**Data-Dirven Hierarchical Hub Labeling for Fast Shortest Path Queries on Large Road Networks**

**Abstract**

最短路径查询在许多的应用中都作为核心的元素，因此响应时间是至关重要的因素。目前已有的方法大多都是在预处理过程中建立索引和保存辅助信息来实现在线的查询，为了减小预处理和查询的时间和空间开销，会针对网络的特征提取出不同形式的层级结构（Hierarchy）。但这些方法大多只考虑了网络节点（Vertex）或者路径（Path）在路网结构特征和最短路径构建中的重要性。而我们通过观察每天产生的大量查询数据发现，少部分的节点被大量查询，而很多节点查询次数很少甚至为0，即满足二八定理，大约百分之二十的节点的查询次数覆盖了总查询次数百分之八十。因此我们提出了一种将查询频率作为Hierarchy分层指标的重要影响因素的Data-Driven Hierarchical Hub-based Labeling Model，根据历史数据以及对未来时段节点查询分布的预测，通过减小高频查询点的访问开销，构建出以Hub Labeling为基础的层级最短路径查询索引结构，并使用缓存和批量查询，在更小的存储开销下大大加快查询的响应速度。

**Keywords**

Shortest paths, query processing, query frequency, contraction, hub labeling

**1. Introduction**

由于最短路径查询的广泛应用，每秒钟都会产生大量的查询和访问次数，这对查询响应速度提出了更严峻的挑战。最经典的算法是Dijkstra 算法，可以在接近线性的时间内回答任何最短路径的查询，时间复杂度为，但当起点和终点相距较远时搜索范围增大导致其无法应用于大型网络图。目前主要是基于无向有权图进行相关算法的介绍，我们的方法可以很容易地应用到有向图。



Figure 1: Overview of the Data-Driven hierarchical labeling algorithm framework

如Figure 1所示，我们整个算法的处理流程为：首先通过历史数据分析预测出未来某个时段路网中节点的查询次数（频率），然后使用我们的排序算法对节点进行排序定义出Hierarchy，然后利用点的重要程度对路网进行收缩提取出Overlay Graph，再使用Pruned Landmark Labeling（Push）算法构建Hub Labels，并采用优化手段对Labeling进行压缩减小构建开销和存储开销，在此基础上根据查询频率提出有效的缓存策略，并能够针对不同的时间分片进行Labeling的更新和重建。

**2. Preliminaries**

**2.1 2-hop Labeling**

2-hop Labeling 是给每个节点（无向图）都赋予一个集合，对于Label集合中的每一个都计算出其到的距离，即，并存储为，其中被称为hub。这样，对于图中任意两个点都可以使用它们共有的Label hubs计算出最短距离：

但对于任意两个节点需要满足Cover Property：即至少要包含最短路径上的一个点。对于有向图则需要分别构建forward label 和reverse label ，2-hop Labeling可以有通过sweep over遍历起点和终点的Label Vertices List得到最短路径，查询操作类似于merge sort，时间复杂度为，因此要解决的问题就是使得最小。Cohen et al.最早提出了一种基于set-covering的贪心算法，选择使得最大的，即能覆盖的最短路径和增加的label size开销的比值，能在平均时间内(优化可达)产生a distance oracle，即不超过最优大小的。即使大量的基于该框架的改进算法被提出，但依然无法应用于大型网络图。

**2.2 HHL（Hierarchical Hub Labeling）**

我们方法主要基于的思想是Abraham et al.提出的改进方法HHL。它使得2-hop labels可行并对于大型图更加scalable，即通过限制能增加到Labels的点需要满足的属性来限制labeling的选择。HHL定义了一种优先级：对于两个不同的点，当且仅当包含，则，如果满足偏序关系，则称Labeling是hierarchical，并且可以延伸为全序关系(the order-extension principle)。对于一个给定的全部点的排序，被定义为一种双射 ：，如果满足，且 (即 ranks higher than )，则称Label是hierarchical的。为了使得到的Label Size最小，又提出了CHHL（Canonical Hierarchical Hub Labeling），即对于任意的，， has the highest rank，和中只包含上具有最高rank的节点，即没有，使得。因此最佳HHL问题就变成了两个子问题：(i)找到最佳的使得label size最小;(ii)根据给定的计算出CHHL。

**2.3 PLL（Pruned Landmark Labeling）**

已经被证明，要找到最佳的HHL是一个NP-hard问题，而它的第一个子问题即找到最佳的排序同样也是一个NP—hard问题。PLL算法的提出是一个巨大的突破，对于一个给定的，它能够有效地计算CHHL。

