



Highway Hierarchies

一种最短路径搜索算法的构建与查询

一些基本的寻路算法

n个点，m条边

- 广度优先搜索 (BFS) : $O(n + m)$
- 经典的Dijkstra算法: $O(n^2)$
- 二叉堆的Dijkstra算法: $O((n + m)\log(n))$
- Fibonacci堆的Dijkstra算法: $O(m + n\log n)$
- A*算法: 复杂度与估价函数有关
- 各种双向优化的算法: 只能降低部分搜索的时间

对于上千万个节点的网络，使用上述算法肯定是无法满足最短路径查找的需求！

Route Planning in Road Networks

- Highway Hierarchies (公路层次算法)
- Highway-Node Routing
- Many-to-Many Shortest Paths
- Transit-Node Routing



对于几百万顶点的交通图，只需要进行几十分钟的预处理，就能将每次最短路径的查询时间降到1s一下！



公路层次算法 (Highway Hierarchies)

- 主要思想
- 名词定义
- 公路层次的构造
- 最短路径的查询

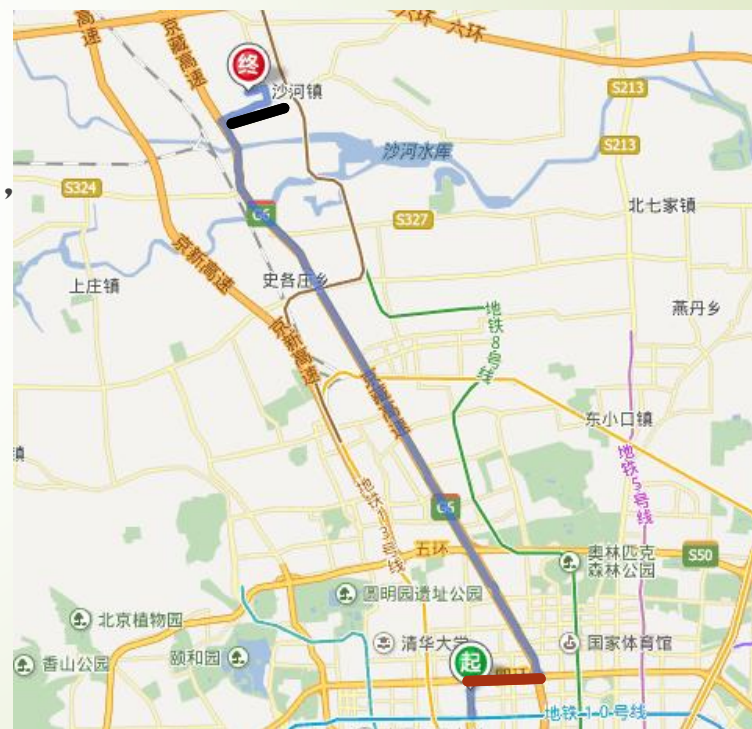
主要思想


我们在实际生活中寻路时，基本上遵循着这样一套简单的规则：

- 寻找一条可以通向目的地的高速公路，进入高速公路
- 在这条高速公路上行驶直至离目的点很近时，我们离开高速公路
- 寻找一条通向目的地的公路

比如右图：北航主校区→北航沙河校区：

- 学院路
- 京藏高速
- 百沙路





主要思想

一些商用的寻路算法的主要思想也是基于上述思想，即：

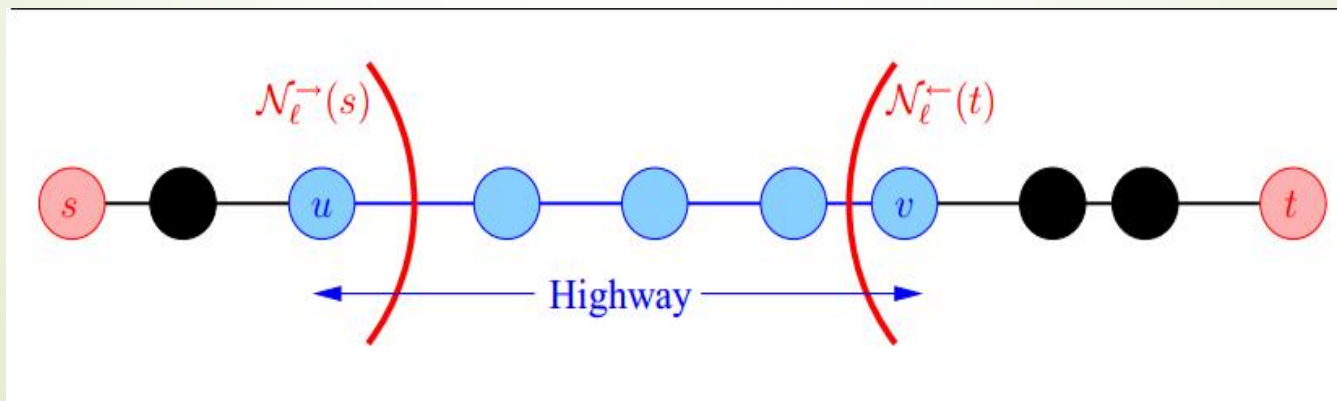
- 同时在源点和终点周围一定距离（比如20 km）中搜索所有路线
- 同时在源点和终点周围一定距离（比如100 km）中只搜索国道（notion roads）和高速公路（motorways）
- 再在离源点和终点周围更远的距离中只搜索高速公路

