图上最短路径距离 查询算法集成演示

双向BFS Dijkstra CH HubLabel Pruned landmark labeling H2H HighwayLabel

一、内容简介

回答精确的最短路径距离查询。毫无疑问，这是图论中的一项基本任务，它有着广泛的应用。例如，在社交网络上，两个用户之间的距离被认为表示亲密程度，并用于社交敏感搜索，以帮助用户找到更多相关的用户或内容，或者是分析有影响力的人和社区。在网络图上，网页之间的距离是相关性的指标之一，并用于上下文感知搜索，以对与当前访问的网页更相关的网页给予更高的排名。距离查询的其他应用包括对链接数据的top-k关键字查询、发现代谢网络中化合物之间的最佳途径以及管理计算机网络中的资源。

当涉及到大型图形时，我们可以使用宽度优先搜索（BFS）或Dijkstra算法计算每个查询的距离。但是这些算法需要一定的计算时间，特别是对于社交敏感搜索或上下文感知搜索等需要实时交互的应用程序，它们需要多对顶点之间的距离来对每个搜索查询的项目进行排序。因此，距离查询的回答速度应该快得多，比如说，微秒。另一种极端方法是预先计算所有顶点对之间的距离，并将其存储在索引中。虽然我们可以立即回答距离查询，但这种方法也是不可接受的，因为预处理时间和索引大小都是二次的，而且非常大。

由于大量图形数据的出现在我们身边应用中，因此本实验我们进行了对大图中的最短路径距离查询算法进行了探究。

二、理论基础

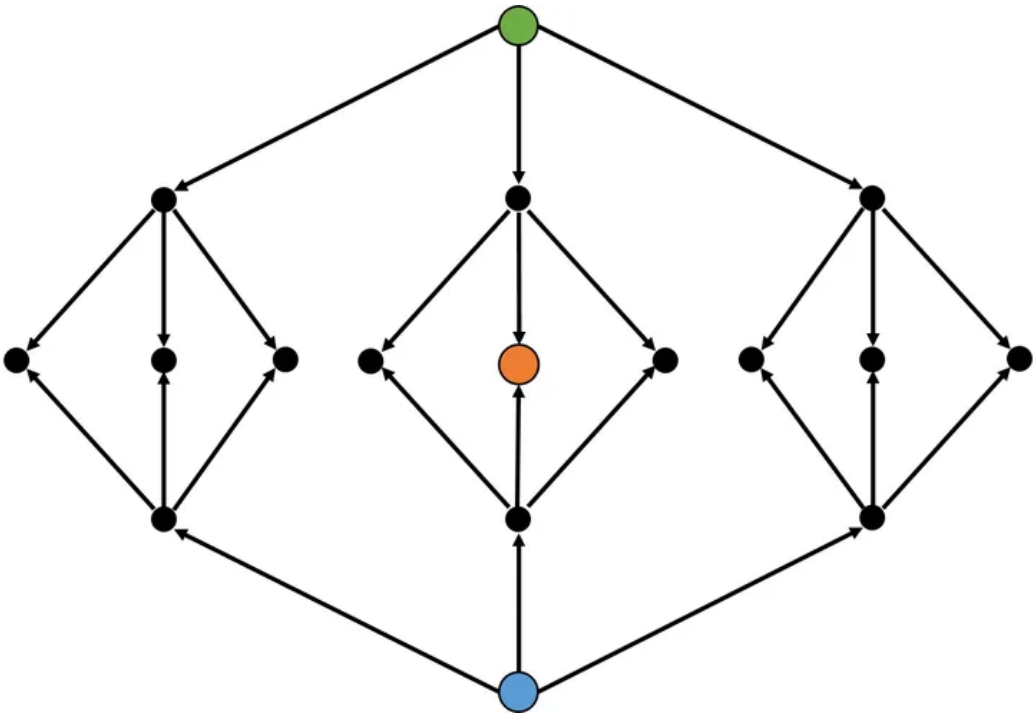
知识点一：Bi-BFS算法

2.1.1 算法背景及原理介绍

双向BFS（Bidirectional Breadth-First Search）算法是一种优化的BFS算法，用于在图中搜索两个节点之间的最短路径。相较于单向BFS算法，双向BFS算法可以减少搜索的状态空间和搜索时间，特别是对于较大的图形或搜索空间。

双向BFS算法是一种在两个方向上同时进行搜索的算法，它通常用于在起点和终点较远的情况下，找到最短路径或最小代价的问题。与传统的BFS算法不同，双向BFS算法从起点和终点同时开始，每次都向外扩展一层节点，直到两个搜索的路径相遇。这种方法比单向BFS算法更快，因为它可以避免在整个

搜索空间中进行过多的扩展。此外，通过双向BFS算法，我们可以减少需要访问的节点数量，从而更快地找到最短路径。



2.1.2 算法流程

具体来说，Bi-BFS算法可以分为以下几个步骤：

1. 从起点开始，向外扩展一层节点，同时记录它们的父节点。
2. 从终点开始，向外扩展一层节点，同时记录它们的父节点。
3. 检查从起点扩展的节点是否与从终点扩展的节点重合。如果重合，表示找到了一条最短路径。
4. 如果没有重合，继续扩展下一层节点，直到找到重合点或者没有更多节点可以扩展。
5. 如果找到重合点，就从两个方向分别按照父节点逐步回溯，构建最短路径。

注意：双向BFS算法中的起点和终点可以交换，算法流程不变，只需要最后回溯时从相遇点的两个方向分别回溯即可。

2.1.3 输入输出

输入：起始节点、终止节点

输出：最短距离

2.1.4 优缺点

Bi-BFS的优缺点如下：

优点：

1. 可以找到最短路径：BFS算法从起点开始扩展，每次都将距离起点更近的节点先加入队列，这样可以保证当第一次到达目标节点时，所经过的路径就是最短路径。

2. 算法的正确性较高：BFS算法可以避免陷入局部最优解的情况，因为每次都是从当前距离起点最近的节点开始扩展。

3. 可以显著减少搜索空间，双向BFS算法只需要存储两个方向的访问记录，因此在存储空间上相对较小。从而加快搜索速度。由于双向BFS从两个方向同时搜索，因此在搜索的过程中，可以更早地发现目标节点，从而提高搜索的效率。