**2.4 CH（Contraction Hierarchy）**

CH是一种基于预处理的算法，而不是构建Label的算法。CH同样也需要指定全局的节点排序，然后根据此顺序来依次对每个结点进行检查。当处理到时，暂时去掉已经操作过的节点，剩下的点构成，。对于每一个起始点及和终点及，进行witness search（Dijkstra）来进行计算，需要时间为，如果，则添加一条Shortcut，。CH最终的输出是增加了辅助结构的图，是Shortcut集合以及中间节点的编号，即在处理哪个节点时新增的Shortcut，以便于全路径展开。

CH的是通过双向Dijkstra算法进行查询，当两个方向遇到整个路径上rank最高的点时终止搜索。CH的查询性能以及同样取决于节点的排序，已有的排序方法基于图的局部特征进行在线启发式估计，会增加条Shortcut，标准的CH时间开销为，最优方案减小到。由于搜索只需要处理rank更高的点及其边，速度得到提升，因此CH常常作为一些HL方法的预处理，可以加快构建Label的过程。

HHL是一种Top-Down（TD）方法来选择节点，从最重要的到最不重要的点，和CH刚好相反，CH是一种bottom-up的方式。

**3. Problem Definition**

Table 1: Summary of notations

|  |  |
| --- | --- |
| **Notation** | **Meaning** |
|  | a graph with vertex set and edge set |
|  | number of vertices |
|  | Number of edges |
|  | the exact distance of shortest path from to (sum of weight) |
|  | may not equal to in the context |
|  | a sequence of pairs , where is a hub |
|  | label size (the number of pairs) |
|  | shortest path query from source vertex to target vertex |
|  | The shortest path query result, a list of vertices |
|  | the size of (number of vertices) |
|  | importance of vertex |
|  | query times of |
|  | query frequency of vertex |
|  | betweenness centrality of vertex |
|  | hubs set (vertices) |
|  | query cost of vertex |
|  | the rank of , 0 is the highest |
|  | the total vertex order |

首先我们暂时只考虑无向图，把查询对起点和终点同等看待，基于之前的分析，预期的查询时间cost取决于两点：(i)每个点的查询频率；(ii)每个点的查询访问代价。为了衡量query performance，我们定义了Query cost function如下所示：

其中是指查询的开销（Expense），从前面的分析可以得出，即查询开销就等于的Label Size。，其中是在某一时段的查询次数，是在时段所有节点查询次数之和。

这样，我们的问题就变成了使得最小，即Query cost最小。

LEMMA 1. Optimal subpath property；

最短路径包含最短路径，如果，。一般地，，则有，，，。

**4. Overlay Graph Contraction（删除低频点的图分解算法）**

**4.1 节点隔离**

设计了结点隔离、图分解、生成隔离点label和隔离点查询的算法，主要思想是从原始图中将低频点隔离，删除隔离点组成的隔离区，生成overlay图，然后在该图上再通过改order等其它方法建立label，优化高频点查询速度。被隔离的结点查询速度会减慢，根据曼哈顿路网数据，低频点查询量非常少，部分低频点在高峰期查询量为0，这些低频点查询速度减慢并不会对总体查询时间造成太大影响。在不恢复隔离点全局label的情况下（仍能保证查询正确性），隔离20%结点时，全图平均label大小减少约15%-25%。

