TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic KNN Queries on Road Networks.

**Student name:** 王丙昊

**Student number:** 1160300302

**Class number:** 1603108

**一、论文概述**

1. **本篇论文解决的问题**

本篇论文主要设计了TOAIN算法，用于在道路网路上实现**动态**的KNN查询。因为优化目标是使用这个算法达到比较高的**吞吐量**，所以作者实际上想要设计的是一个**系统**，在这个系统里面使用TOAIN算法，以达到较高的吞吐量。

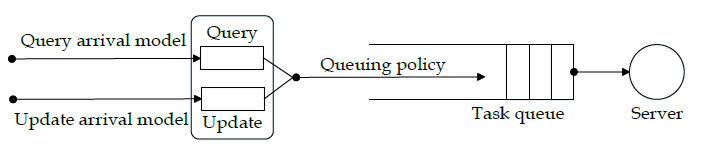
**动态**的KNN查询，动态指的是这个系统既接收查询请求，同时也接收更新请求。现有的许多工作仅仅考虑了在Road Network中如何减少KNN Query的执行时间，而没有将其放在一个既有查询也有更新的系统里面，这样就出现了一个**问题**：在一个既有查询也有更新的系统里面，仅仅减少KNN Query的执行时间，吞吐量可能不会得到提高，反而有可能会下降。

TOAIN算法解决了上面这个问题，可以在既有查询请求、又有更新请求的系统里面达到较高的吞吐量；同时可以对该算法的一些参数进行调整，以便适应不同特点的Road Network.

1. **证明现有方法的处理方式存在的问题**

在该部分，主要证明**“在一个既有Query请求又有Update请求的系统里面，仅减少Query执行时间，系统的吞吐量可能不会提高”**这一问题。

在证明该问题之间，首先介绍一下系统模型的结构（图1所示）：在这个系统中，有两种任务——Query和Update，到达该系统的任务首先存于队列中，然后基于某种队列存取策略，Server从队列中取出任务执行。



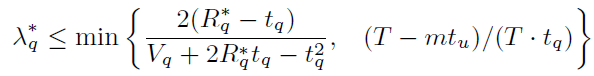
**图1：系统模型结构、**

在本篇论文中，讨论了两种比较有代表性的系统模型，这里分别针对这两种模型，来证明现有方法在这样的系统中处理时存在的问题。

* **BUA + QF模型**

这种模型的含义是，Query请求时随机到来的，Update请求是批量到来的，队列存取策略是查询优先(Query-first)策略。Query-first的含义为在队列中，query请求总是比update请求优先。嘀嘀打车可以看作是这种模型，用户发出打车请求是随机的，汽车司机的位置更新是批量到达的（比如5秒更新一次）。

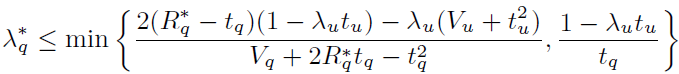
**BUA+QF模型的主要思想**是划分时间段，周期为T。在一个周期内，首先执行用户的query请求，还需要在每个时间段内为更新留下足够的时间。这种模型下吞吐量的上界为：



当更新负载比较小时，吞吐量取决于第一个式子，在这种情况下，减少query执行时间可以增大吞吐量；但是当更新负载大时，吞吐量取决于第二个式子，此时减小update执行时间（tu）比减小query执行时间（tq）效果要更好。

* **RUA + FCFS模型**

这种模型的含义是，query请求和update请求都是随机到来的，在队列中这两种请求地位相同，谁先到谁先执行（first-come-first-serve），将系统的处理能力考虑在内，可以得到这种模型下的吞吐量的上界：



当吞吐量由第一个式子所决定时，减少query执行时间可以增大吞吐量；由第二个式子所决定时，减小update的执行时间效果要更好。

1. **TOAIN算法采用什么思想解决的这个问题**

由上一部分的分析可知，在一个既有query请求又有update请求的系统当中，系统的吞吐量并不仅与query的执行时间有关，还与update的执行时间等一些因素有关。**TOAIN的主要思想**是将query time与update time进行**tradeoff**，来提高系统的吞吐量，并且可以通过调整TOAIN中的某些参数来适应不同特点的应用场景。

