哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

数据库系统

论文概述

(2019年度春季学期)

姓　　名　朱明彦
学　　号　1160300314
学　　院　计算机学院
教　　师　高宏

计算机科学与技术学院

目录

论文概述

第1章 解决的问题

选择的论文 [1]为2018年PVLDB上面的一篇文章，**主要解决的问题是Road Networks上面动态的kNN查询，论文编号35**。

针对这个问题，作者主要有如下**三个方面**的工作：

1. 对于Road Networks上需要进行kNN查询的系统建立新的数学模型，并找到其中影响系统整体吞吐量的关键因素。
2. 建立了一个以Shortcut Graph为基础的索引，SCOB。
3. 设计了可以根据Update/Query的所用时间动态调整SCOB以最大化系统吞吐量的算法，TOAIN。

第2章 采用的思想

作者为了解决Road Networks上面动态的kNN查询，采用的思想是**先明确系统的目标（最大化系统的吞吐量），建立数学模型衡量寻找影响体统吞吐量的关键因素，对于关键的因素建立索引并利用设计的算法尽量提高关键因素影响下的部分，最终实现最大化吞吐量的目的。**

Notation	Description
(t_q, V_q)	expected/variance-of query time
(t_u, V_u)	expected/variance-of update time
λ_q, λ_u	query/update arrive rate
R_q	average query response time
R_q^*	average query response time bound, a QoS measure
λ_q^*	largest average throughput subject to an R_q^* constraint
T	update periodicity (under the QF model)
$m = \|\mathcal{M}\|$	number of objects
$G = (V, E)$	graph representing a road network
$d_G(u, v)$	shortest distance between u and v in G
$u \rightsquigarrow v$	a shortest path from u to v
$v \curvearrowright u$	a shortcut from u to v
$u_\downarrow$	the set of downhill objects of u

图 2.1: Notations

2.1 数学模型方面

作者主要考虑了两种不同的情况，分别对应于现实中两种不同的应用，即**如滴滴类似的打车软件系统和捕捉精灵球的Pokemon**。作者在考虑数学模型时，从**Query**和**Update**分别考虑

其影响，然后针对这两种不同的应用区分它们在Query和Update上面的差别，从而建立起来不同的数学模型。

以下两种应用都假设所有的查询到来是随机的，论文中为了方便假设Query符合泊松过程(Poisson Process)。

2.1.1 打车软件（BUA+QF Model）

BUA(Batch Update Arrival)模型是针对Update操作来说，其假设所有的元素都会进行Update操作，且都是在时间片的开始时刻进行，如果在某一个时间片里没有完成相应的Update操作，则直接舍弃未做操作，进行下一轮操作即可。**QF(Query First)是针对系统中队列模型来说的**，其假设Query在队列中的优先级高于Update。

基于上面的假设，在打车软件这样的系统中，对于每个Query是来自用户的，查询在其附近最近的K个车辆；每个Update是来自汽车的，其在指定间隔内向系统返回其最新的位置。这样，高优先级的Query可以尽量减少用户请求的查询的延迟，并且丢失少量Update使得某些车辆的信息具有很短距离的差距，所以这样的假设对于该应用是合理的。

利用在 [2]中的结论，可以得到式(1)，其中R_q为平均查询响应时间，t_q, V_q分别是查询时间的期望和方差，λ_q为Query到达的速率，即泊松过程中的参数。

$$R_q = \frac{\lambda_q \left(t_q^2 + V_q\right)}{2\left(1 - \lambda_{q^t q}\right)} + t_q \tag{1}$$

通过式(1)可以看到，**整体的查询响应时间随λ_q的增大而增大，这也与我们的直觉相符合，说明此模型具有一定的道理**。进而通过两个约束，**查询的响应时间不能超过用户能够忍受的最大值**和**在一个时间间隔内需要最少的用于Update的时间限制**，得到最终的λ_q的上界，如式(2)所示。

$$\lambda_q^* \le \min\left\{\frac{2\left(R_q^* - t_q\right)}{V_q + 2R_q^* t_q - t_q^2},\quad \left(T - mt_u\right)/\left(T \cdot t_q\right)\right\} \tag{2}$$