主要启发来自于CH的shortcut，结点隔离的过程与之类似，但不需要考虑该shortcut是否为局部最短路径。通过一个例子来介绍结点隔离的思想和方法。

O

A

C

B

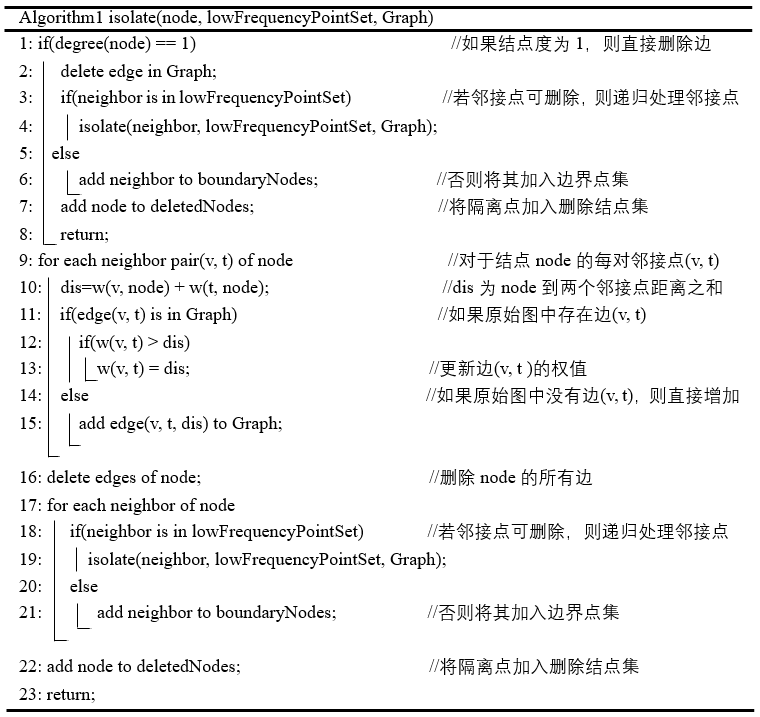
D

E

F

Figure 2. 结点隔离示意图

图2中实线为原始图中存在的边，图中点画线A-D-B为原始图中结点A到B之间的最短路径。假设度为3的结点O是需要隔离的低频点，为保证查询正确性，在删除边OA、OB、OC的同时要保留这三条边的信息。对于边OB、OC，隔离结点O的同时，需要增加一条边BC，其权值等于边OB与OC权值之和，如果B-O-C为原始图中结点B到C的最短路径，由于隔离结点O时增加了边BC，因此仍能得到BC的最短距离。对于边OA、OC，由于原始图中存在边AC，则只需比较AC的权值与边OA、OC的权值之和的大小，取小值作为AC的权值。对于边OA、OB，同样增加一条边AB，但显然AB不是最短路径，但隔离结点O后，对生成的overlay图建label的push过程会保证得到A-D-B的最短距离，在查询E-F最短距离时，仍能得到正确结果。

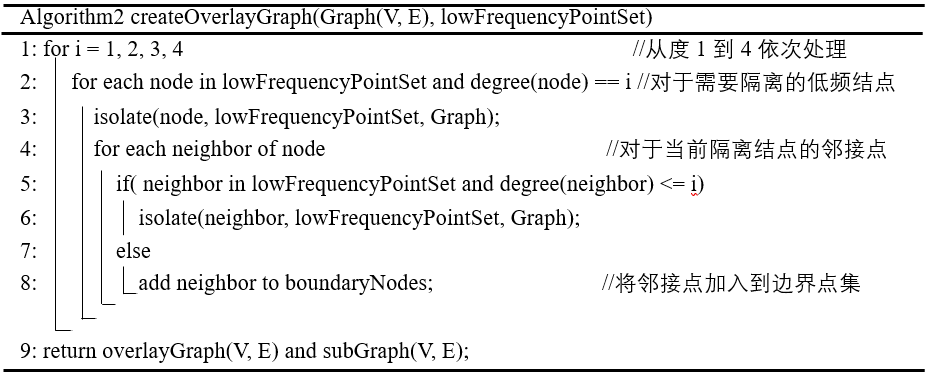


算法1给出结点隔离算法，该算法是一个递归过程。对于度为1的结点（1-8行），可直接删除边，递归处理其邻接点。对于度大于1的结点，依次处理其邻接点对(v, t)（9-15行），初始化dis为node到其邻接点v和t距离之和（10行）。若原始图中存在边(v, t)，则将权值与dis比较后更新为最小值（11-13行），否则直接增加边(v, t, dis)，然后删除node的所有边（16行）。递归处理所有邻接点（17-21行）。

结点隔离过程与CH最大的区别是不需要进行结点的全局排序，也不需要进行局部dijkstra来计算shortcut是否为局部最短路径，在保证overlay图查询正确性的情况下，大大缩短结点隔离建立overlay图的时间。

**4.2 图分解**

图分解过程的实质是重复进行结点隔离，邻接的隔离结点形成隔离区。

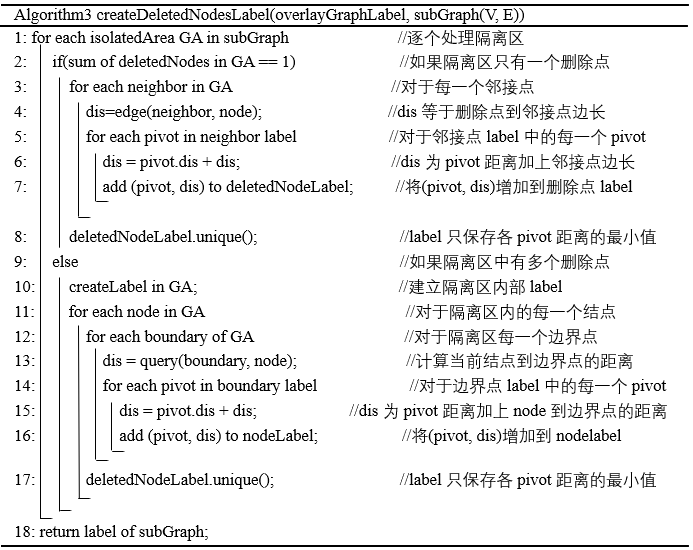


按照度从小到大的顺序，依次隔离低频点，目前代码实现的隔离点的最大度为4。算法2中，第3行调用isolate函数处理当前需要隔离的低频点，第4-8行处理当前隔离点的邻接点，如果邻接点也是可以隔离的点，则直接调用isolate函数，否则该点成为边界点，将其加入到boundaryNodes中。