1. **TOAIN算法的描述**

TOAIN算法借助了CH算法（Contraction-Hierachy algorithm）的主要思想，在CH算法的基础上，建立了一个比较巧妙地索引结构——SCOB Index。下面分别介绍CH算法、SCOB索引以及TOAIN算法。

* **CH algorithm**

介绍CH算法前，首先引入几个要用到的概念：

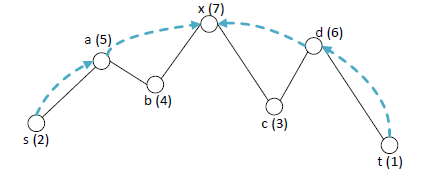
**Shortcut**：直连，直接相连两个顶点的边，边的权值为原图中两点之间最短路径的长度。

**Shortcut graph**：由原图中的顶点和shortcuts构成的图。

**Summit node：**峰节点，s到t的最短路径经过的ranking最高的节点，为该路径上的峰节点。

CH算法是一个在无向图中求最短距离的算法，主要思想是利用了**Shortcut**来表示两个点之间的最短距离，Shortcut如图2中蓝边所示。如果不加任何限制，任意两点之间都有一条shortcut（O(n2)），所以引入了**排名函数（ranking function）**来计算顶点的排名，通过对shortcut端点排名进行约束，来减少shortcut的数量。

最终剩下的shortcut形成了**shortcut graph**。如果要计算从s到t的最短距离，CH算法的做法是**在shortcut graph中分别从s和t做dijkstra搜索**，当两个搜索在x点（summit node）相遇时，便可以得到s到t的最短距离，也就是dG(s, t) = dSC<(s, x) + dSC<(t, x)。



**图2：shortcut等一些概念说明**

* **SCOB Index**

首先引入几个要用到的概念：

**s-climb**：CH算法求最短距离时，从s开始的dijkstra搜索为s-climb。

**t-climb**：CH算法求最短距离时，从t开始的dijkstra搜索为t-climb。

**Downhill object：**在shortcut graph中，如果从u到v有一条路径，那么位于顶点u上的物体为顶点v的downhill object。

**KDNNS：**给定一个顶点u，距离u最近的K个downhill object为该顶点的KDNNS，记为KDNNS(u)。

假设query请求的起点为q，当前图中有m个物体，由CH算法可知，可以通过执行m次s-climb和m次t-climb来获得距离q最近的k个物体。注意到s-climb都是从顶点q出发的，所以可以将m次s-climb缩减到一次；**为了减少t-climb的执行次数，提出了SCOB Index的概念**。

**SCOB Index的主要思想**是在每个顶点上保存其KDNNS，这样在执行s-climb的过程中，每当dijkstra搜索访问到一个顶点时，通过访问其KDNNS就可以获得距离该顶点最近的K个物体及距离，然后决定是否要把该物体加入 KNN query的top-k结果中，不需要再去执行t-climb。

通过这种方式，可以将m次t-climb减少到0次，也就是**仅通过一次s-climb**就可以获取KNN query的结果，这样做可以大大的减少query执行时间，代价是需要动态地维护SCOB Index。

* **TOAIN：Query**

经过上面SCOB Index的分析，可以将KNN Query的执行过程缩小到一次s-climb，仅通过一个从起点开始的Dijkstra过程，搜索过程中借助SCOB Index，就可以得到KNN Query的结果。**算法执行过程**（假设查询开始顶点为q）**：**

