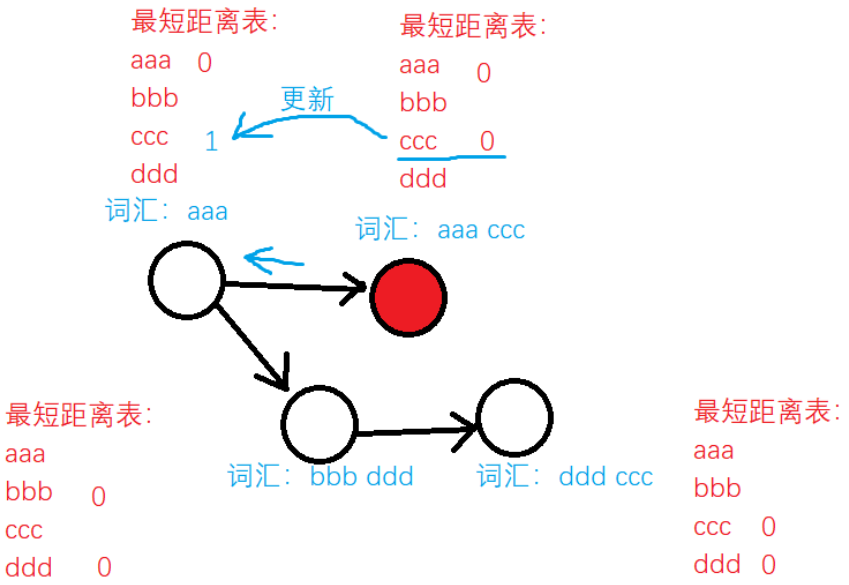
```
for i in n:
    if i节点包含某个查询词汇:
        bfs_rev(src)

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
```

下面通过图解来描述反向 bfs 算法的过程：

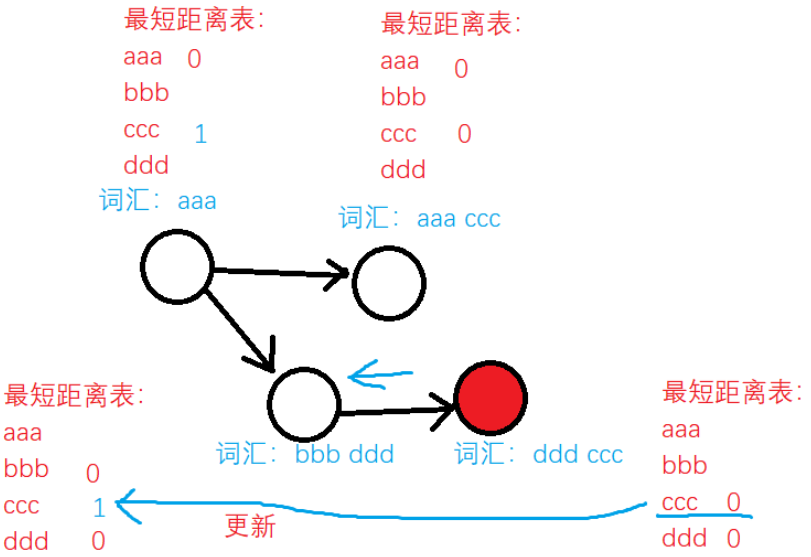
我们给每个节点都分配距离表 dis，并且初始化这些距离表，下面是初始状态



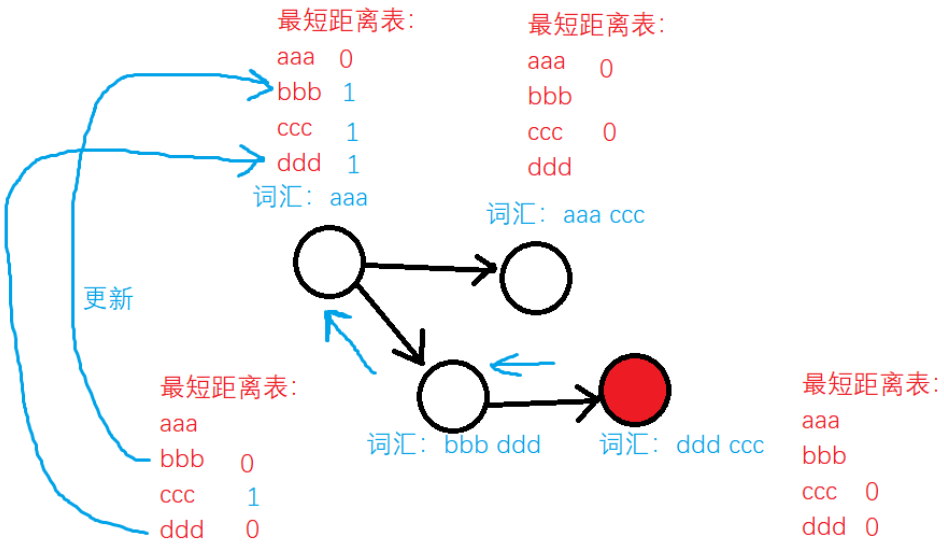
假设查询词汇是{ddd, ccc, bbb}我们开始从包含查询词汇的节点（下图标红）开始 bfs