公路层次算法的主要思想和上述思想相差不大，也是将公路网分层再进行查询。

名词定义

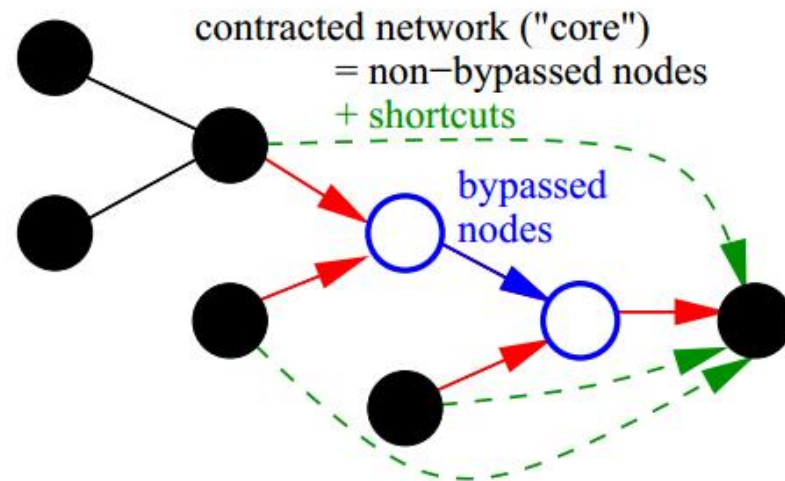
局部区域 (local area) : 对于公路网络中的每个顶点 u , 我们设定一个邻近半径 (neighborhood radius) H , 所有离 u 的最短距离不超过 H 的点称作 u 的邻居 (neighborhood)。通常来说 H 是一个可调参数, 对于不同的公路网络拥有不同的 H 值。

高速公路网 (highway network) : 对于网络中的一条边 $e=(u, v) \in E$, 如果 e 在图中某两个点 s, t 的最短路上, 且 u, v 不全是 s, t 的neighborhood。那么我们称 e 在highway network中。



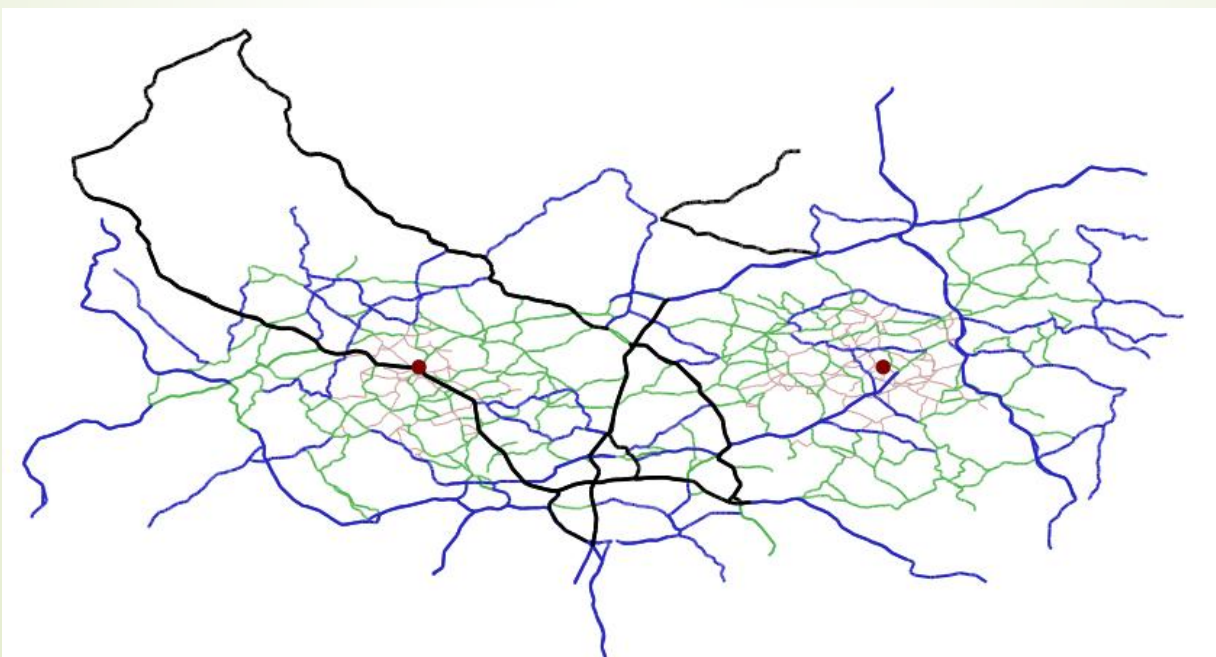
名词定义


收缩的高速公路 (**contracted highway network**)：在Highway中，包含了很多度很小的顶点，这些点 (*bypassed nodes*) 在寻路中没有任何作用，所以我们把它们删除，并添加上捷径 (*shortcuts*)，也就是将高速公路的多条边简化成一条边。我们把简化后的高速公路称作*contracted highway network*，也可以称作*core*。