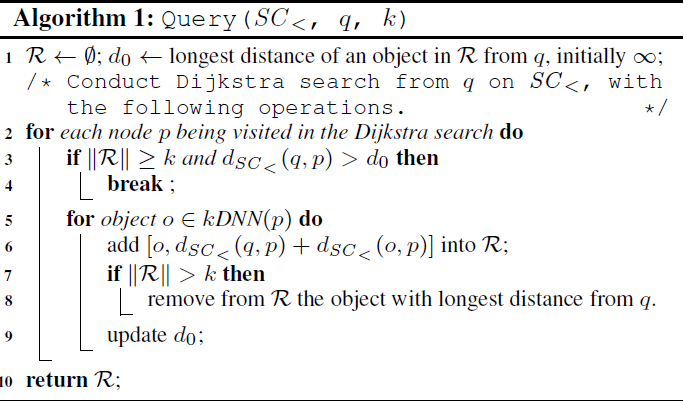
1. 初始化结果集合R为空；

2. 从q开始的dijkstra搜索每经过一个顶点p，执行3和4步。

3. 如果R大小为K，且q到p的距离大于q到R中任何一个物体的距离，什么都不做。

4. 否则，对于任一物体 o∈KDNNS(p)，首先将o加入R，如果R的大小超过K，则从R中删除距离最远的物体。

5. 最终返回结果集合R。



* **TOAIN：SCOB Index的维护（Update）**

SCOB Index的维护包括两部分，向图中**插入**一个物体和从图中**删除**一个物体，而更新可以通过执行一次删除再执行一次插入来实现。

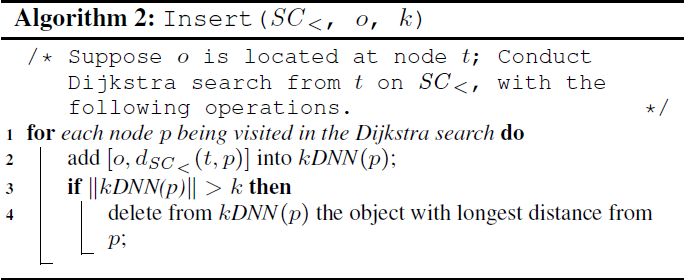
假设要将物体o插入到顶点t，或从顶点t删除物体o，算法的执行是通过**一次t-climb**实现的，也就是在shortcut graph中通过一次从t开始的dijkstra搜索来实现。

**插入操作算法执行过程**（假设将物体o插入到顶点t）：

1. 从顶点t开始做dijkstra搜索，每经过一个顶点p，执行下面的操作

2. 将[o, SC图中t到p的距离] 加入KDNNS(p)