缺点：

1. 实现相对复杂：相较于单向BFS算法，双向BFS算法需要维护两个队列以及两个 visited 集合，实现起来相对较为复杂。

2. 双向BFS算法的实现需要确定起点和终点节点，如果起点和终点之间的路径非常复杂或不存在，那么双向BFS的优势将大打折扣。

3. 双向BFS算法适用于有明确起点和终点的搜索问题，如果没有明确的目标节点，则双向BFS并不适用。

总的来说，双向BFS算法比单向BFS算法更高效，因为它可以减少搜索的空间和时间。但是，它需要更多的空间来存储两个方向的已访问节点，并且实现起来比较复杂。

知识点二：Dijkstra算法

2.2.1 算法背景及原理介绍

Dijkstra算法是一种用于解决加权图中单源最短路径问题的贪心算法。它的基本思想是从源节点开始，逐步计算到所有其他节点的最短距离。在这个过程中，算法会维护一个集合S，其中包含已经找到最短路径的节点，以及一个距离数组dist，用于记录从源节点到各个节点的当前最短距离。在每一步中，算法会选择距离源节点最近的未访问节点，并更新其邻居节点的距离值，直到所有节点都被访问为止。

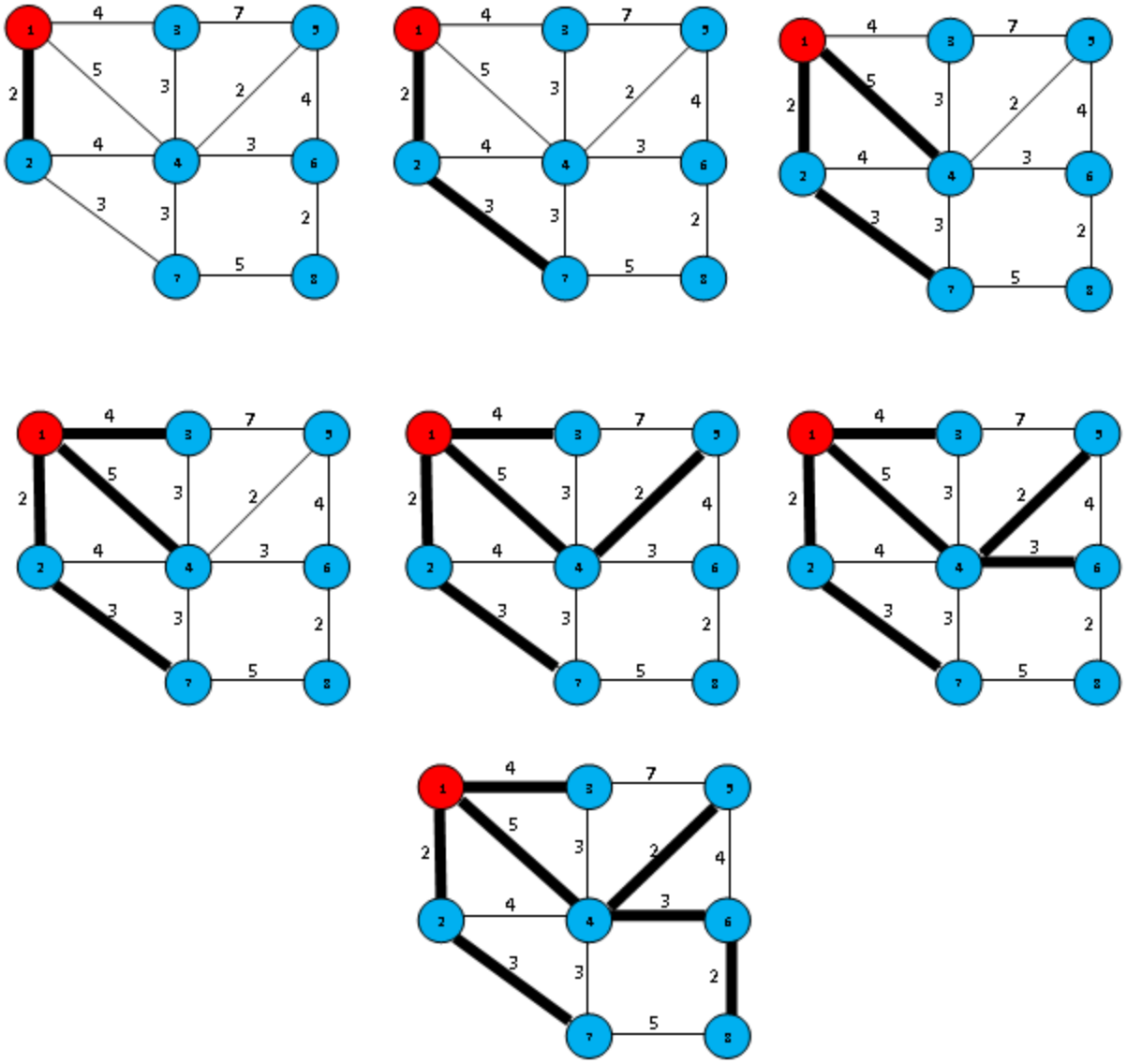
具体而言，Dijkstra算法会维护一个优先队列，用于存储候选节点。开始时，将源节点加入队列，并将其距离值设为0。之后，重复以下步骤，直到队列为空：

1. 从队列中取出距离值最小的节点u，并将其加入集合S中。

2. 遍历节点u的所有邻居节点v，并更新它们的距离值dist[v]。具体地，对于每个邻居节点v，如果通过节点u可以获得更短的距离，则更新dist[v]的值，并将节点v加入队列中。

在整个过程中，算法会保证已经加入集合S中的节点的距离值是最短的，因此当算法结束时，dist数组中存储的就是源节点到所有其他节点的最短距离。需要注意的是，Dijkstra算法仅适用于没有负权边的