为了可解释性，将式(2)转化为式(3)。

$$\lambda_q^* \le \begin{cases} 1/t_q, & \text{if } \alpha\beta < 1/2 \quad (\text{ QoS-bound mode }) \\ (1-\beta)/t_q, & \text{if } \alpha\beta \ge 1/2 \quad (\text{Update-bound mode}) \end{cases} \tag{3}$$

其中$\alpha = R_q^*/t_q;\quad \beta = mt_u/T;\quad \gamma = V_q/t_q^2$；$\alpha$用来衡量平均查询相应时间的上界与平均查询时间的比值，β用来衡量一个时间片内处理Update所用时间的占比，γ是离散系数（coefficient of variation）的平方，相对于标准差其为无量纲量可以用在不同的索引算法中进行比较，一般$\gamma \in [0.1, 0.9]$。

2.1.2 Pokemon游戏（RUA+FCFS模型）

对于pokemon类似的以所在位置为中心，查询周围存在的Pokemon，并进行捕捉的这类游戏，其Update不是按照某一指定的时间间隔返回的，而是随机的，所以其Update操作不能在

使用BUA模型，而是转为RUA(Random Update Arrival)模型，同样的为了方便研究，**作者假设RUA的Update符合另一个泊松过程（与Query到来的泊松过程不同）**。

对于Query的优先级，Pokemon游戏很注重每个Pokemon的位置更新，所以该模型的队列模型使用了FCFS（First-come-first-served），即Query和Update具有相同的优先级，根据到来的顺序来进行操作。

其分析过程与2.1.1节类似，最终得到式(4)，衡量在该模型下的Query到来频率的上界。

$$\lambda_q^* \leq \min\left\{\frac{2\left(R_q^*-t_q\right)\left(1-\lambda_u t_u\right)-\lambda_u\left(V_u+t_u^2\right)}{V_q+2R_q^* t_q-t_q^2}, \frac{1-\lambda_{\mathrm{u}} t_u}{t_q}\right\} \tag{4}$$

2.2 索引SCOB方面

2.2.1 假设条件

在索引和算法方面，作者做出了两个假设，分别是：

1. 抽象后的图G是无向图
2. 所有节点的Rank是不一样的

2.2.2 符号定义

SHORTCUT SET $SC_<$ 给定一个图$G=(V,E)$和一个定义在$V\to\mathbb{N}$上的Rank Function，则一条捷径(shortcut)$u\curvearrowright v\in SC_<$当且仅当1.$u,v\in V$，2.$r(u)<r(v)$，3.对于从$u$到$v$上最短路上的中间任意节点$z$均有$r(z)<r(u)$。

从上面的定义我们可以直接得到一个结论，如式(5)，其中$SC_<$是从G上导出的捷径图(shortcut graph)，因为不一定所有的G上的最短路都符合$r(z)<r(u)$即上述定义的第3条规则。

$$d_G(u,v)\leq d_{SC_<}(u,v)\quad \forall u,v\in V \tag{5}$$

SUMMIT NODE 给定一个最短路径(shortest path)$s\rightsquigarrow t$，则对于该最短路的Summit Node是$s\rightsquigarrow t$上具有最高权重的节点。