3. 如果KDNNS(p)大小大于K，从中删除距离最远的物体

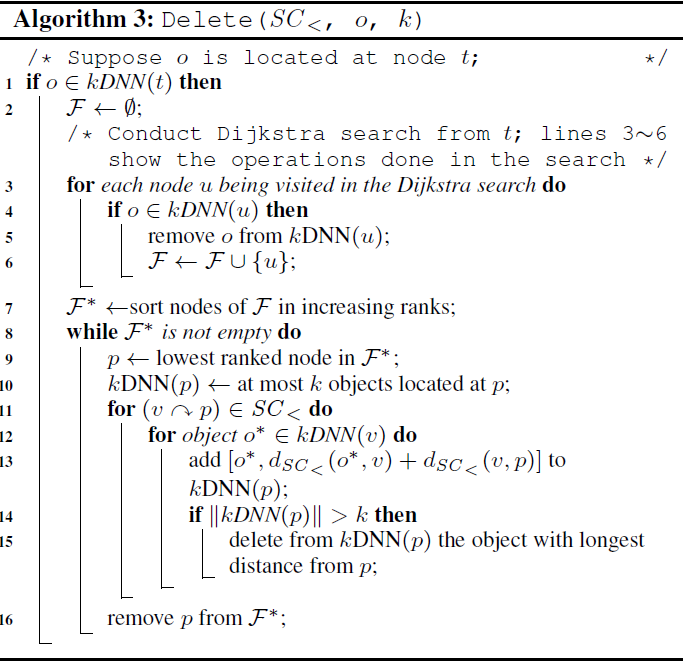


**删除操作算法执行过程**（假设要将顶点t上的物体o删除）：

1. 从顶点t开始做dijkstra搜索，每经过一个顶点p，执行下面的操作

2. 如果o∈KDNNS(p)，将o从KDNNS(p)中删除，然后从位于p或者与p直接相连的顶点上找最近的物体，来填充从KDNNS(p)删除的物体。

3. 如果o不在KDNNS(p)内，什么都不做。



* **TOAIN：Ranking Function**

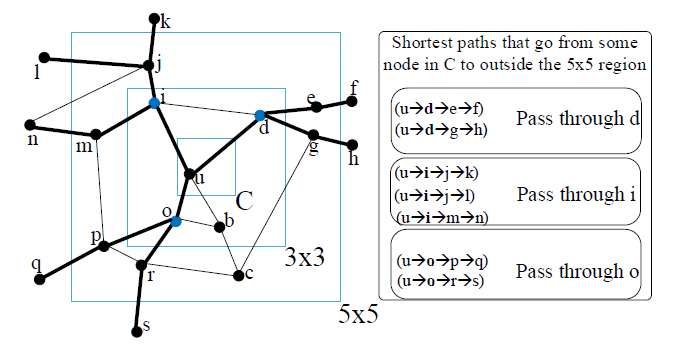
计算顶点排名的基本思想是: 越是重要的节点，它的ranking越高。衡量一个顶点的重要程度，一个比较直观的做法是计算其**中介中心度（betweenness centrality）**,但是这种做法实现起来时间复杂度较高。在本篇论文中，使用了**cover node和cover dimension**（覆盖点和覆盖度）来计算顶点的排名。

将原图用**K \* K的网格**进行划分，对于顶点u，其处于一个网格内，考虑该网格周围的3 \* 3和5 \* 5的网格，该网格的cover node与cover dimension定义如下：

Cover node：u与5 \* 5网格之外的点的最短路径在3 \* 3的网格内经过的点

Cover dimension：cover node的数量。

如图3所示，顶点u所在的网格的cover node为i、d、o，cover dimension为3。



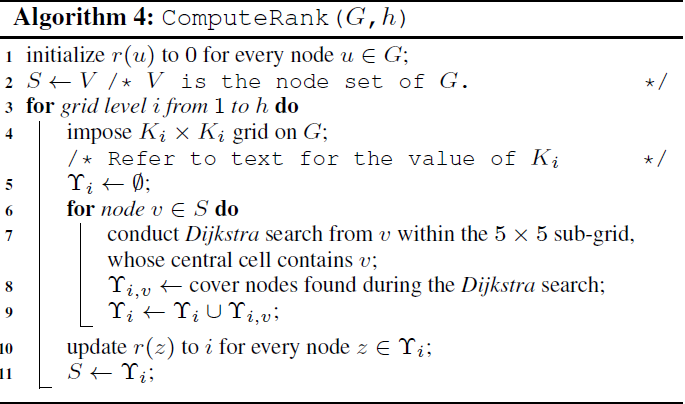
**图3：cover node/dimension举例**

Ranking Function是使用cover node的迭代算法，迭代轮数为h，算法执行过程如下：

1. 初始化所有点的排名为0，初始化顶点集S为V全体

2. 每次迭代过程中将G使用Ki \* Ki的网格进行划分，对于S中的顶点，计算其所在网格的覆盖点，将覆盖点的ranking置为当前迭代轮数。

3. 通过本轮所发现的覆盖点来更新顶点集S，S在下一轮迭代中使用。



* **TOAIN：如何调整参数适应不同应用场景**

CH算法要求图中各顶点的ranking不相同，也就是在shortcut graph中，所有边终点的ranking一定大于起点的ranking。在TOAIN算法中，对该条件进行了relaxing，允许一条边两个顶点的ranking相同。如果搜索过程中，所有边两顶点的ranking不同，称该搜索为**straight-climb**，否则称为**gentle-climb**。Gentle-climb的搜索速度比straight-climb的慢。

**调整参数的第一种方法：**经过上面算法的分析，TOAIN：Query可以看作是一次s-climb，而TOAIN：Update（也就是维护SCOB Index）可以看作是一次t-climb，所以可以调节s-climb和t-climb，使其一个为straight-climb，另一个为gentle-climb，通过这种方式可以对query time和update time进行tradeoff。