图。如果图中存在负权边，则需要使用另外一种算法，比如Bellman-Ford算法或SPFA算法。



2.2.2 算法流程

Dijkstra算法的流程如下：

1. 创建一个记录节点距离的列表，初始值为无穷大，但源节点的值为0，表示源节点到源节点的距离为0。

2. 创建一个记录节点是否被访问的列表，初始值为False，表示所有节点都未被访问过。

3. 当所有节点都被访问时，算法结束。

4. 从未被访问过的节点中选择距离源节点最近的节点，标记为当前节点。

5. 遍历当前节点所有的邻居节点，如果邻居节点的距离大于当前节点距离加上当前节点到邻居节点的边的距离，则更新邻居节点距离为当前节点距离加上当前节点到邻居节点的边的距离。

6. 将当前节点标记为已访问。

7. 重复步骤4–6，直到所有节点都被访问。

8. 记录源节点到所有其他节点的距离。

9. 算法结束。

注：Dijkstra算法也可以使用优先队列来实现，这样可以提高效率。

2.2.3 算法实现结果

输入：起点、终点

输出：最短距离

2.2.4 优缺点

Dijkstra算法的优点包括：

1. 可以处理有权重的图形：与BFS不同，Dijkstra算法可以处理有权重的图形，因为它使用每个节点的实际距离来确定最短路径，而不是简单的步数。

2. 保证能够找到最短路径：Dijkstra算法使用贪心策略来选择当前路径中最短的边，因此它保证了能够找到从起点到终点的最短路径。

3. 可以生成最短路径树：Dijkstra算法可以生成一棵最短路径树，这棵树的根节点是起点，树的每个叶子节点是一个终点，且每个节点都是在生成它的时刻已知的从起点到该节点的最短路径。

Dijkstra算法的缺点包括：

1. 无法处理负权重的边：由于Dijkstra算法的贪心策略，它假设所有的边都是非负权重的，因此如果图形中包含负权重的边，Dijkstra算法将无法正确地处理最短路径问题。

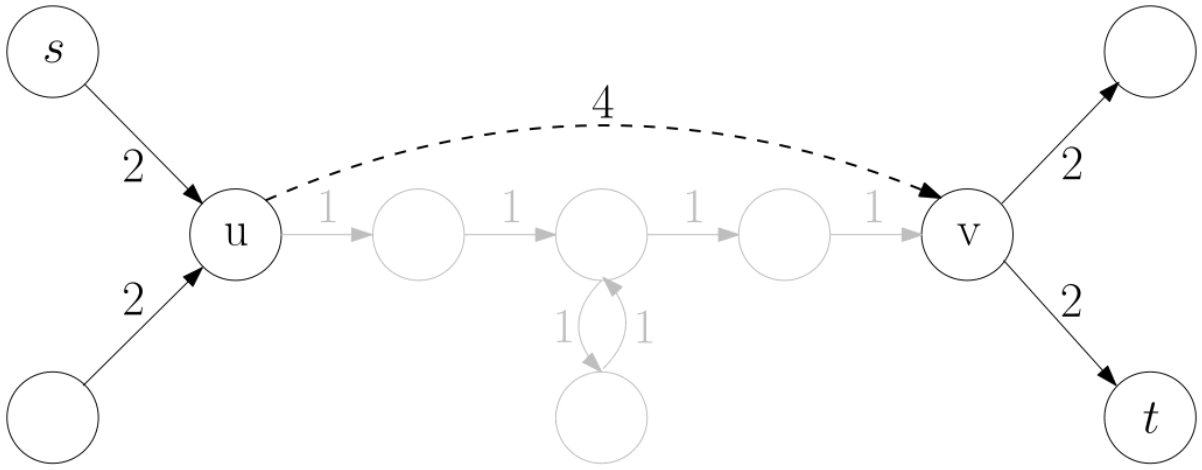
2. 空间复杂度高：Dijkstra算法需要使用一个优先队列来维护节点的距离信息，因此空间复杂度比较高，尤其是对于大型图形。

3. 时间复杂度高：Dijkstra算法的时间复杂度与图形中的节点数和边数成正比，在某些情况下，Dijkstra算法可能会因为太多的节点和边而变得不实用。

知识点三：CH算法

2.3.1 算法背景和原理介绍

收缩层次方法（Contraction Hierarchies）是一种用于在图中寻找最短路径的加速技术。图中的最短路径可以使用Dijkstra算法计算，但考虑到道路网络由数千万个顶点组成，这是不切实际的。收缩层次结构是一种加速方法，经过优化以利用表示道路网络的图形的属性。加速是通过在预处理阶段创建快捷方式来实现的，然后在最短路径查询期间使用快捷方式跳过“不重要”的顶点。



2.3.2 算法流程

收缩层次 (CH) 算法是一种解决最短路径问题的两阶段方法，包括预处理阶段和查询阶段。

1. 预处理：

a. 收缩原始图中的节点和边，得到一个新图G'。

b. 构建两个有向无环图G1和G2，分别包含所有源点可达的节点和所有汇点可达的节点。这两个图是G'的子图，它们包含了原始图中的部分节点和边。

c. 在G1和G2中运行Dijkstra算法，计算每个节点到源点的最短距离和每个节点到汇点的最短距离。同时，为每个节点分配一个层次（level），即节点在图G'中被收缩的次数。