名词定义

高速公路网络结构 (**highway hierarchy**) : core是可以递归计算的, 也就是说, 当我们计算出原公路网络的core后, 可以再此基础上再计算该网络的core.....我们把原本的公路网络称作第0层网络 (level 0), 从原网络中计算出的core称作第1层网络 (level 1) 等等。





公路层次的构造

- 算法总述
- 计算高速公路网 (Highway Network)
 - 构建局部最短路径有向无环图
 - 选择属于Highway的边
- 计算高速公路核心 (core)

算法总述

对于一个图 G 来说，它的highway hierarchy是 $G_1, G_2, G_3, \dots, G_L$ ，我们可以通过下述方法求得各 G_i ：

- 对于第0层的网络图，我们定义 $G_0 = (G'_0, G_0)$ ， G_0 即是原本的网络图， $G'_0 = G_0$
- 第一步，我们计算 $(G'_l \rightarrow G_{l+1}, 0 \leq l \leq L)$ ：根据给定的neighborhood radii，我们计算 G'_l 的highway network G_{l+1}
- 第二步，我们计算 $(G_l \rightarrow G'_l, 1 \leq l \leq L)$ ：先找出所有的bypassed nodes，再计算出第 l 层的core G'_l

更具体的总述

我们要明确上述的每一步我们需要得到了什么：

- 第一步，从 $G'_l = (V'_l, E'_l)$ ，我们先计算出所有在 G_{l+1} 中的边 E_{l+1} ，然后我们就可以得到 V_{l+1} ——在 V'_l 中构成 E_{l+1} 的最小子集。
- 第二步，从 $G_l = (V_l, E_l)$ ，我们先计算出所有的bypassed nodes B_l ，再计算出所有的捷径 S_l ，那么我们就得到 $V'_l = V_l \setminus B_l$ ， $E'_l = (E_l \cap (V'_l \times V'_l)) \cup S_l$ 。将 G_l 中所有属于core的点都去掉则得到了省略点的部分 (components of bypassed nodes)。
- 最后我们得到一个简单的图 $\zeta = (V, E \cup \bigcup_{i=1}^l S_i)$ ，在这幅图中，所有的点和边在不同的网络层次都有着它自身的信息。对于边 e 来说， $l(e) = \max(l | e \in E_l \cup S_l)$ 。

计算Highway Network

- **Dijkstra秩 (Dijkstra rank)** 我们规定Dijkstra算法中第 k 个出优先队列的点的Dijkstra秩为 k 。
- **邻近半径 (Neighborhood Radii)** 我们将公路网络图都看作无向图。对于一个给定的参数 H_l ，对于一点 u ，若 $u \in V'_l$ ，那么该点的邻近半径为 $r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = d_l^{\leftrightarrow}(u, v)$ ，其中 v 在 u 的SSSP（单源最短路径）中的Dijkstra rank是 H_l ，是若 $u \notin V'_l$ ，那么该点的邻近半径为 $r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = \infty$ 。

计算Highway Network

为了构建 G'_l 的highway network G_{l+1} ，对于图中的每个点 s_0 我们需要做两件事：

- 构建它的局部最短路径有向无环图 (a *Partial Shortest-Path DAG*) B
- 遍历 B 中每个叶子节点到源点 s_0 的边，判断它是否属于highway network。

我们可以通过以 s_0 为源点的单源最短路径 (SSSP) 来它的构建局部最短路径有向无环图。

在 B 中的属于Highway network的所有边即构成整幅图的Highway network。

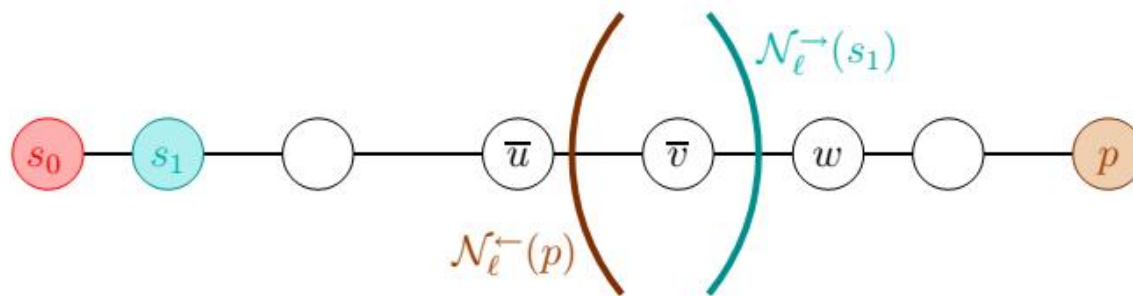
构建局部最短路径有向无环图

对于图 G'_l 中的每个点 s_0 ，我们求它的SSSP，我们规定每一个点都有一个状态 $active$ 或者 $passive$ ，一个点插入队列时的状态为 $active$ 当且仅当它的所有父节点（从源点 s_0 到该点有可能不止一条最短路径）都是 $active$ ，如果它满足