**调整参数的第二种方法**：修改ranking Function中的迭代轮数。随着迭代轮数的增加，顶点ranking的最大值也随着增加，通过调整迭代轮数，可以间接地调整s-climb或t-climb的时间，也就是在间接地调整query time和update time。

1. **算法分析的结论**

* **算法复杂度分析**

（1）TOAIN：Ranking Function，计算图中顶点的排名，时间复杂度为O(nlogn)，其中n为顶点的个数。

（2）为了减少shortcut graph中shortcut的个数，通过顶点的ranking添加约束条件，这个减少Shortcut数量的过程所需的时间复杂度为O(nlogn)，其中n为顶点个数。

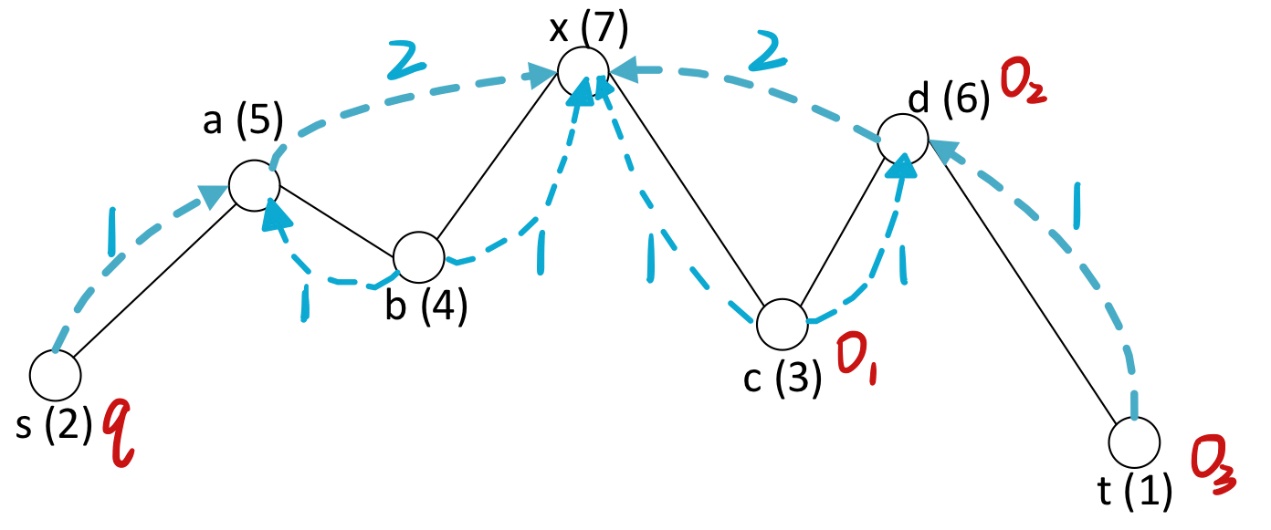
（3）减小后的shortcut数量为O（n），n为顶点的个数。

（4）对于TOAIN的query算法和update算法，都是通过一次dijkstra搜索实现的，时间复杂度就是dijkstra算法的复杂度。

* **算法效果分析**

在模拟数据集以及实际数据集上都进行了测试，测试结果表明，使用了SCOB Index的TOAIN算法是一个具有高度适应性的算法，它可以在不同的应用场景下使系统达到一个比较高的吞吐量；与现有的仅考虑减少query time的算法相比，TOAIN在许多方面效果都要优于现有算法。

1. **算法举例模拟**



**图4：完整的shortcut graph**

这里主要模拟一下TOAIN：Query以及TOAIN：Update（SCOB索引的维护）算法的执行过程，也就是假设图中所有顶点的ranking已经计算完成。

图2的完整的shortcut graph如图4所示，图中虚线的边表示shortcut，上面的数字表示shortcut的权值；图中有7个顶点，括号中为它们的ranking；当前图中发出query请求的物体位于s节点；有三个物体o1，o2，o3，分别位于c、d、t节点处。