根据$SC_<$和summit node的定义以及第一个假设，我们可以直接得到式(6)，其中x是最短路径$s\rightsquigarrow t$上的summit node。

$$d_G(s,t)=d_{SC_<}(s,x)+d_{SC_<}(t,x) \tag{6}$$

DOWNHILL OBJECTS 论文中作者为了形象表示不同节点的rank不同，具有越高rank的节点在图中的位置越高，如图2.2所示。基于这种形象的表示，如果在$SC_<$中有一条从v指向u的shortcut，则称v是u的downhill object。

kDNNs 给定一个节点$u\in G$，离它最近的k个downhill objects被记作$kDNN(u)$。

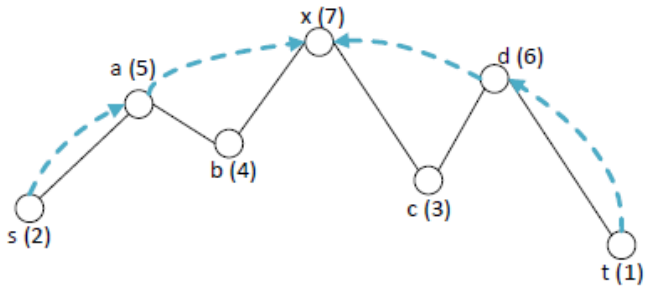

图 2.2: Downhill Objects Example

2.2.3 基本思想

根据式(5)和G是无向图的假设，可以有结论给定一个节点s和一个位于节点t的物体o，如果o是s的kNN之一，则有$o \in kDNN(x)$，x是最短路$s \rightsquigarrow t$上的$summit\ node$（其证明使用了证明其逆否命题的方法，主要利用三角不等式、式(5)进行放缩）。**通过以上结论，可以利用kDNN和一些特定的summit node可以得到一些源点的KNN结果。**

SCOB索引本质上也是利用了Dijkstra在图上进行搜索，而传统的方式对于一个有M个节点的图，需要进行M次s-climb(从源点到summit node的搜索)以及M次t-climb(从目标点到summit node的搜索)；而使用了SCOB索引**在每个summit node上记录在Shortcut Graph中的k个最近的downhill objects的id和其距离，以减少s-climb(t-climb)的次数，来提高效率，最终实现了仅需要从Query发出的节点进行一次s-climb就可以找到对应这次Query的kNN结果。**

具体如何维护SCOB索引，见3.1节。

2.2.4 假设条件放松

对2.2.1节提到的两个假设，论文中分别进行了放松。对于第一个假设，其仅仅提到了一些做法，但在实际中可能并不适用，具体解释见6节。

对于第二个假设，作者修改了2.2.2节中shortcut的定义，其中第2点改为$r(u) \leq r(v)$，进而将如此导出的图记为$SC_{\leq}$。进而称在$SC_{<}$上的climb为straight climb，而在$SC_{\leq}$上的climb为gentle climb。

2.3 算法TOAIN方面

2.3.1 符号定义

COVER NODES, COVER DIMENSION 将Road Network放在二维空间上的地图，然后利用$K \times K$个方格将其分成K^2个cell，对于给定的cell C，其外面有3 $times$3和5×5的邻居，则对于处在C的节点u，如果$u \rightsquigarrow v$必须穿过3×3的邻居，其中在最短路径上且跨过边界的边(x, y)，那么称x为C的一个Cover Node，所有这样的Cover Node的数量称为Cover Dimension。**例子见5.1节说明。**

2.3.2 基本思想

TOAIN能够通过调整Rank function来调整SCOB索引的参数，寻找Query和Update之间的折中，进而达到最大化吞吐量的目的。所以**TOAIN的主要思想就是如何确定更合理的Rank Function，使得越重要的顶点具有越高的Rank**。具体的算法描述见3.2节所示。

第3章 基本算法描述

文中以伪代码形式讲述的算法共有4个，下面将分别进行描述。

3.1 SCOB Index

3.1.1 Query

Algorithm 1: Query ($SC_<$, q, k)

```
1  R ← ∅; d_0 ← longest distance of an object in R from q, initially ∞;
   /* Conduct Dijkstra search from q on SC_<, with
      the following operations.                    */
2  for each node p being visited in the Dijkstra search do
3      if ||R|| ≥ k and d_SC<(q, p) > d_0 then
4          break ;
5      for object o ∈ kDNN(p) do
6          add [o, d_SC<(q, p) + d_SC<(o, p)] into R;
7          if ||R|| > k then
8              remove from R the object with longest distance from q.
9          update d_0;
10 return R;
```