$\forall \text{shortest path } P' = \langle s_0, \dots, p \rangle:$

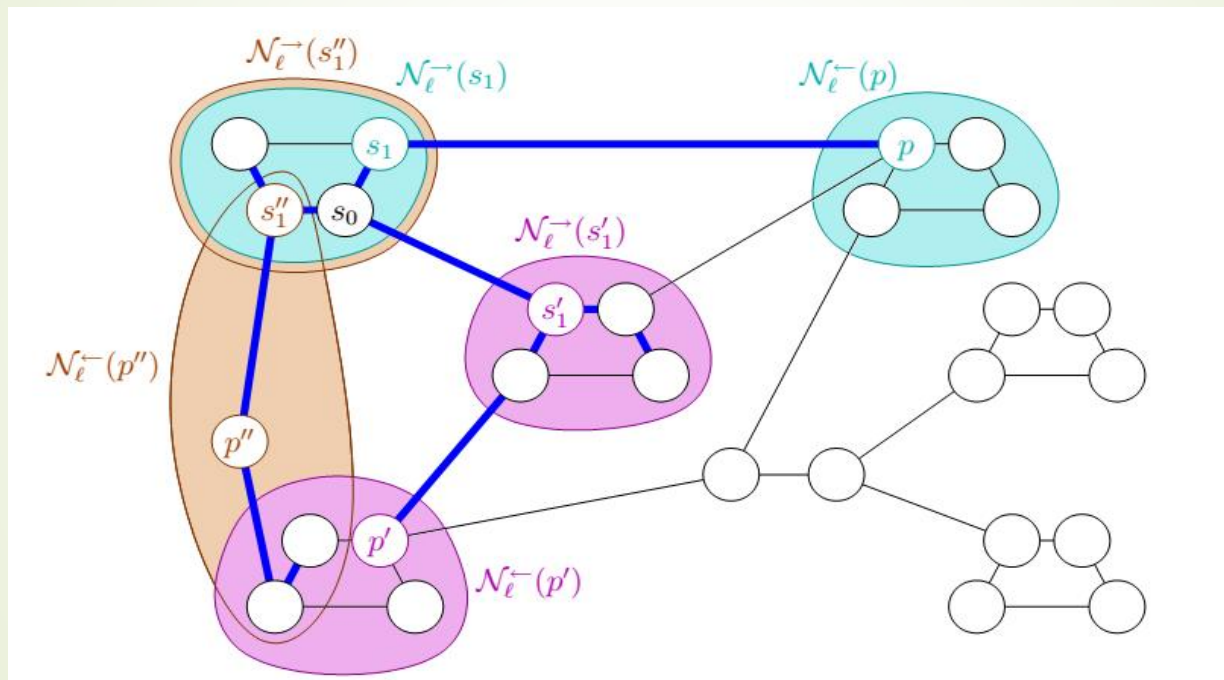
$$s_1 < p \wedge p \notin N_l^{\rightarrow}(s_1) \wedge s_0 \notin N_l^{\leftarrow}(p) \wedge |P' \cap N_l^{\rightarrow}(s_1) \cap N_l^{\leftarrow}(p)| \leq 1$$