代码实现时为了逐项验证正确性，生成overlayGraph和各区域的subGraph是分开构建的。处理后的overlayGraph平均度会增大，根据隔离点的数量不同，平均度约为3到6左右。

**4.3 生成隔离点Labe**l

生成隔离点label分为两种方式：一种方式是只生成各隔离区域内部的label，每个隔离点label中增加到所在隔离区边界点的距离（实验验证得到所有边界点数量小于等于4）；另一种方式是将各隔离区域边界点label“合并”到各隔离区域内所有隔离点。第一种方式存储的label非常少，目前实验的平均label大小在3-7左右，但查询时最多需要比较4×4个label（设计了算法提升查询速度）；后一种方式由于利用边界点生成了“全局”label，查询速度与原有方法相同，但平均label大小较之前有所增大。这里给出第二种方式的查询算法，第一种方式实质是将“合并”过程放到查询阶段进行。

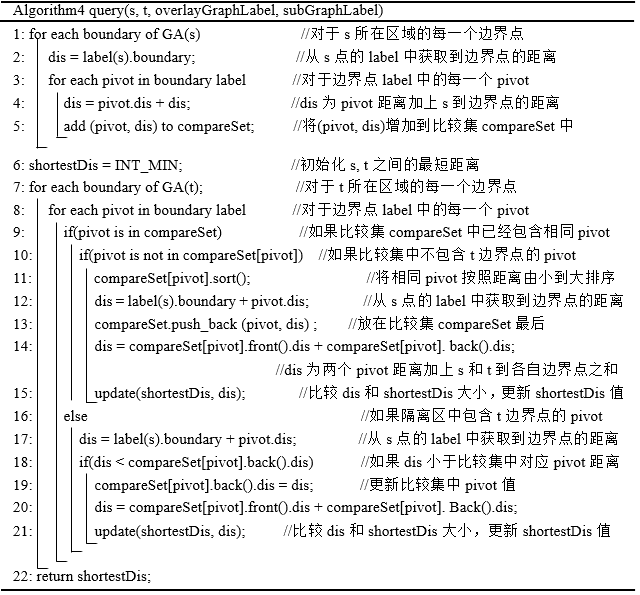


算法3给出了将边界点label“合并”到隔离区内结点的过程。如果隔离区内只有一个隔离点（2-8行），则其邻接点就是边界点，只需在邻接点的所有pivot中，加上隔离点到邻接点的边长，然后取最小值即得到隔离点的label。如果隔离区有多个点（10-17行，曼哈顿所有隔离区中最多有42个点），则先对隔离区域内部建立label，利用label查询每个结点到边界点的距离，再按照类似一个隔离点的处理方法，即得到隔离区内部所有结点的label。

前期代码实现过程中，为了验证正确性，各区域内实际使用dijkstra求出了区域内所有结点间的最短距离作为label，因此label大小较大，后续再编写使用betweeness等方法建立区域内部label的代码。

**4.4 隔离点查询**

隔离点的查询与隔离点label生成的方法相对应，也分为两种情况。第一种方法对应隔离点label只包含隔离区内部信息，则查询过程实质是将边界点label的“合并”过程放在查询阶段，因此查询速度较慢，但label大小较小；第二种方法对应隔离点label已根据边界点label生成，其查询过程与普通的label查询过程完全相同，查询速度较快，但label大小较大。这里给出第一种查询算法。



查询过程的实质是1.3中生成隔离点label的过程。建立一个二维数组compareSet，将s点所在区域所有边界点label的pivot加上s到边界点的距离放入compareSet中（1-5行）。初始化s-t最短距离shortestDis（6行），对于t所在区域的每一个边界点boundary，若boundary的pivot与compareSet中的相同，则将compareSet中已存的pivot按照距离大小排序（11行），取最小pivot距离加上boundary中该pivot值与t到boundary距离之和为计算的当前距离dis（14行），将dis与shortestDis进行比较更新。若boundary已经存有s和t边界点的pivot，则比较pivot大小，计算dis后取最小值与shortestDis进行比较更新。