**取K = 2，**则KDNNS集合元素最大个数为2**。**

**KDNNS(t) = { o3 }**

**KDNNS(d) = { o2, o3 }**

**KDNNS(c) = { o1 }**

**KDNNS(x) = { o1, o2 }**

* **TOAIN：Query算法的执行过程**

初始化结果集合R为空。

在shortcut graph图中，从s节点开始做dijkstra搜索，过程中会依次访问s、a、x节点，由于s和a的KDNNS集合为空，所以不对R进行任何操作；由于KDNNS(x) = { o1, o2 }，所以将o1和o2加入到R中，此时R的大小刚好为2，不进行任何删除操作，返回R，算法结束。

* **TOAIN：Update中Insert的执行过程**

**假设在顶点c处插入一个新物体o4。**

首先将o4加入顶点c的KDNNS中，由于加入后KDNNS(c)大小为2，不需要进行删除操作，所以更新后**KDNNS(c) = { o1, o4 }**

搜索到x时，首先将o4加入到KDNNS(x)中，此时KDNNS包含三个物体（大于Ｋ），分别是o1, o2, o4，然后需要从中删除举例ｘ最远的物体，o1和o4到x的距离为1，而o2到x的距离为2，所以更新后**KDNNS(x) = { o1, o4 }**

最终结果：

**KDNNS(t) = { o3 }**

**KDNNS(d) = { o2, o3 }**

**KDNNS(c) = { o1, o4 }**

**KDNNS(x) = { o1, o4 }**

* **TOAIN：Update中Delete的执行过程**

此时还是从图4这个初始状态进行操作，也就是不考虑上面的Insert。

**假设删除位于顶点t处的物体o3。**

首先访问顶点t，因为o3∈KDNNS(t)，所以将其从中删除，由于图中没有指向t的shortcut，所以KDNNS(t)得不到填充，**更新后KDNNS(t)为空**。

然后访问顶点d，因为o3∈KDNNS(d)，所以将其从中删除，删除后由于有一条从c到d的shortcut，所以将c上的物体o1加入KDNNS(d)，**更新后KDNNS(d) = { o1, o2 }.**

最后访问顶点x，由于o3不在KDNNS(x)中，所以不进行任何操作。

最终结果：

**KDNNS(t)为空**

**KDNNS(d) = { o1, o2 }**

**KDNNS(c) = { o1 }**

**KDNNS(x) = { o1, o2 }**

**二、已有工作综述**

1. **已有的研究工作**

道路网络上的KNN查询问题，已经被广泛的研究，包括:

1. 提出新的查询算法，如DisBrw算法

2. 提出新的Road network框架，如IER和ROAD

3. 提出新的灵活的索引结构，如G-tree和V-tree

1. **已有研究工作综述**

* **Dijkstra algorithm**

著名的单源最短路径算法，可以通过求解query点的单元最短路径，然后取前k个作为KNN query的结果。

由于该算法没有使用任何索引，所以更新代价会非常大。并且通常情况下，系统的更新负载是非常大的，所以这种方法不适合在一个既有query请求又有update请求的系统中使用。

* **IER**

IER是一种可以用于空间数据库的灵活的体系结构，其主要思想是将道路网络与实体数据体分离。它采用了基于磁盘的网络表示来保留连接和位置，而空间实体则通过各自的空间访问方法进行索引，以支持欧几里得查询和动态更新。由IER体系结构产生的算法通过连接集成和位置信息来扩展传统处理技术，以有效地修剪搜索空间，然后使用Dijkstra算法来求解网络距离。

在新的体系结构的基础上，还提出了几种新算法，涵盖了空间网络数据库中最常见的处理任务。实验结果表明，IER框架以及基于该框架的算法在范围搜索和KNN查询方面具有优越的性能。但毕竟该框架是为了众多查询提供一套完整的框架及算法，所以对每种方法的优化做的还不够，该框架在当时提出后具有很大的优化改进空间。