，那么它的最终状态就是 $passive$ 。



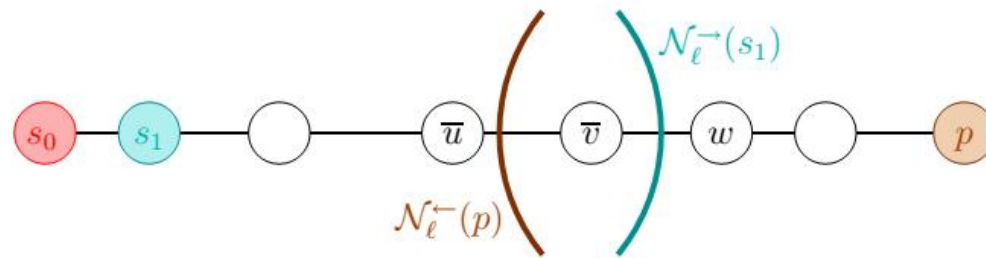
一般来说， s_1 选择 s_0 到 p 最短路上的第一个顶点。

构建局部最短路径有向无环图



当图中所有active的点都被遍历过后，我们就得到了某一点的局部最短路径有向无环图B

构建局部最短路径有向无环图



- 对于每个属于B的点 s_0 ，我们定义一个边界距离 (border distance) $b(x)$ 和一个参考距离 (reference distance) $a(x)$ 。
- 边界距离 $b(x)$ 就是从源点 s_0 到 s_1 邻居的边界的距离。这个距离随着SSSP的构建，沿着路径继承下来。这样就可以寻找到最短路径上第一个不在 $N_l^-(s_1)$ 内的点 w (如Figure4中所示)，它的父节点 v 是最后一个在 $N_l^-(s_1)$ 内的点。为了确认 v 是唯一一个属于 $N_l^-(s_1) \cap N_l^-(p)$ 的点，我们还要检查 v 的父节点 u 是否属于 $N_l^-(p)$ ，我们将 s_0 到 u 的相对距离 $d_l(s_0, u)$ 作为 $a(x)$ 传递给点 w 。通过 $d_l(s_0, u)$ ， $d_l(s_0, p)$ 和 $r_l^-(p)$ ，我们很容易就能知道 u 是否属于 $N_l^-(p)$ 。PS：如果从 s_0 到 x 有多条最短路径，则选择最长的 $a(x)$ 和 $b(x)$ 。

构建局部最短路径有向无环图

- 定义 $\wp(x)$ 为 x 在 B 中的所有父节点集合。
- 对于根节点 s_0 , $\wp(s_0) = s_0$, $b(s_0) = 0$, $a(s_0) = \infty$ 。
- 对于节点 $x \neq s_0$, 且其父节点的状态是 active:

$$b'(x) = \begin{cases} d_l(s_0, x) + r_l^{\rightarrow}(x), & s_0 \in \wp(x) \\ 0, & otherwise \end{cases}$$

$$b(x) = \max(\{b'(x)\} \cup \{b(y) | y \in \wp(x)\})$$

$$a'(x) = \max\{a(y) | y \in \wp(x)\}$$

$$a(x) = \begin{cases} \max\{d_l(s_0, u) | y \in \wp(x) \wedge u \in \wp(y)\} & a'(x) = \infty \wedge d_l(s_0, x) > b(x) \\ a'(x), & otherwise \end{cases}$$

计算 $d_l(s_0, \bar{u})$

将 s_0 到 s_1 邻居边界的距离向下传递

出了 s_1 的邻居范围

- 判断 x 点的状态是否是 passive: 若 $a(p) + r_l^{\leftarrow}(p) < d_l(s_0, p)$, 则 x 点的状态是 passive。
- 若节点都遍历完, 所有状态是 active 的节点都属于 s_0 的局部最短路径 DAG。

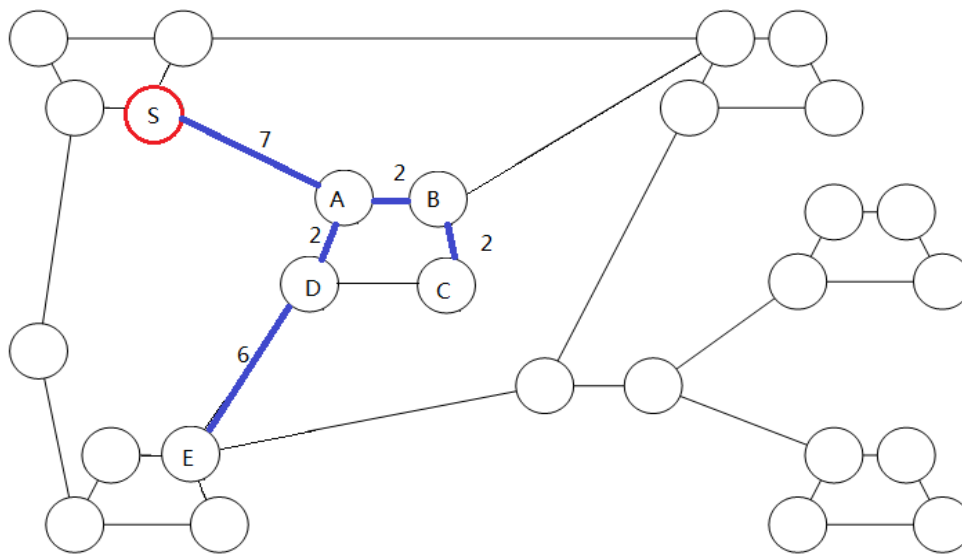
构建局部最短路径有向无环图

$$b'(x) = \begin{cases} d_l(s_0, x) + r_l^{\rightarrow}(x), & s_0 \in \wp(x) \\ 0, & \text{otherwise} \end{cases}$$

$$b(x) = \max(\{b'(x)\} \cup \{b(y) | y \in \wp(x)\})$$

$$a'(x) = \max\{a(x) | y \in \wp(x)\}$$

$$a(x) = \begin{cases} \max\{d_l(s_0, u) | y \in \wp(x) \wedge u \in \wp(y)\}, & a'(x) = \infty \wedge d_l(s_0, x) > b(x) \\ a'(x), & \text{otherwise} \end{cases}$$