设overlayGraph结点的平均label大小为n，查询结点所在区域边界点数量为b，则在不生成隔离结点“全局”label的情况下，使用原始查询方法时间复杂度为O(b2×2n)。在使用上述查询算法情况下，利用空间换取查询时间，避免了不相同pivot的比较运算，设s和t所在区域边界点label中相同的pivot数量为m，排序运算时间复杂度为blog(b)m，不考虑赋值运算只考虑比较运算的情况下，总的时间复杂度为O(blog(b)m+2bm)，其中b≤4，m<n。此算法还未进行代码实现。

**5. Ordering Algorithm**

对已有的方法经过调研发现，都是在试图寻找一种层级构建方法，vertex-centric或者path-centric。因此找到网络图中更具有代表性的节点或路径对于加快预处理和查询时间、降低存储空间开销都是至关重要的。

基于Hub Pushing Algorithm的算法会造成更为重要的节点（Higher Rank），即先作为BFS起点的节点Label Size更小，因此构建出的unique canonical hub labeling的Size与给定的节点order 保持一致。前面已经分析过在不考虑对已生成的Labels进行优化的前提下，查询的时间开销为，即两个点的Label size的线性相加。因此最终的query performance实际上由给定的order 决定，construction time同样也是和Label Size是成线性相关的。因此找到一个最佳的排序方式是至关重要的，能使得构建和查询时间开销得到提升。而query performance又和节点的查询频率有关系，因此如果能尽量减小高频查询点的Label size，以适当增加大部分低频点的Label size为代价，也可以达到大大提升查询速度的目的。

**5.1 线性组合**

目前广泛应用的排序strategies主要包括基于Degree，Betweenness，Closeness，Significant path，这都能表明节点在网络图结构中的重要性。Degree：有更大度的点与其它点会有更强的连接，因此会有很多最短路径经过这些点以及邻近的点，由于路网图的节点都较小，以Manhattan图为例，统计发现节点的度大部分都在2-4之间，在规模图上当两点距离较大时，以度作为指无法选出具有代表性的节点，因为度只表示局部的重要性没有包含全局信息；Closeness centrality：反映的是到达其他节点的难易程度，也就是到其他所有结点距离的平均值的倒数，精准的计算需要的时间开销，因此采用随机采样的方法进行计算，实验表明Closeness strategy和Degree strategy效果差别不大，在big diameter。Significant path：选择出路网中具有代表性的Path，即有更多的最短路径经过的，在Significant Path上的点也是具有高重要性的点，实验表明该方法的效果不如Betweenness。Betweenness：是指最短路径经过的数量，因此高Betweenness的节点即更重要的点。

实验表明，设置Betweenness参数大的效果更好，基于的是算法中如果能够在刚开始就选择出更重要的点，即具有更高的点能够覆盖更多的最短路径，那么后面的节点就可以被剪枝（Pruned），不再添加到已经能根据已有的Label Hub查询到最短路径的的和中。这样，就能达到使得所有节点的Label Size更小的目的。但考虑到我们的最终目的是要牺牲部分低频点的Label Size，从而减小高频点的Label Size来提升查询效率。因此，查询频率也是反应节点重要性的关键性指标，基于这个动机，我们提出了一个更系统的简单却有效的解决方案：

其中是节点的重要性（Importance），值越大，越小，即优先选择作为Push算法的起始节点；，，对每项指标进行归一化处理。通过实验绘制图像找出使得查询评价函数值最小的参数值和，初步实验粗略选择的最佳参数能达到使得所有节点的Average Label Size增加2%左右，查询评价函数提升14%左右。

这是一种Top-down的使用先验（a-priori）参数和指标值进行排序的方法，初步验证表明和是具有鲁棒性（Robustness）的，对于不同的查询分布以及不同的路网图参数最优值波动范围很小。

**5.2 启发式贪心算法**

在排序过程中，需要选择下一个节点添加到中，是已经被选作为Hub来进行Push算法的节点集合，初始化为空，对于任意的，满足，即，中的下标表示。新选择的节点就是，即要使得以下比值最大：

其中是指选择节点作为能够覆盖的Uncovered最短路径的数量，指already covered最短路径数。

的值越大表明该点能够覆盖更多的未覆盖的最短路径，这有利于之后的Pruned过程，使总的Label Size更小。而越小则表明能使总的查询代价越小，即better query performance。

该算法的实现可以采用离线（off-line）和在线（on-line）计算。在线计算方法和SHP算法类似，即我们在建Label过程中边建Label边进行排序，在每个迭代过程中贪心地选择使得比值最大的Significant Vertex作为下一个进行Push构建的起始点。由于增加了排序比较，建Label的时间开销会比提前给定Order的方法增大。