图 3.1: Query Algorithm

在SCOB索引上进行kNN查询，主要是利用从查询所在的节点开始Dijkstra算法，对于每一个遇到的节点利用其kDNN更新最终的结果集合$\mathcal{R}$，直到遍历完所有节点或者$\mathcal{R}$中已有k个节点且当前遇到的节点在$SC_<$中的距离已经超过$\mathcal{R}$中的最大距离停止。算法如图3.1所示。

由于其利用的是Dijkstra从原点进行搜索，所以其复杂度为$O(k|V|log|V|)$。

3.1.2 Insert

对于插入一个物体o，则进行从o所在节点t的Dijkstra算法，更新所有中间经过的节点的kDNN集合。算法如图3.2所示。

复杂度同样是Dijkstra的复杂度，即$O(|V|log|V|)$。

3.1.3 Delete

删除一个物体o，则需要首先判断该物体是否在其所在节点的kDNN集合中，如果不在算法直接结束；否则，需要从该顶点进行Dijkstra算法，对所有经过且kDNN包含物体o的节点，

Algorithm 2: Insert($SC_<$, o, k)

```
/* Suppose o is located at node t; Conduct
   Dijkstra search from t on SC_<, with the
   following operations.                          */
1 for each node p being visited in the Dijkstra search do
2     add [o, d_{SC_<}(t, p)] into kDNN(p);
3     if ||kDNN(p)|| > k then
4         delete from kDNN(p) the object with longest distance from
          p;
```

图 3.2: Insert Algorithm

Algorithm 3: Delete($SC_<$, o, k)

```
/* Suppose o is located at node t;                 */
1  if o ∈ kDNN(t) then
2      F ← ∅;
       /* Conduct Dijkstra search from t; lines 3~6
          show the operations done in the search */
3      for each node u being visited in the Dijkstra search do
4          if o ∈ kDNN(u) then
5              remove o from kDNN(u);
6              F ← F ∪ {u};
7      F* ←sort nodes of F in increasing ranks;
8      while F* is not empty do
9          p ← lowest ranked node in F*;
10         kDNN(p) ← at most k objects located at p;
11         for (v ⇝ p) ∈ SC_< do
12             for object o* ∈ kDNN(v) do
13                 add [o*, d_{SC_<}(o*, v) + d_{SC_<}(v, p)] to
                   kDNN(p);
14                 if ||kDNN(p)|| > k then
15                     delete from kDNN(p) the object with longest
                       distance from p;
16         remove p from F*;
```

图 3.3: Delete Algorithm

将o从其对应的kDNN集合中删除，并且对于所有上述节点补充新的物体进入kDNN集合。算法如图3.3所示。

复杂度为$O(|V|log|V| + |V||E|)$。

对于系统中涉及到的Update操作，没有直接给出相应的算法，而是改为用一次Insert和一次Delete操作进行代替。

3.2 TOAIN

3.2.1 Compute Rank

根据2.3.1节的定义，可以有结论**较大Cell对应的Cover Nodes具有较高的重要性，且较高重要性的Cover Nodes应具有较高的Rank**。

Algorithm 4: `ComputeRank` (G, h)

initialize $r(u)$ to 0 for every node $u \in G$;
$S \leftarrow V$ /* V is the node set of G. */
for *grid level* i *from* 1 *to* h **do**
  impose $K_i \times K_i$ grid on G;
  /* Refer to text for the value of K_i */
  $\Upsilon_i \leftarrow \emptyset$;
  for *node* $v \in S$ **do**
    conduct *Dijkstra* search from v within the 5×5 sub-grid, whose central cell contains v;
    $\Upsilon_{i,v} \leftarrow$ cover nodes found during the *Dijkstra* search;
    $\Upsilon_i \leftarrow \Upsilon_i \cup \Upsilon_{i,v}$;
  update $r(z)$ to i for every node $z \in \Upsilon_i$;
  $S \leftarrow \Upsilon_i$;