点	b	a
S	0	∞
A	11	∞
B	11	∞
D	11	∞
C	11	∞
E	11	7

选择属于Highway的边

一条边 (u, v) 如果满足下述条件则属于Highway:

- (u, v) 在第一步中所得到的局部最短路径DAG中, 也即 (u, v) 在某一点 s_0 到 p 的最短路径 $\langle s_0, \dots, u, v, \dots p \rangle$ 上。
- $v \notin N_l^{\rightarrow}(s_0)$ 且 $u \notin N_l^{\leftarrow}(p)$ 。

我们对第一步中所得到的局部最短路径DAG中每一条边进行判断, 即可得到Highway。

选择属于Highway的边

对于 $u \in B$ ，我们定义 $\beta(u)$ 为由 u 及其后裔构成的集合。定义松弛变量 $\Delta(u) = \min_{\omega \in \beta(u)} (r_l^{\leftarrow}(\omega) - d_l(u, \omega))$ 。

很显然，对于一个叶子节点 b 来说， $\beta(b) = b$ ， $\Delta(b) = r_l^{\leftarrow}(b)$ 。

对于一个中间节点 u 来说，我们可以通过它的所有孩子节点 $\gamma(u)$ 得到 $\Delta(u)$ ：

$$\Delta(u) = \min(r_l^{\rightarrow}(u), \min_{c \in \gamma(u)} \Delta_c(u))$$

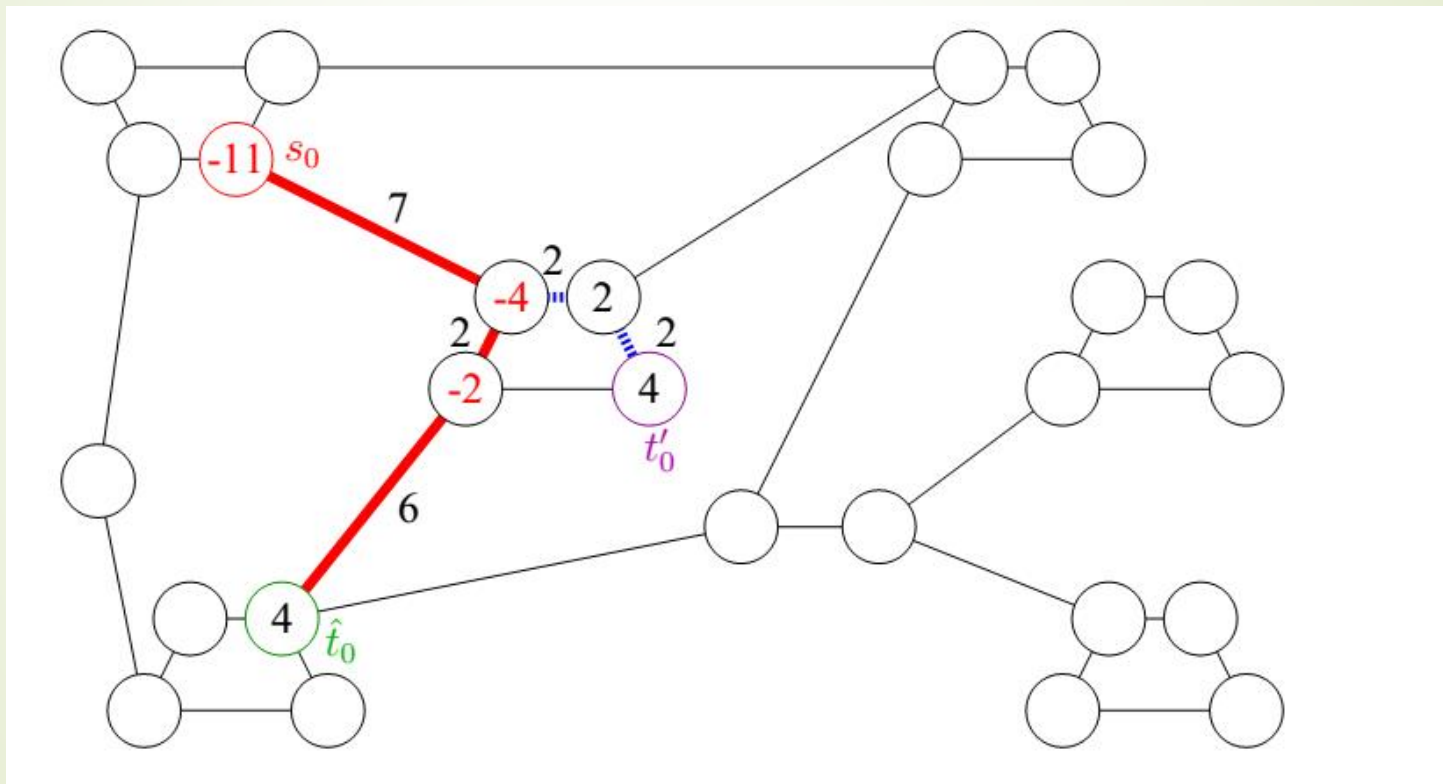
其中 $\Delta_c(u) = \Delta(c) - d_l(u, c)$ 。