离线计算方法和采用随机采样计算Betweenness的方法类似，也可以直接用Betweenness进行近似估计，由于网络图是static的，因此只需计算一次，时间开销比在线计算大大减小。

**5.3 优化**

分段processing：对于我们来说，具有更高的点对于构建和查询的时间和空间开销影响更大，因此我们可以进一步缩减开销，用于寻找那些排在前面的更重要的点。设定一个阈值（Division），将更重要的点和不太重要的点进行分割：

满足的节点采用更加耗时的排序方法进行排序，当时采用更加简单快速的排序方法，例如直接用度排序。

Tie-breaking：当节点的排序权值出现相同或相近的情况时，采用加上一个随机值的方法，经验证能达到更好的tie breaking效果。

**6. Time dependent更新和重建**

**6.1 时间分片的粒度**

将一天之内分为四个时段：早高峰、午高峰、晚高峰和平常时段，由于交通流量具有潮汐式的特征。早高峰和晚高峰查询高频点比较接近，查询反向相反。

**6.2 自适应更新**

对于查询点及其频率的分布情况进行简单的数学统计，利用相似度计算方法，如一范数、二范数或者余弦相似度，并设定阈值，当高频点分布的差值时就进行更新和重建，具体选择更新和重建还需要进一步细化时间开销后决定。

**7. 查询优化**

**7.1 Rearrange labels**

默认的Labels是按照Hub(Vertex)的rank顺序从高到低进行排序的，这基于的是当两个点距离较远时它们的最短路径更可能经过更为重要的点，即rank更高的点，因此按照优先级进行排序更有利于提高命中率。

对于查询速度的评估，目前都是默认按照正比于Label Size，即需要遍历完整个Label才能保证求出最小距离。我们可以改变Hub的顺序，设置一个，即sentry hub，不需要遍历完所有的Labels，只需在之前就能保证查找到最小距离，即提高命中率。但要找到，需要增加预处理时间的开销。

**7.2 Batch/Shared Query Execution**

减小内存访问，负载均衡。基于我们的分区，对同时段的并发高频点间的访问，我们可以采用聚类将其映射到SuperNode上，采取批处理的形式以加快响应速度，减小内存访问开销。

**8. Cache**

**8.1 缓存内容**

由于基于Hub Labeling的Shortest-path distances查询可以达到很快的速度，而要查询具体的全路径的查询结果用表示，即一系列包括起始点和终点的节点list：，，。需要依次进行展开，这将会消耗比只查询距离更多的时间。利用查询主要集中在某些Vertex pairs-，对于经常查询的高频点及其最短路径建立缓存。

**8.2 缓存策略**

由于需要存储最短路径上经过的每个点，因此我们的缓存策略就是使得per bit缓存效益最大化，并构建了Cache benefit Model。表示已有得缓存，基于Lemma 1，表示能够回答的查询，则整个缓存能够回答的最短路径查询。当我们要选择下一条路径加入时，不能仅仅只考虑查询频率，因为可能存在已有的使得，即已经可以通过已有的缓存进行应答，选择作为缓存是冗余的。为了使得Cache Benefit最大化，提出了一种贪心算法—在每次选择缓存路径时，使得Incremental benefit-per-size of path最大化：

其中指的是查询开销，，指的是查询对的查询频率

可以用每个点的查询频率进行估算，。

**9. Experiments**

**9.1 实验设置**

实验环境：C++, Ubuntu 18.04，AMD Ryzen 7 PRO 4750G with Radeon Graphics，1 processor and 8 cores，16GB of main memory。

Dataset：Manhattan road network:4360，9537，无向有权全连接图

根据历史查询数据随机得到的三个间隔的点查询次数（值最大查询次数，指平均查询次数，指查询总次数）：

（1）,,；

（2）,,；

（3），,,。

**9.2 Vertex ordering strategies**

实验目的：分析对比三种已有的节点排序方法和只考虑查询频率的排序方法。

实验方法：分别使用基于Degree，Betweenness，Significant path，Query frequency的策略对节点进行全局排序，然后使用PPL算法构建Labels。利用随机选取根据历史查询数据得到的时间间隔分别为的节点查询次数，分析不同排序方式的查询性能。