**Query processing in spatial network databases. In VLDB, pages 802-813, 2003**

* **DisBrw**

是一种使用网络距离以最佳方式在空间网络中找到K个最近邻居的算法。考虑到空间网络通常是静态的，而只有上面的对象是可能发生变化的，该算法基于预先计算的网络中所有可能顶点之间的最短路径，然后将其进行编码，编码充分利用了从顶点u到剩余所有顶点的最短路径可以分解为子集这一特点。因此，在最坏的情况下，算法的复杂度取决于所query的对象的数量以及从q到它们的最短路径上的链接的数量，而不取决于网络中顶点的数量。

该算法的优点在于每当网络上的对象发生变化时，不必重新计算最短路径，这种预计算的方式可以用于本次所研究的TOAIN算法的改进。

**Scalable network distance browsing in spatial databases. In SIGMOD, pages 43-54, 2008**

* **ROAD**

ROAD是一种评估位置相关查询（LDSQ）的通用框架，用于搜索Road Network上的控件对象。它利用了搜索空间修剪技术并提供动态对象映射机制，对大规模网络上的对象的各种类型的查询（如KNN）非常有效且灵活。ROAD以其两个组件命名，即路由覆盖和关联目录，旨在解决框架的网络遍历和对象访问问题。

在ROAD中，一个大型的Road Network被组织成一个互联的区域子网络（称为Rnets）,在其基础上增加了1）加速网络传输的shortcut；2）用于引导遍历的对象索引。

Road的这种设计提供了许多优点。首先，它提供了网络和对象之间的分离。实际上，地图服务提供商可以提供Rnet的shortcut，而内容提供商可以动态地将对象映射到顶点、边和Rnet，这样可以促进对象和网络灵活的更新；另外，可以在同一网络上支持不同的对象类型，因此可以有效地处理各种基于位置的query。

**Fast object search on road networks. In EDBT, pages 1018-1029, 2009**

* **G-tree**

G-tree是一种高度平衡且可扩展的索引结构，该索引不仅可以有效地处理非常大的数据集，还可以在基于位置的系统中有效地支持一些典型的道路网络查询算法，如单源最短路径算法、KNN query以及关键字KNN query。G-tree是受R-tree的启发提出的。G-tree递归地将道路网络划分成自网络，并在子网络之上构建树结构，其中每个G-tree节点对应于子网络。

在道路网络上，如果采用预计算的技术，将使用O(n2)的存储空间，查询时间为O(1)；如果不使用预计算的技术，空间代价为O(n)，查询时间为O(nlogn)。G-tree避免了这两种极端的情况，它只预计算了G-tree上基本对之间的最短距离，这样任何最短路径距离都可以由这些组合在一起。**使用G-tree索引的空间代价为O(nlogn)，时间开销为O(n)**，这样G-tree可以扩展到非常大的数据集，并且查询速度快。

**G-tree: An efficient and scalable index for spatial search on road networks. TKDE, 27(8):2175-2189, 2015**

* **V-tree**

V-tree是针对两个问题提出的一种索引结构：1）如何索引road network上的移动物体2）如何找到k个最近的移动物体。这种索引结构有两个显著的特征。首先，它是一个平衡的搜索树，可以有效地支持KNN搜索；其次，他可以支持移动对象的动态更新。

为了构建V-tree，需要迭代地将网络划分成自网络，并在子网络上构建树结构，然后将移动对象关联到V树中最近的顶点。更新对象的位置时，只需要更新从相应叶节点到根地路径上的树节点。通过对道路网络中大量的不相关节点进行剪枝，使用V-tree设计了一种新颖的KNN搜索算法。

**V-tree：Efficient knn search on moving objects with road-network constraints. In ICDE, pages 871-882, 2016**

1. **相关工作比较**

**（1）**与**IER**框架相似的有INE，它们都使用了**Dijkstra算法，但是使用的程度不同**，INE在遍历图时使用它来查找邻居，但是IER首先使用欧几里得距离查找邻居，然后使用Dijkstra算法找到他们的最短路径，从而找到真正的网络距离。