d. 对所有节点按照层次排序，得到一个节点的收缩序列。

2. 查询：

a. 对于查询中的起点和终点，将它们在图G'中收缩成新的节点，得到新的起点和终点。

b. 根据新的起点和终点在收缩序列中的位置，确定它们的层次。

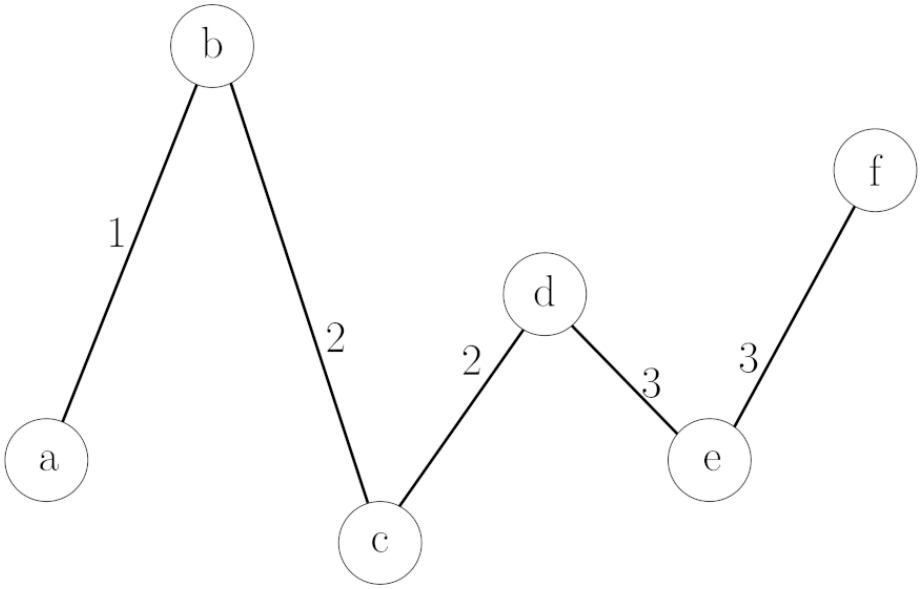
c. 从收缩序列中的起点层次开始，依次遍历层次，更新当前最短路径长度。

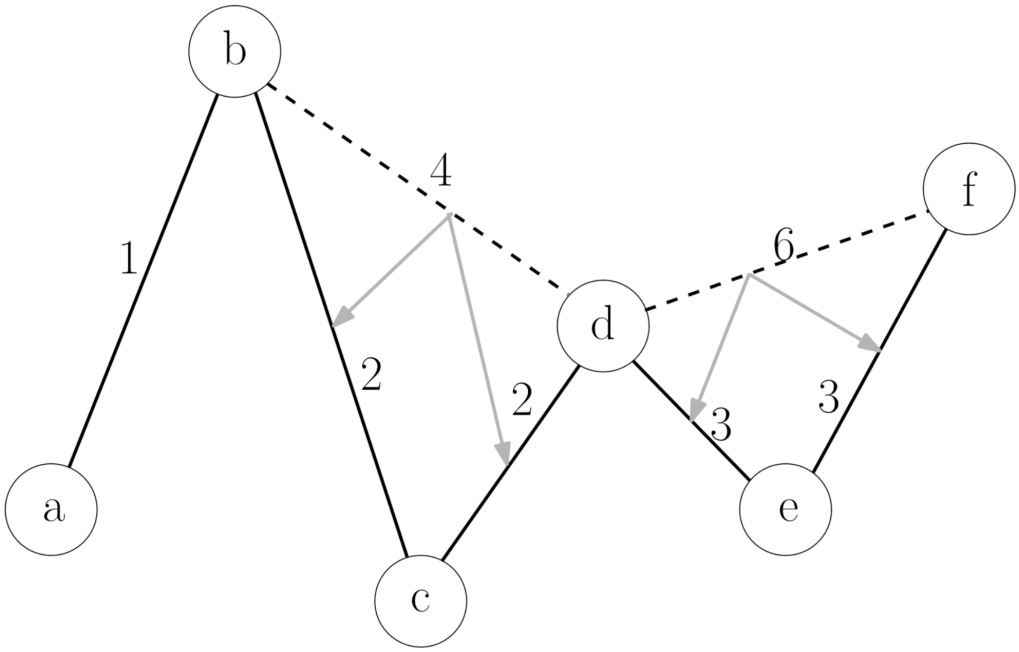
i . 如果当前节点的估计距离小于当前最短路径长度，则更新当前最短路径长度。

ii . 对于当前节点的每个后继节点，计算它到起点的距离，使用已经计算好的距离向量在常数时间内计算出当前最短路径长度。

d. 返回起点到终点的最短路径长度。

CH算法的核心思想是通过预处理将原始图转换为两个有向无环图，并在预处理阶段计算出这两个图上的最短距离向量。在查询阶段，根据起点和终点在收缩序列中的位置，依次遍历每个节点，并在每个节点处使用最短距离向量更新当前最短路径长度。CH算法的时间复杂度为O(n log n + m)，其中n是节点数，m是边数。