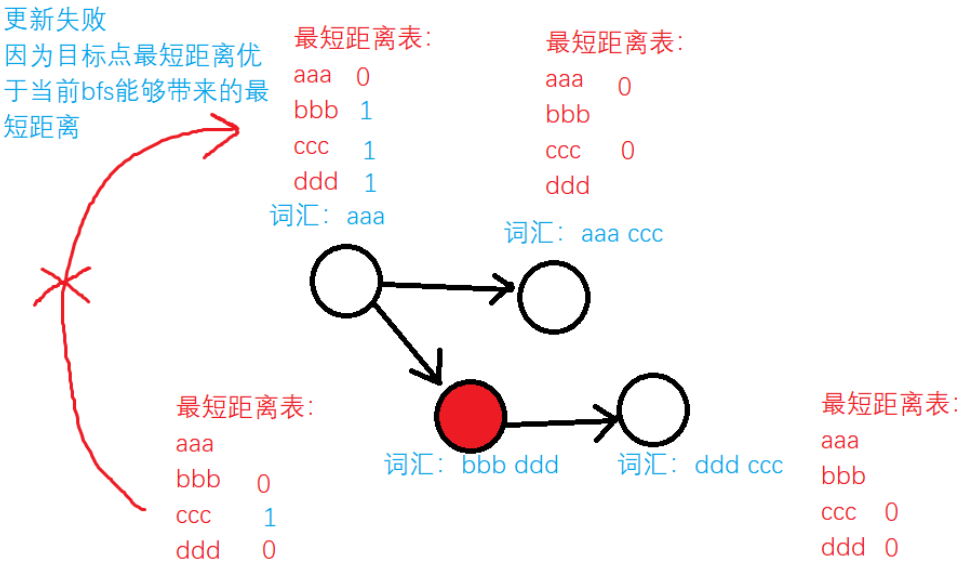
已经无路可走（没有反向边了），我们开始从下一个包含 ccc 的顶点（下图标红）做 bfs