所以我们可以递归的计算各点的松弛变量。

选择属于Highway的边

- 对于 $\forall u \in B$, $\Delta(u) = r_l^{\leftarrow}(u)$ 。
- 按第一步Dijkstra算法顺序反向遍历各点 u , 若 $u \in N_l^{\rightarrow}(s_0)$ 则结束算法, 若 $u \notin N_l^{\rightarrow}(s_0)$ 则进入第4步。
- 计算 $\Delta_u(p) = \Delta(u) - d_l(p, u)$, p 是 u 的父节点, 若 $\Delta_u(p) < 0$, 那么边 $(u, p) \in E_{l+1}$, 否则若 $\Delta_u(p) < \Delta(p)$, $\Delta(p) = \Delta_u(p)$, 返回第3步。

选择属于Highway的边





计算Highway的core

Highway core的计算能够大幅度降低建立Highway Hierarchy和查询路线的时间！可以让我们使用更小的 H_l 而不会影响Highway的收缩速率。但另一方面，由于捷径的添加，使得整个图的平均每点的度数增加，这也就意味着查询时最短路径的搜索会变慢，所以我们要谨慎的选择绕过的点（bypass nodes），来保证我们绕过这些点利大于弊。

计算Highway的core


我们规定当

$$\#shortcuts \leq c \cdot (\deg_{in}(u) + \deg_{out}(u)) \dots\dots (*) \text{ 时,}$$

我们绕过 u ，其中 $\#shortcuts$ 是去掉 u 后需要添加的捷径数； $\deg_{in}(u)$ 和 $\deg_{out}(u)$ 分别表示 u 的入度和出度； c 是可变参数，可以通过改变它来改变Highway收缩率。

算法：

- 将 V_l 内所有点初始化为一个栈。
- 若栈不为空我们进入第3步，若栈为空结束算法。
- 栈顶元素 u 出栈，判断其是否应该被绕过，若 $(*)$ 成立则进入第4步；若不成立则返回第2步。
- 移除 u ，并添加相应的捷径，遍历所有捷径对应的终点，若其已出栈则再压入栈中，返回第2步。



最短路径查询

- 基本思想
- 基本算法
- 算法优化
- 最短路径输出

基本思想

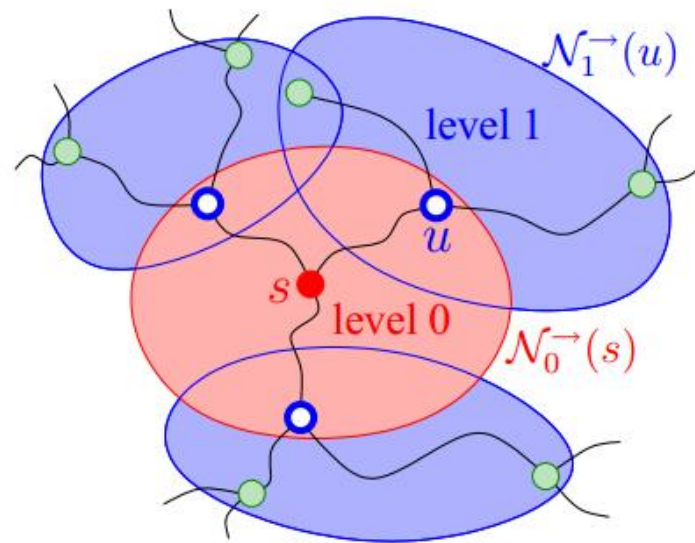
- 公路层次查询算法 (*highway query algorithm*) 是双向Dijkstra算法的一个变体。
- 对于搜索时的每个点来说，除了包含起点到它的距离 (Dijkstra算法所需的数据) 之外，还要包含搜索到它时的网络层次以及它到当前搜索边界的距离 (*the gap to the 'next applicable neighborhood border'*) :
 - 对每一 l 层第一个搜索的点 s_l 来说，它的gap等于它的邻居半径 (neighborhood radius) 。
 - 对其他的点 v 来说，它的gap等于它的父节点 u 的gap减去 (u, v) 的长度。

基本思想

当 v 的gap大于等于0时，说明我们仍然在 s_i 的邻居内进行搜索，这时我们只需继续经典的Dijkstra算法即可；当一个点 v 的gap小于0时，我们可以判断它已经不在 s_i 的邻居里了，这时将 v 的父节点 u 称作进入上一层网络的入口。对于 v 来说，它有可能在第 l 层网络中，也有可能在第 $l + k$ ($k > 0$) 层网络，这需要看 (u, v) 所在的网络层次：