2.3.3 输入输出

输入：起点、终点

输出：最短距离

2.3.4 优缺点

CH算法的优点：

1. 预处理时间复杂度较低：预处理阶段的时间复杂度是O(n log n + m)，其中n是节点数，m是边数，这比一些其他最短路径算法的预处理时间复杂度更低。

2. 查询速度快：在查询阶段，CH算法使用预处理得到的最短距离向量，在常数时间内计算节点之间的最短路径长度，因此查询速度非常快。

3. 空间占用少：CH算法在预处理阶段只需要构建两个有向无环图和计算最短距离向量，因此空间占用较少。

CH算法的缺点：

1. 预处理过程需要占用大量内存：由于预处理阶段需要构建两个有向无环图和计算最短距离向量，因此需要占用大量的内存空间。

2. 无法处理动态图：一旦预处理完成，CH算法不能处理原始图的动态变化，例如节点和边的添加或删除。

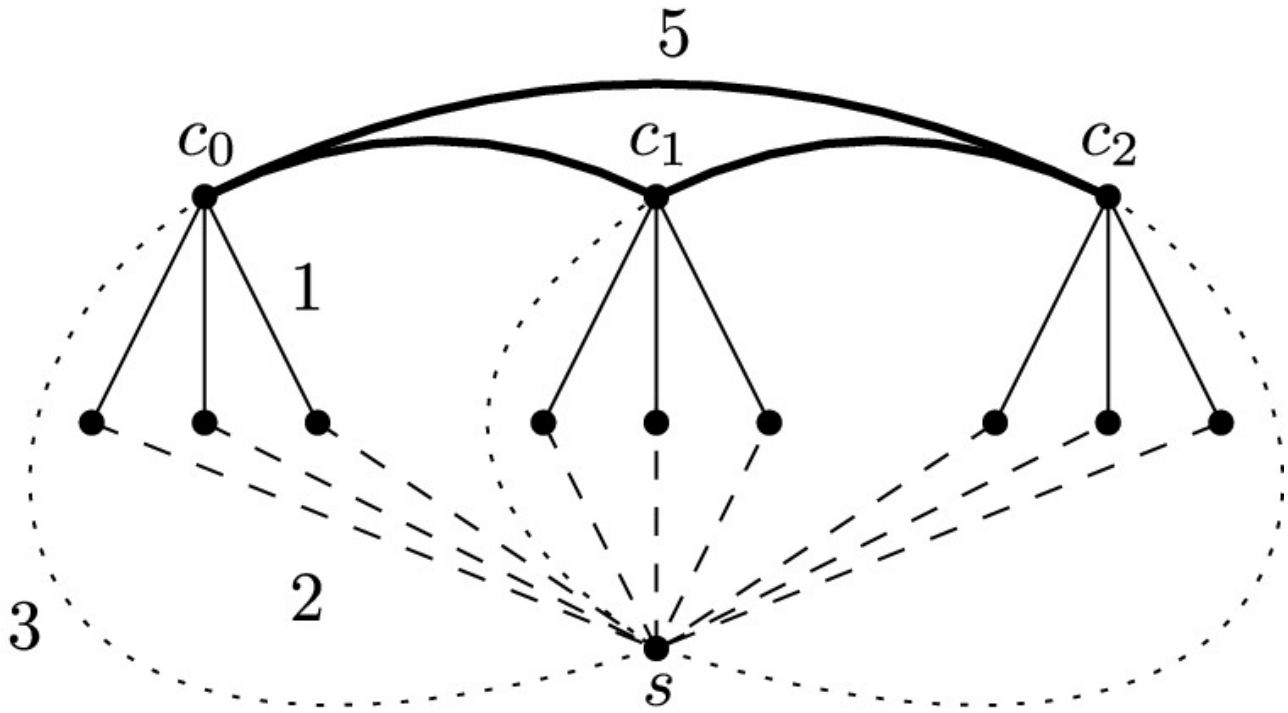
3. 不支持负权边：由于Dijkstra算法无法处理负权边，因此CH算法也无法处理负权边。

4. 精度受限：CH算法使用最短距离向量估算节点之间的距离，因此其结果可能会略微偏差。

知识点四：Hub-Based Labelling索引

2.4.1 算法原理

Hub Labeling算法的主要思想是在预处理阶段，使用一组称为“标签”的数据结构来存储节点对之间的最短路径。这些标签存储在网络中的一组称为“hub”的节点上。Hub节点通常是度数很高的节点，即它们与其他节点之间有很多连接。通过将这些hub节点作为标签存储在预处理阶段中，可以加速查询过程。



2.4.2 算法流程

Hub Labeling算法的流程如下：

1. 构建hub节点集合：选择网络中具有高度连接性的节点作为hub节点。

2. 计算标签：对于每个hub节点，计算其到其他节点的最短距离，并将这些最短距离作为标签存储在该节点上。这可以使用Dijkstra等最短路径算法来完成。

3. 预处理标签：将标签排序并对其进行压缩，以减少存储空间。例如，可以使用前缀压缩等技术来减小标签的大小。

4. 查询过程：对于给定的起点和终点，从起点开始，根据当前节点的标签和相邻节点之间的边权值，估算到达终点的距离。根据估算的距离值，选择下一个节点进行查询，并使用动态规划来修正估算距离，直到到达终点为止。

具体来说，Hub Labeling算法的查询过程如下：

1. 从起点开始，将其加入查询队列，并初始化到起点的距离为0。

2. 对于队列中的每个节点，获取它的标签和相邻节点之间的边权值，计算出到达终点的估算距离。

3. 根据估算距离值，选择下一个节点进行查询。在选择下一个节点时，可以使用各种优化技术，例如选择距离终点最近的节点或使用堆来加速查询过程。

4. 对于选择的下一个节点，根据其与当前节点之间的边权值，计算实际距离，并使用动态规划来更新估算距离。

5. 重复上述步骤，直到到达终点或查询队列为空。

总的来说，Hub Labeling算法的流程类似于其他基于标签的最短路径算法，但是其重点在于选择和处理hub节点，以加速查询过程。

2.4.3 输入输出

输入：起点和终点的大规模图数据集，中间用空格隔开。

输出：无

2.4.4 优缺点

Hub Labeling算法的优缺点如下：

优点：

1. 查询速度快：Hub Labeling算法可以快速查询任意两点之间的最短路径，相比其他最短路径算法，其查询速度更快。

2. 空间占用小：Hub Labeling算法只需要存储每个hub节点的标签，相比其他最短路径算法，其空间占用更小。

3. 可扩展性强：Hub Labeling算法可以根据网络规模和结构的不同进行优化和扩展，例如选择合适的hub节点、使用压缩技术等

缺点：

1. 构建时间较长：构建hub节点集合和计算标签需要耗费较长的时间，特别是对于大规模网络来说，构建时间可能会很长。

2. 前期预处理时间较长：Hub Labeling算法需要对标签进行排序和压缩，这些操作也需要较长的时间。

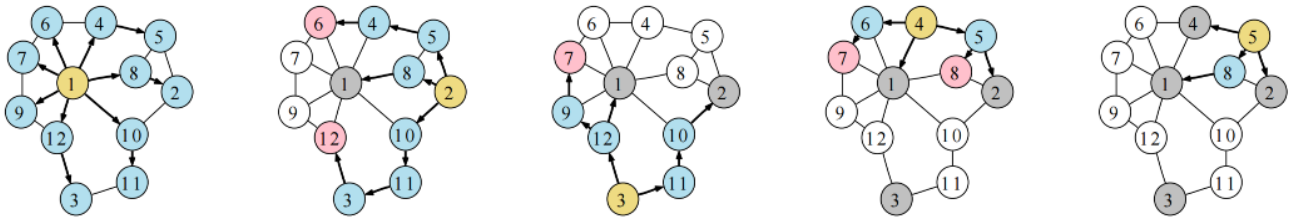
3. 不适用于动态网络：Hub Labeling算法的标签是根据网络结构固定计算的，一旦网络发生变化，就需要重新构建标签。因此，它不适用于动态网络的情况。