图 3.4: Compute Rank Algorithm

所以如图3.4所示的算法，先初始化所有的顶点rank为0，然后从1到h开始循环，较大的K逐步利用Dijkstra搜索进行更新遇到节点的rank值，同时不断减小K。

由于每次K对半分，所以Dijkstra经过的节点数目不超过$O(\zeta^*)$（Cover Dimension的上限），所以整体的复杂度为$O(h\zeta^*|V|log|V|)$

3.2.2 climb

对于TOAIN下面的climb，每次所经过的节点不会超过$h\zeta^*$个，根据上一节的分析显然成立，所以进行一次climb的时间复杂度为$O(h\zeta^*)$。

第4章 结论

根据2.2.3节，文中提出的算法针对每一次Query仅需一次s-climb，而根据3.2.2可知每次climb的复杂度不超过$O(h\zeta^*)$，每一次Query时间复杂度为$O(h\zeta^*)$。调整h可以调整Query和Update之间的时间，从而达到最大化吞吐量的目的。

根据 [3]，ζ^*不会很大，所以Computer Rank实际的复杂度接近$O(|V|log|V|)$。

最后根据论文中的实验结果，也侧面证实了其在相关算法中，无论对于实际例子还是生成数据，无论是BUA+QF模型还是RUA+FCFS模型，SCOB和TOAIN都是用时最少的。

第5章 举例说明

5.1 COVER NODES, COVER DIMENSION

如图5.1所示，对于给定的Cell C，位于其中的节点u，其3×3, 5×5的邻居均在图中标出。其中对于u与任何节点的最短路而言，其均需要通过i, d, o中的一个才可以到达，而$(u, d), (u, i), (u, o)$均跨过了C的边界，所以i, d, o均为Cover Nodes，且C的Cover Dimension为3。