实验结果：

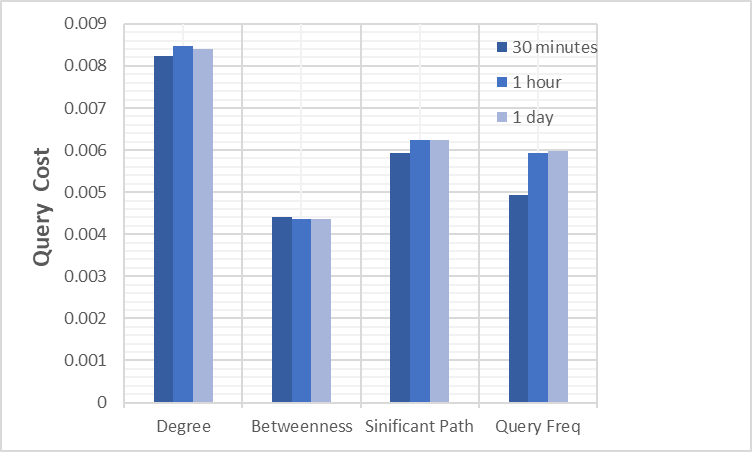


Figure 3: Query cost evaluation

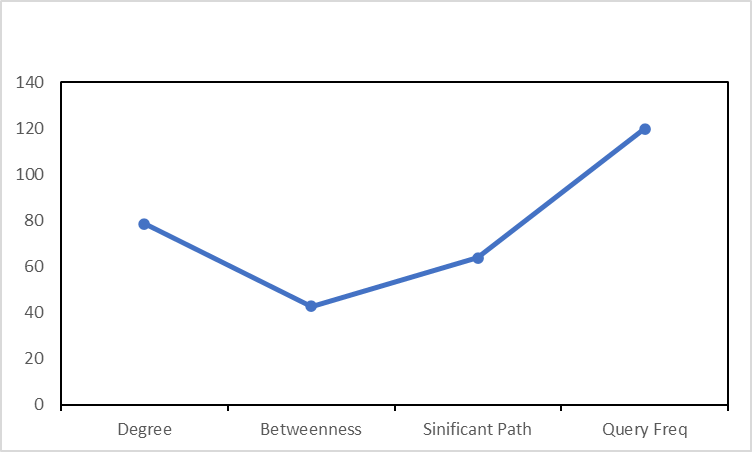


Figure 4: Average Label Size

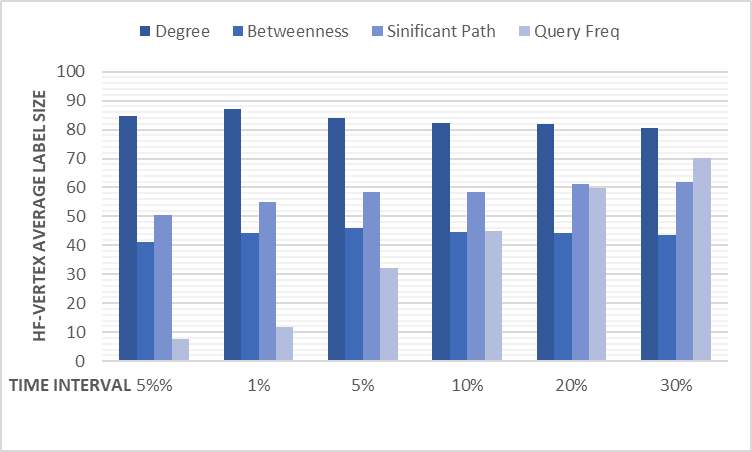


Figure 5: 高频点Average Label Size

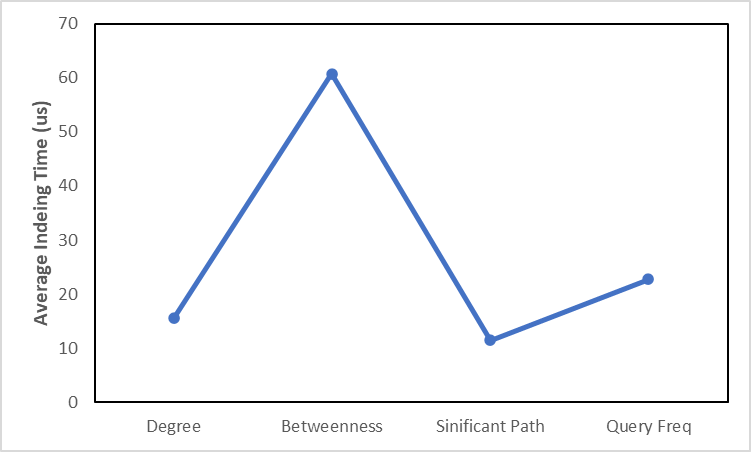


Figure 6 Average Indexing Time of Labels

从实验结果可以看出，基于Betweenness的排序方法拥有最小的平均Label Size以及最佳的查询性能，但由于计算节点的Betweenness时间开销大，因此相较于其它方法构建Label所需时间也最大。直接基于查询频率的排序策略的查询性能仅次于Betweenness，这是因为按照这种排序方式高频点的Label Size大大减小了，但同时也造成了全部点的平均Label Size比基于Betweenness增大了两倍。基于前面的分析我们知道，由于构建时间和Label Size也成正相关，因此时间开销也是无法接受的。基于Significant和Degree的排序方法查询代价更大，因此查询性能也较差，但这两种排序方式的时间开销较前两种小。