4. 对网络结构要求较高：Hub Labeling算法需要选择合适的hub节点，并且要求网络中有明显的连接性，否则可能会影响其查询效率。

知识点五：Pruned Landmark Labeling索引

2.5.1 算法原理

Pruned Landmark Labeling（PLL）算法是一种基于Landmark的最短路径查询算法，它旨在提高在大型网络中查询任意两点之间的最短路径的效率。PLL算法与Hub Labeling算法类似，都是使用预处理的方式来加速查询。PLL算法的核心思想是将节点划分为多个等级，每个等级都有一组标志节点作为Landmark，并对每个节点记录到每个Landmark的最短距离。在查询时，通过跳过一些节点等级，从而减少需要遍历的标签数量，以提高查询效率。与Hub Labeling算法不同，PLL算法使用了更加灵活的节点集合，使得其适用于更加广泛的网络结构。



2.5.2 算法流程

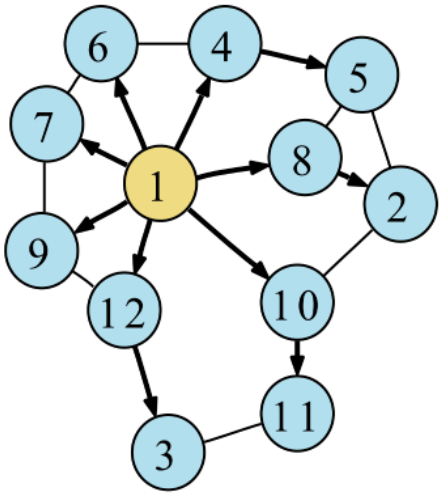
PLL算法的流程如下：

预处理阶段

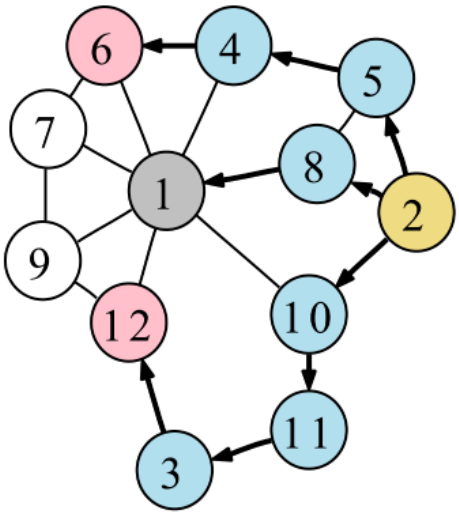
1. 选择一组Landmark节点作为索引。在大多数情况下，Landmark节点的数量越多，PLL算法的查询效率越高，但是预处理时间和空间开销也会相应增加。