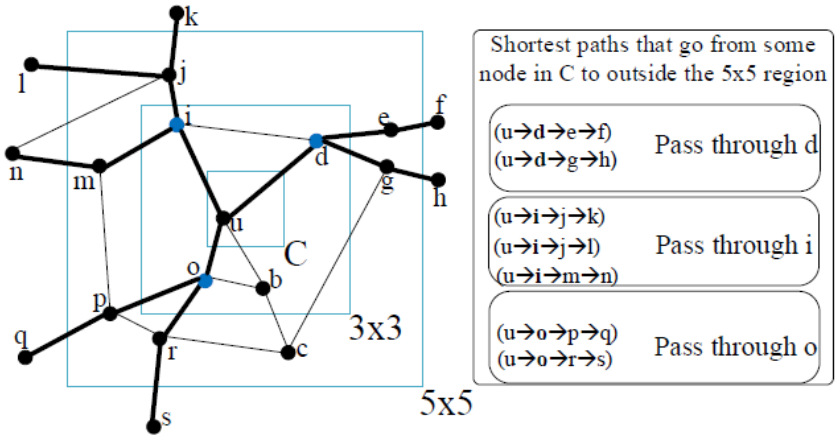

图 5.1: Cover Nodes example

5.2 COMPUTER RANK

如图5.2所示，初始化$K = 10$，$h = 2$，图中所有的蓝色标记表示一个节点，红色数字表示该节点的rank值，橙色线段表示边，黑色方框表示Grid。

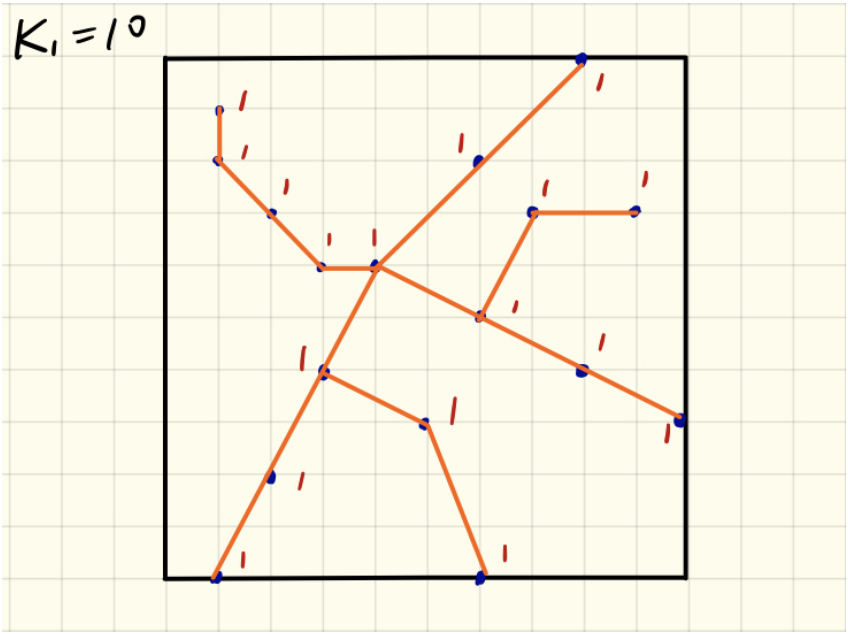

图 5.2: Loop 1

可以看到，在初始化的第一次循化会将所有节点的Rank都更新为1，此时的S仍然包含所有的节点。

进入第二轮循环，此时K被更新为5，所有的Dijkstra算法都不会超过其对应的最小的5 × 5的Grid。所有cover nodes的权值被更新为2，如图5.3所示。

由于继续更新，K会小于5，而此时与循环中需要对其所在的5 × 5的Grid进行Dijkstra算法，所以K不够大导致算法停止，最终结果如图5.3所示。**可以看到，最终Rank值较大的节点，也**