**（2）Disbrw**算法与**基于Dijkstra算法的方法（如IER）的关键区别**在于，在Disbrw算法中最短路径仅计算一次，而在基于Dijkstra算法的方法中，一些顶点之间的最短路径作为查询对象重复计算。在特定空间网络上进行许多查询时，使用Disbrw算法性能要更好；但是如果在给定的空间网络上仅进行很少的查询，尤其是期望的邻居非常接近查询对象的情况下，则基于IER的方法是优选的，因为不需要搜索整个空间网络。

Disbrw算法与基于Dijkstra算法的方法的另外一个区别在于，在Disbrw算法中空间网络与对象集是分离的，因此算法可以将空间网络与不同的对象集一起使用；相反，在基于Dijkstra算法的方法中，空间网络与对象集之间的区别并不明确。

**（3）Road框架与Disbrw算法相似的地方**在于，都实现了**将空间网络与对象集的分离**，并且可以在同一网络上**支持不同类型的对象集**，这样提高了系统的灵活性和可扩展性。这种做法已经成了道理网络框架设计的基本做法，在CH算法中，也是采用了将shortcut graph与对象集分离的做法，这种做法可以避免最短路径的重复计算，对于查询较多的网络可以极大的减少整体查询时间。

Road框架也是基于Dijkstra算法的，但与Dijkstra算法相比，Road以更新成本为代价来提供更快的查询时间，这可以看作是一种query time 与update time之间的tradeoff。

**（4）G-tree优于ROAD**，因为G-tree使用动态规划算法计算不同子图中两个对象的最短距离，而ROAD基于子图边界节点使用Dijkstra算法计算最短距离。

**（5）G-tree与V-tree**都是高度平衡且可扩展的树形索引，都是将道路网络划分为子网络并在子网络之上构建树结构。V-tree扩展了G-tree以支持**移动对象**，在这一方面，V-tree要优于G-tree：首先，在G-tree中，它只能知道哪些子树包含活动顶点，并且需要使用动态规划算法来计算不同子图中对象的距离，在V-tree中，使用一种新结构(LNAV)来保存每个边界最近的活动顶点，使用这种结构可以避免重复计算；其次，G-tree使用自顶向下的方式遍历树结构，会访问到不必要的节点，而V-tree仅基于LNAV结构访问活动节点，因此V-tree的搜索空间远小于G-tree。

**三、已有算法的不足与改进**

**（１）在shortcut graph中使用G-tree索引加快TOAIN算法的执行速度**

正如第一部分TOAIN算法综述中说的，要在shortcut graph中使用CH算法来求解最短距离，由于原始shortcut的数量太多（O(n2)数量级），要定义排名函数(ranking function)来对shortcut进行约束以减少其数量。在算法分析中提到了减少后的shortcut graph中shortcut的数量大幅减少（O(n)数量级）。

在TOAIN算法中，没有对shortcut graph使用任何**预计算**的方法，这样在执行TOAIN：Query以及TOAIN：Update时，每次都需要online地去寻找最短路径，这样的时间开销是O(nlogn)的，空间开销是O(n)的。

因为空间网络通常是静态的，所以其对应的short graph也是静态的，只有上面的对象是动态变化的，所以我认为这里可以使用一种**预计算的数据结构**，比如说提前建立关于shortcut graph中最短距离的索引结构，如灵活可扩展的**G-tree索引**。这样在TOAIN：Query和TOAIN：Update两个算法的执行过程中，不必每次使用时间复杂度较高的Dijkstra算法来进行搜索或更新。这种改进的主要思想是将**空间代价与时间代价进行tradeoff**，以空间代价（预计算并建立新的索引结构）来换取算法的执行效率，使算法性能得到提升。

在shortcut graph上使用G-tree索引后，空间代价为O(nlogn)，每次执行TOAIN：Query以及TOAIN：Update的时间开销为O(n)。**也就是时间代价从O(nlogn)变为了O(n)，空间代价从O(n)变为了O(nlogn)**，这样仍具有大规模数据集的可扩展性，并且同时缩短了query和update的执行时间，可以使吞吐量得到提升。