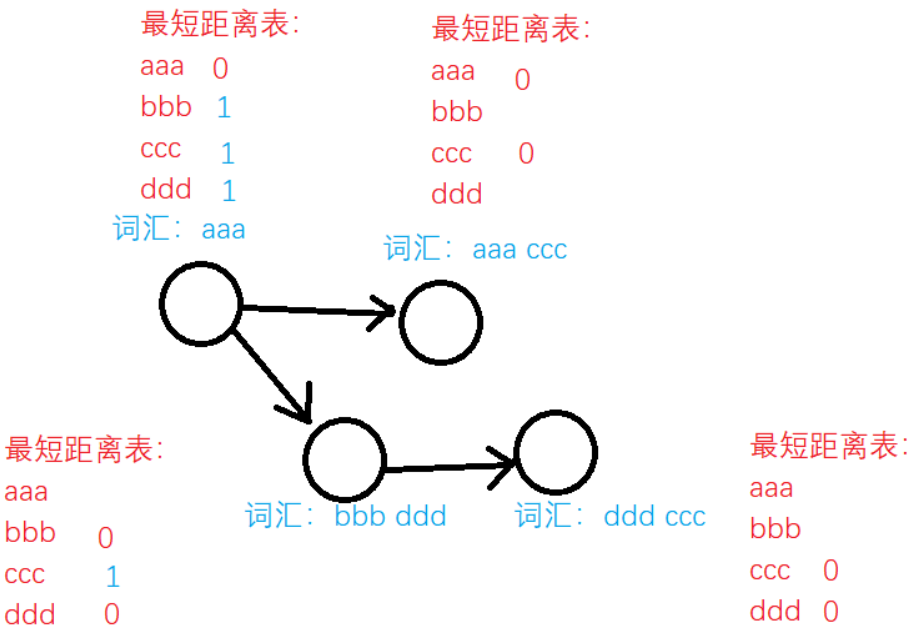
继续沿着反向边 bfs 并试图更新最短距离（如下图蓝色箭头所示）



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。



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：

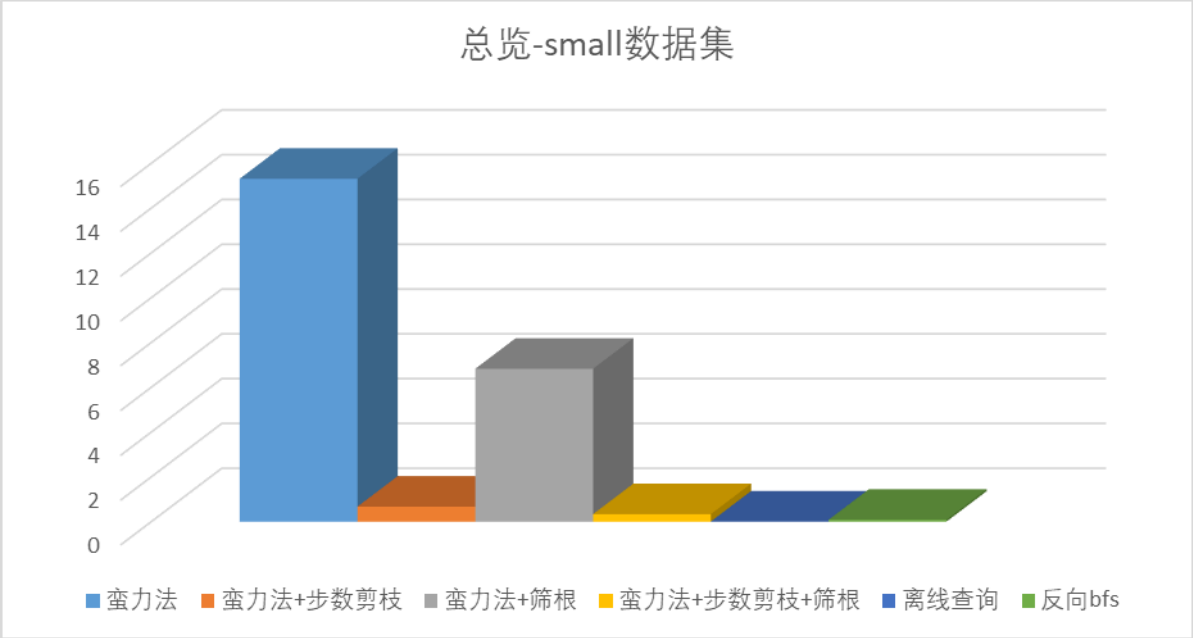
随机生成不同数目的随机查询词汇集合，其中查询词汇的个数从 1 到 5 个不等，查询词汇随机选择集合中的词汇。步数剪枝使用 k=3 步进行搜索。

用相同的查询词汇集合，不同的查询方法查询 skyline，并且记录各个方法平均运行时间时间

小规模数据：

任何方法均可以在小规模数据上实现

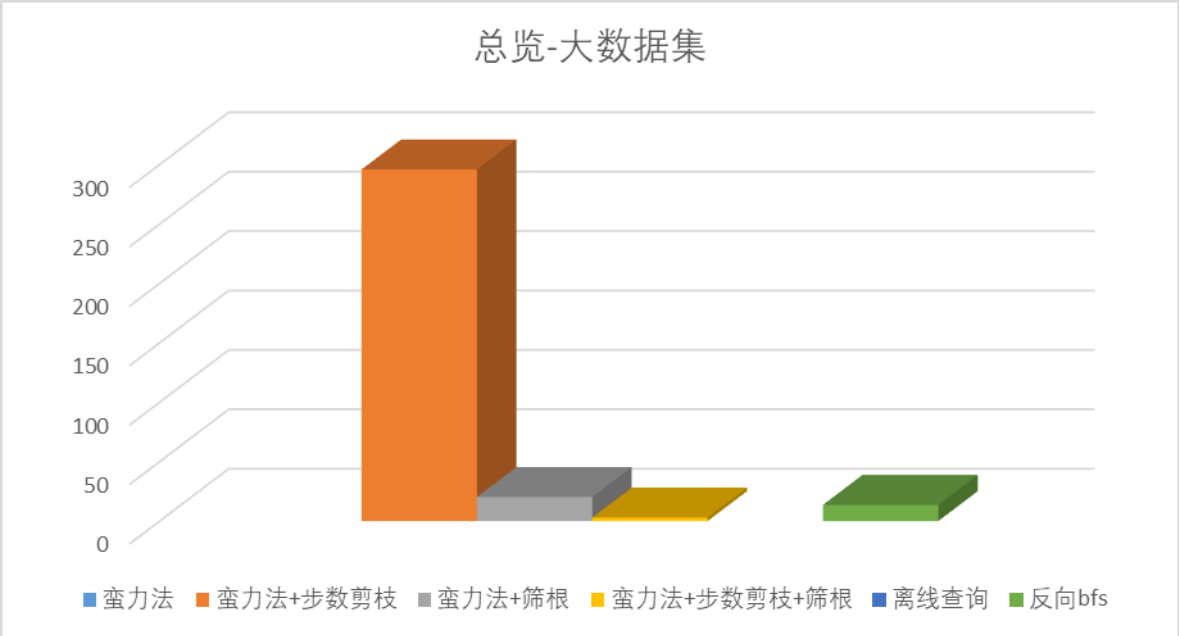
方法	平均时间
蛮力法	15.308
蛮力法+步数剪枝	0.6714
蛮力法+筛根	6.825
蛮力法+步数剪枝+筛根	0.3348
离线查询	0.0064
反向bfs	0.0858



大规模数据：

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。

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。

指导教师批阅意见：   成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。