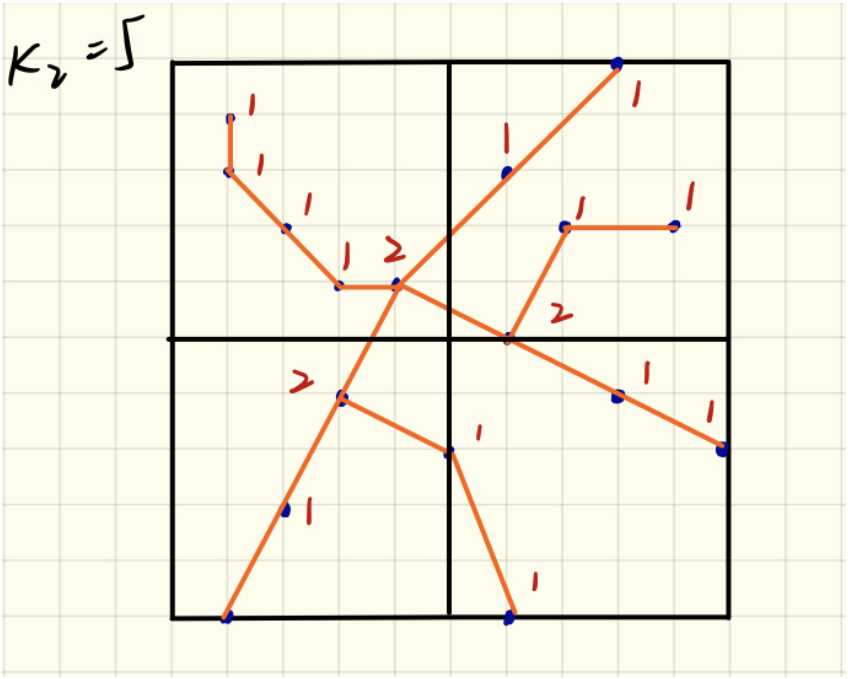

图 5.3: Loop 2

是具有较高连通度的节点，这与使用*betweenness centrality*进行rank赋值，具有相似性，说明了此算法的合理性。

第6章 算法的不足

在论文中，作者提出放松2.2.1节中第一个假设的方式，是针对将有向图的所有边均取反，这种做法可能导致整体计算Shortcut Graph时间复杂度变化，并且在现实中确实**存在单行线**，如果简单对其取反可能缺少实际的物理意义，继而可能引入并不存在的边导致实际距离估计偏差较大的问题，如存在最短路径$s \rightsquigarrow t$，且x为其上的summit node，而此时$d_{SC_<}(t,x) \neq d_{SC_<}(x,t)$，进而导致式(6)不成立。

在相关的实验中，作者也没有进行关于Road Network中真正存在单向边，无法将其转化为无向图处理的情况提供相应的实验数据，也侧面说明了在这一部分TOAIN和SCOB结合起来对于存在大量单向边的情况处理有所困难。

第7章 改进方法

由于SCOB索引和TOAIN算法设计的非常精巧，如果简单的想做基于这个索引的改进，会破坏掉其上面的性质，导致时间复杂度的变化较大，所以很抱歉我没有想到较好的改进方法。

第8章 相关工作(小综述)

使用Dijkstra算法进行kNN查询，从查询q所在的节点开始进行Dijkstra算法，直到有k个最近的物体被搜索到即可。这样没有维护索引的需要，但对于

ROAD [4]同样是基于Dijkstra算法，但是主要是针对物体在network上面稀疏分布的情况。主要思想是将大图划分成若干个小图(文中称为Rnets)，然后在利用Dijkstra搜索时如果某个小图(Rnet)中没有物体，则关于这个小图的搜索就被跳过。 [4]中主要使用的两类索引，Route Overlay和Association Directory，其中前者维护物理上的network结构包括shortcut，后者用来维护物体到节点、边以及Rnets的映射。其基本结构如图8.1所示。

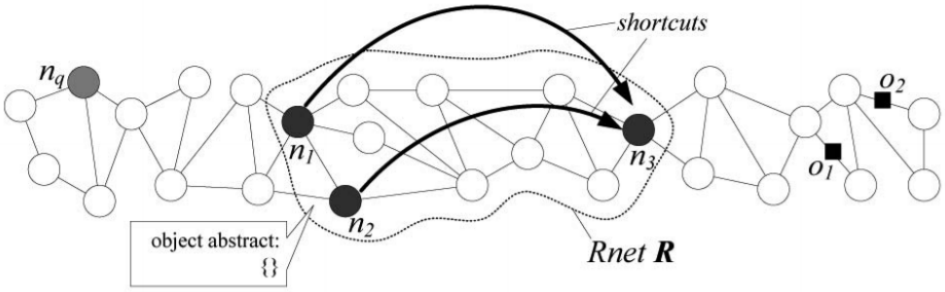

图 8.1: ROAD framework

G-Tree [5]也是建立了子图的层次结构，但是每个子图都与一个Occurrence-List关联，利用OL按照ID降序记录子图的层次结构。在G树上预计算并记录了基础节点之间的最短距离，然后通过这些基础点之间的距离和图的层次结构再计算其余节点之间的最短距离。也因为其记录了更多的信息，所以在G树上查询很快，但更新较慢。G树的基本结构如图8.2所示。在G-Tree上进行KNN查询的时间复杂度为$\mathcal{O}\left(\tau\log\tau+\log_2^2 f\cdot\log_f\frac{|\mathcal{V}|}{\tau}\cdot|\mathcal{V}|\right)$，其中$\tau$为每个叶节点最多含有的顶点数目。