2. 计算每个节点到所有Landmark的最短距离，并将这些距离作为该节点的标签。

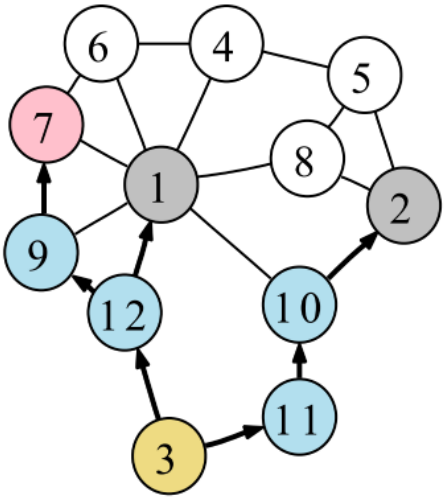
3. 对每个节点的标签按照距离从小到大排序，并对其进行压缩，以减少空间占用。



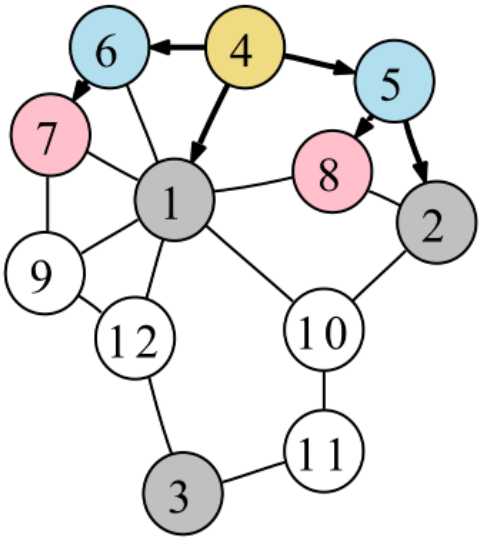
(a) 从顶点1到第一个BFS。我们访问了所有的顶点



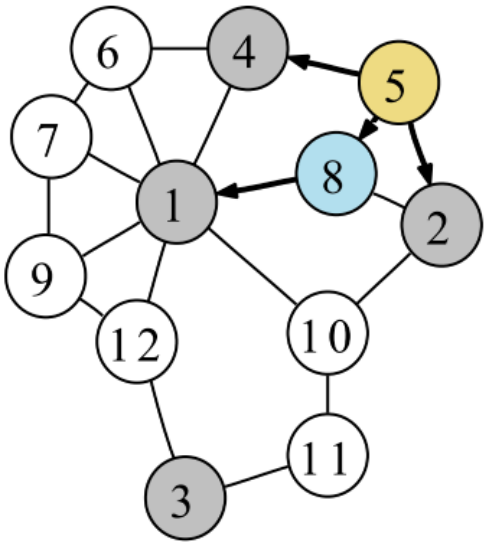
(b)来自顶点2的第二个BFS。我们没有向五个顶点添加标签。



(c)来自顶点3的第三个BFS。我们只访问了顶点的下半部分。



(d)来自顶点4的第四个BFS。这次我们只遍历了上半部分。



(e)来自顶点5的第五个BFS。搜索空间甚至更小。

查询阶段

1. 根据源节点和目标节点之间的距离选择跳过一些节点等级，并在剩余的等级中查找标签，以缩小需要遍历的标签数量。

2. 在剩余的标签中，选取每个节点到源节点的距离加上其标签距离最小的节点作为候选最短路径，并在剩余的路径中找到最短路径

2.5.3 输入输出

输输入：起点和终点的大规模图数据集，中间用空格隔开。

输出：无

2.5.4 优缺点

PLL算法的优缺点：

优点：

1. 查询速度快：PLL算法能够在大型网络中快速查询任意两点之间的最短路径。

2. 空间占用小：PLL算法只需要存储每个节点的标签，相比其他最短路径算法，其空间占用更小。

3. 对网络结构要求较低：PLL算法使用灵活的节点集合，适用于更加广泛的网络结构。

缺点：

1. 构建时间较长：PLL算法需要构建标签和预处理路径，因此构建时间可能较长。

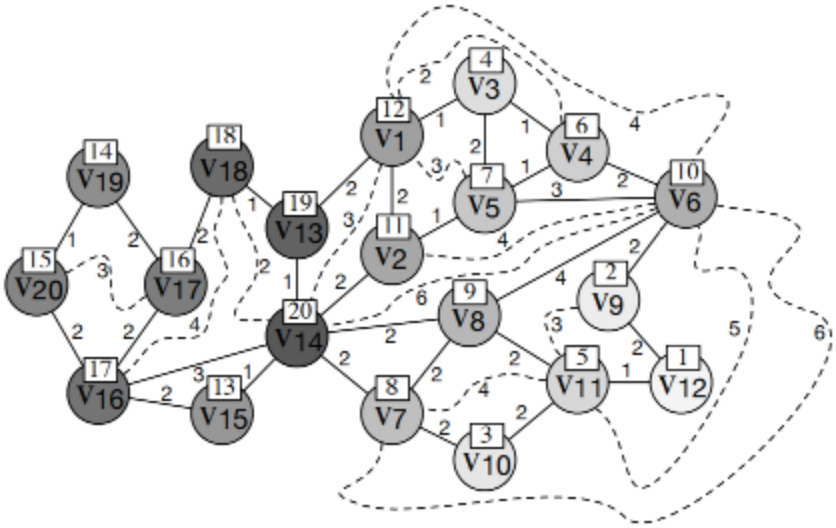
2. 对内存和处理能力的要求较高：PLL算法需要处理大量的标签和路径，因此对计算机的内存和处理能力要求较高。

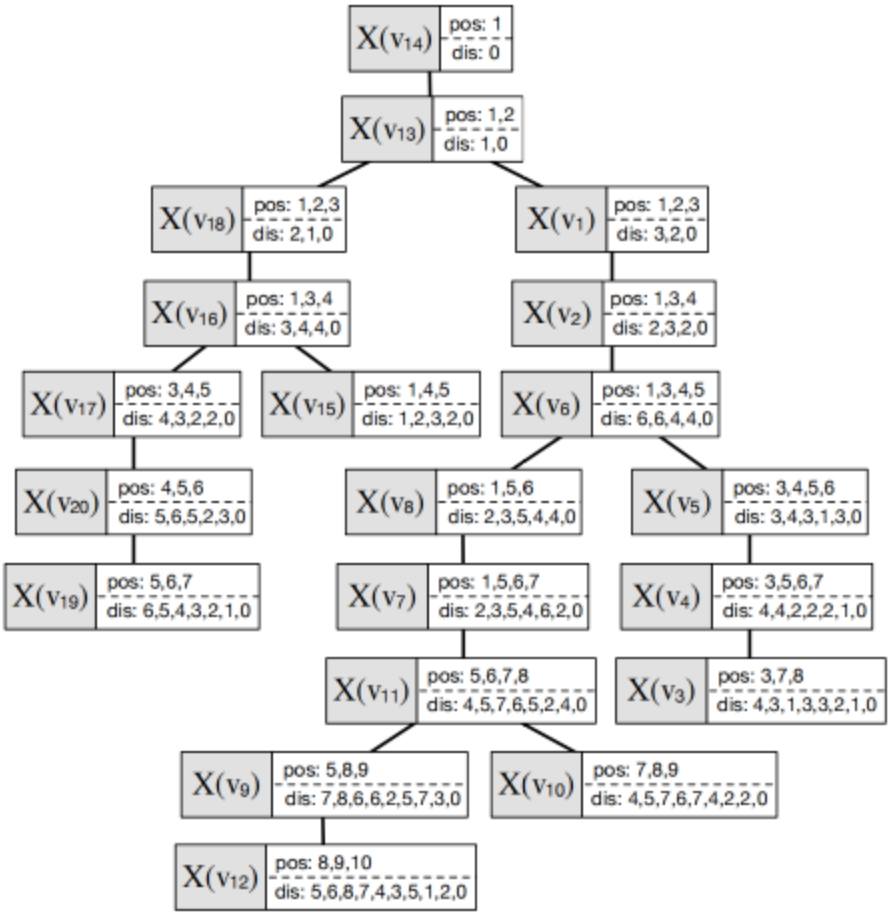
3. 不适用于动态网络：PLL算法的标签和预处理路径是固定的，一旦网络发生变化，就需要重新构建

知识点六：H2H

2.6.1 算法背景和原理介绍

传统的搜索算法如Dijkstra算法在搜索空间较大时效率较低，因此研究转向基于索引的方法。现有的基于索引的解决方案可分为基于层次结构和基于跳数的方法。然而，基于层次结构的方法对于较远距离查询需要较大的搜索空间，而基于跳数的方法对于较短距离查询存在计算浪费。