**9.3 对比试验**

实验目的：基于上一个实验的实验结果，我们准备采用Betweenness和Frequency

两个指标来共同决定点的重要性排序，并分析寻找最佳的参数配置。

实验方法：采用我们自己定义的LHP算法和基于Degree的DHP算法和、基于Betweenness的BHP算法进行对比。

实验结果：

LHP VS DHP

Table 2: LHP VS DHP analysis result

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Compare to Degree Ordering (TimeInterval:30 minutes, hfRate:20%) | | | | | | |
| 实验次数 | Degree Coefficient | Freq Coefficient | Betweenness Coefficient | 高频点平均Label Size改变比率 | 全部平均Label Size改变比值 | Query cost提高比 |
| 1 | 0.00 | 0.90 | 0.10 | -27.46% | 40.8510% | 41.3146% |
| 2 | 0.00 | 0.80 | 0.20 | -29.50% | 29.5055% | 43.2262% |
| 3 | 0.00 | 0.70 | 0.30 | -32.12% | 21.0830% | 45.3039% |
| 4 | 0.00 | 0.60 | 0.40 | -33.86% | 10.9260% | 46.6075% |
| 5 | 0.00 | 0.50 | 0.50 | -36.76% | 1.8419% | 48.7421% |
| 6 | 0.00 | 0.40 | 0.60 | -39.13% | -7.8930% | 50.1700% |
| 7 | 0.00 | 0.30 | 0.70 | -41.79% | -18.4739% | 51.6348% |
| 8 | 0.00 | 0.20 | 0.80 | -44.90% | -28.8447% | 53.0997% |
| 9 | 0.00 | 0.10 | 0.90 | -48.54% | -37.7091% | 54.7931% |
| 10 | 0.00 | 0.08 | 0.92 | -48.72% | -39.1547% | 54.3346% |
| 11 | 0.00 | 0.06 | 0.94 | -49.65% | -41.4417% | 54.3721% |
| 12 | 0.00 | 0.04 | 0.96 | -50.31% | -43.1849% | 54.1905% |

LHP VS BHP

Table 3: LHP VS BHP analysis result

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Compare to Betweenness Ordering (TimeInterval:30 min, hfRate:20%) | | | | | | |
| 实验次数 | Degree Coefficient | Freq Coefficient | Betweenness Coefficient | 高频点平均Label Size改变比率 | 全部平均Label Size改变比率 | Query cost提高比 |
| 1 | 0.00 | 0.90 | 0.10 | 34.70% | 158.8896% | -9.8241% |
| 2 | 0.00 | 0.80 | 0.20 | 30.91% | 138.0361% | -6.2467% |
| 3 | 0.00 | 0.70 | 0.30 | 26.05% | 122.5554% | -2.3584% |
| 4 | 0.00 | 0.60 | 0.40 | 22.83% | 103.8863% | 0.0810% |
| 5 | 0.00 | 0.50 | 0.50 | 17.44% | 87.1895% | 4.0758% |
| 6 | 0.00 | 0.40 | 0.60 | 13.03% | 69.2962% | 6.7479% |
| 7 | 0.00 | 0.30 | 0.70 | 8.08% | 49.8481% | 9.4891% |
| 8 | 0.00 | 0.20 | 0.80 | 2.31% | 30.7862% | 12.2307% |
| 9 | 0.00 | 0.10 | 0.90 | -4.44% | 14.4932% | 15.3996% |
| 10 | 0.00 | 0.08 | 0.92 | -4.78% | 11.8361% | 14.5416% |
| 11 | 0.00 | 0.06 | 0.94 | -6.51% | 7.6324% | 14.6117% |
| 12 | 0.00 | 0.04 | 0.96 | -7.73% | 4.4284% | 14.2718% |

实验结果表明，在粗略找到的最优参数下，相较于最优的BHP算法，对于任意选取的时间段，可以在全部点的平均Label Size只增大2%左右的情况下，查询性能提升14%左右。经过初步验证，如果是高频点分布更集中，即高频点查询次数所占比率更大，提升效果更好。