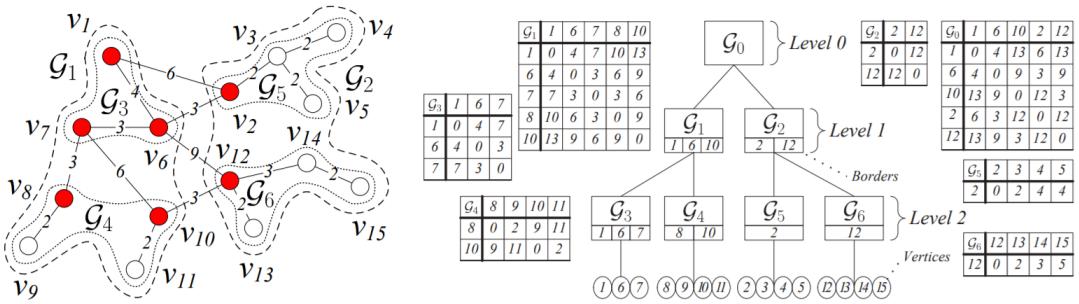

图 8.2: G-Tree Example

V树 [6]采用与G树类似的层次结构，如图8.3所示。V树标识位于子图边界的边界节点。通过维护这些边界节点的最近对象列表，设计了有效的技术来回答kNN查询。

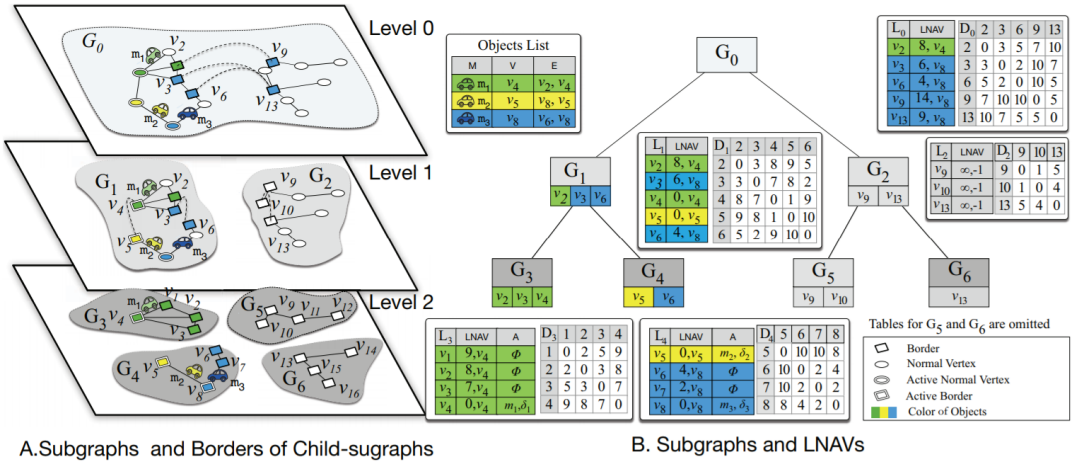

图 8.3: V-Tree Example

参考文献

[1] Luo, S., Kao, B., Li, G., Hu, J., Cheng, R., & Zheng, Y. (2018). TOAIN: a throughput optimizing adaptive index for answering dynamic k NN queries on road networks. Proceedings of the VLDB Endowment, 11(5), 594-606.

[2] J. W. Cohen. The single server queue, volume 8. Elsevier, 2012.

[3] S. Luo, R. Cheng, X. Xiao, B. Kao, S. Zhou, and J. Hu. Fast matching of detour routes and service areas. Technical Report TR-2016-03, The University of Hong Kong, 2016.

[4] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: A new spatial object search framework for road networks. TKDE, 24(3):547–560, 2012.

[5] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. TKDE, 27(8):2175–2189, 2015.

[6] B. Shen, Y. Zhao, G. Li, Q. Y. Zheng, Weimin, B. Yuan, and Y. Rao. V-tree: Efficient knn search on moving objects with road-network constraints. In ICDE, pages 871–882, 2016.