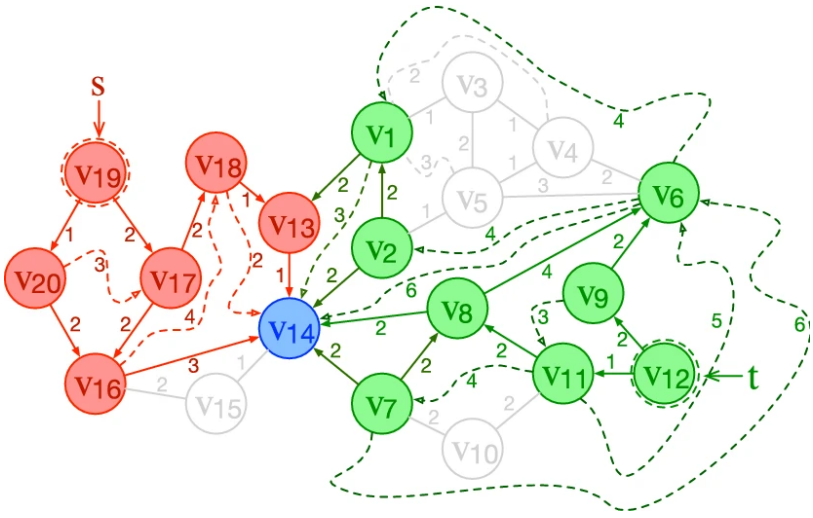




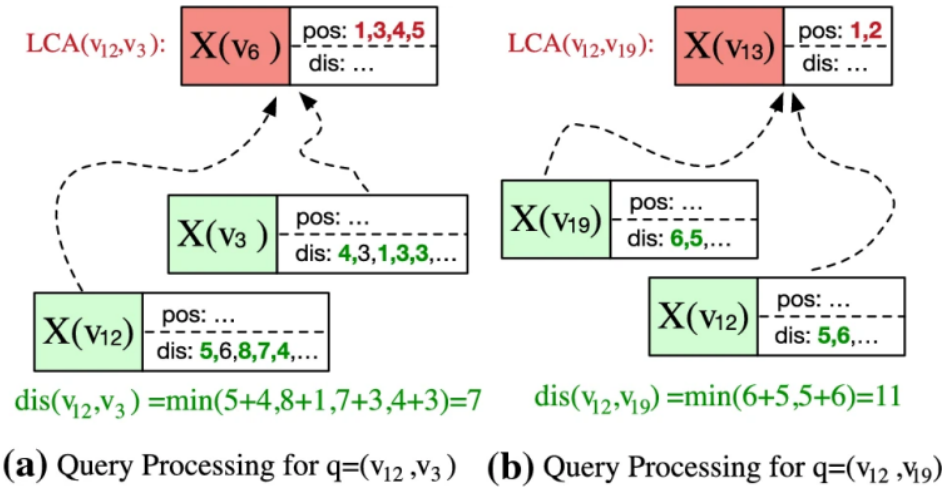
2.6.2 算法流程

H2H通过树分解创建了 2–hop 的层次结构。具体来说，它是通过以下三个步骤构建的：

1. 树节点的形成：按照与 CH的图收缩相同的过程(详细见知识点三)，每个收缩的顶点v 形成一个树节点 Xv，包含它的所有邻居$NG'(v)$和v 和 $u \in NG'(v)$之间的快捷方式。

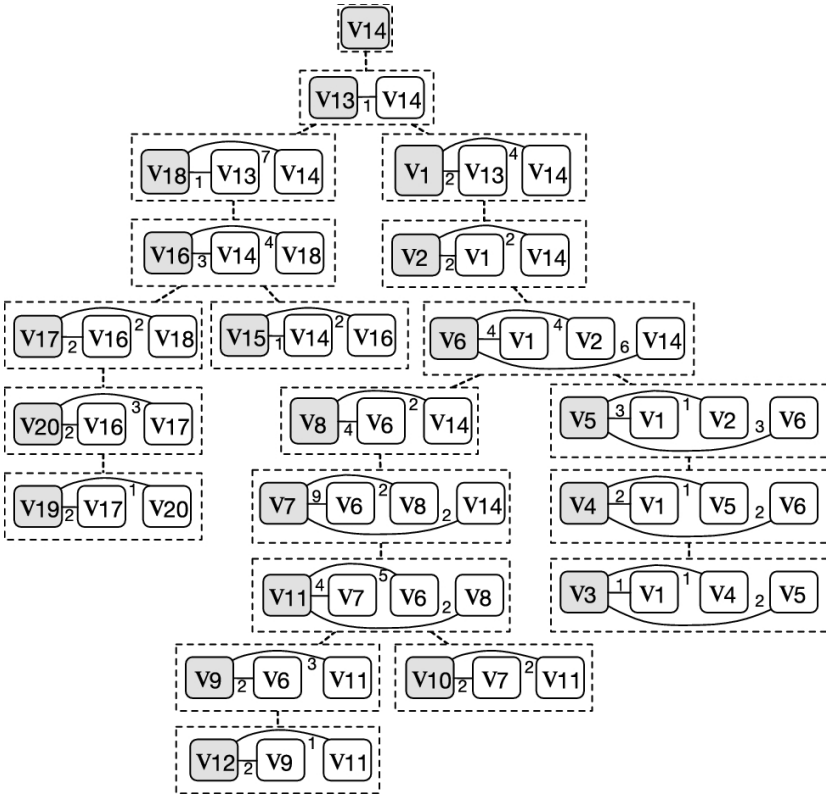


2. 树的构建：通过将Xu设置为Xv的父节点来连接树节点，其中u在Xv中除v之外的最小order。



(a) Query Processing for q=(v_{12} ,v_3) **(b)** Query Processing for q=(v_{12} ,v_{19})

3. 自上而下的标签分配：$L(v)$包含从v到其所有祖先的距离，计算为min{$e(v,u)$ + $d(u, ai)$},∀u∈Xv。因为赋值是从根到叶进行的，d (u, ai) 已经在祖先的标签中得到了。 $q(s, t)$ 可以通过 min{d (s, vi) + d (vi , t)|∀vi∈Xa} 得到，其中Xa是Xs和Xt最小公共祖先(LCA)树节点。



2.6.4 优缺点

知识点七：HL索引

2.7.1 算法原理

Highway Cover Labelling

2.7.2 算法流程

2.7.3 算法实现结果

2.7.4 优缺点

三、演示验证

3.1 运行环境搭建

四、实验环境

操作系统：

平台软件：

所需依赖：

五、界面展示