- 如果 (u, v) 在第 l 层网络，我们将不对 v 进行松弛。这是因为我们的算法是双向的，而Highway Hierarchies的计算保证最短路的边的层次都是足够高的，所以我们不需要关心那些不在邻居范围内的低层次的点。这就是算法的约束1！
- 如果 (u, v) 在第 $l + k$ 层网络，那么我们将 v 的gap置为 v 到 u 邻居边界的距离，然后继续在第 $l + k$ 层网络中进行Dijkstra算法。

基本思想

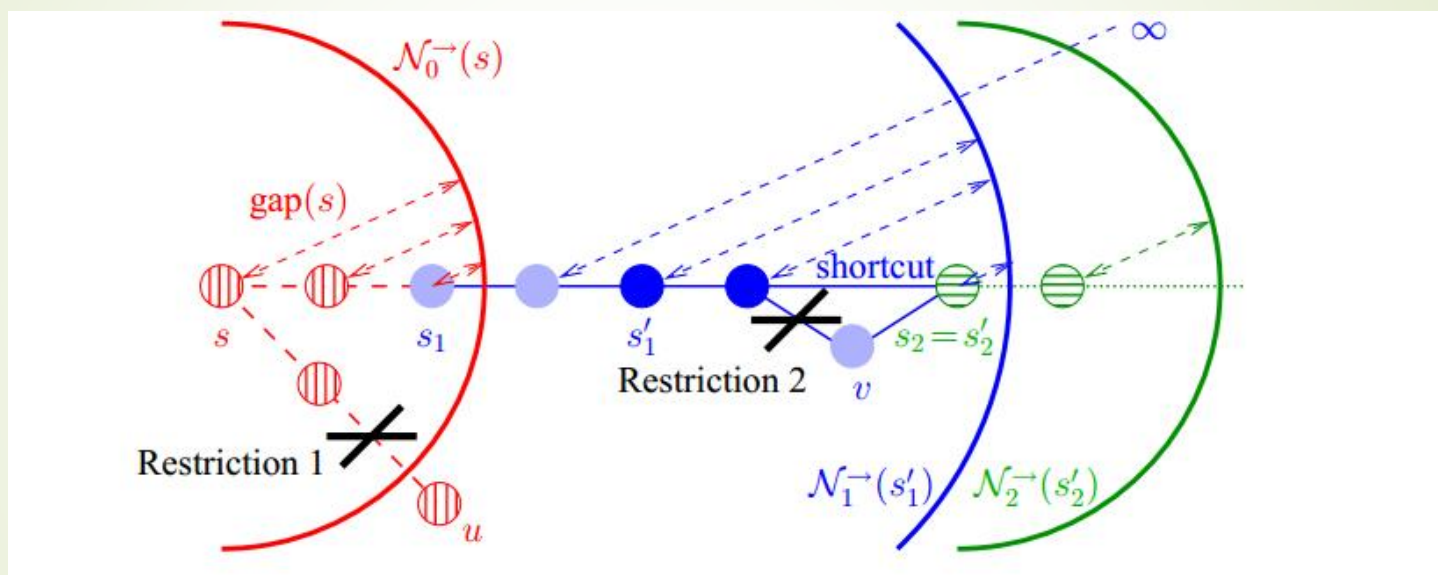


- entrance point to level 0
- entrance point to level 1
- entrance point to level 2

基本思想

- 除此之外，我们还要解决一个问题： l 层的入口不在该层网络的核心内，而是一个绕过的点（bypassed nodes）。这时候算法会在bypassed nodes中继续进行直到第 l 层的核心点 u 到达， u 的gap置为 u 在网络层次 l 中的邻居半径。注意：在 u 没到达之前，bypassed nodes的gap都是无穷大的。
- 在网络核心中我们仍然进行Dijkstra算法，但是要注意一点：当一点 $u \in V_l'$ 被搜索过了，任何一条指向一个bypassed node $v \in B_l$ 的边都不会被松弛；也就是说，当我们进入一个网络核心时，就不会再离开这个核心（除非要进入上一层网络）（约束2）。

基本思想



数据结构

- 对于图中每个点 u 我们都定义一个三元组 $(\delta(u), l(u), gap(u))$ ，并称之为 u 的键(key)：
 - $\delta(u)$ 表示起点（或终点）到 u 的距离；
 - $l(u)$ 表示搜索到 u 时的网络层次；
 - $gap(u)$ 表示 u 到下一个邻居边界的距离。
- 在优先队列中我们仅使用 $\delta(u)$ 作为优先级的判断依据。
- 对于一个点来说，如果它有两个 $\delta(u)$ 相同的key： $k := (\delta, l, gap)$ ， $k' := (\delta, l', gap')$ ，那么我们更希望使用 l 更大， gap 更小的key，也就是说若 $l > l'$ 或者 $l = l' \wedge gap < gap'$ ，我们选择 k 作为key。


算法描述

input: source node s and target node t

output: distance $d(s, t)$


```
1   $d' := \infty$ ;
2  insert( $\overrightarrow{Q}$ ,  $s$ ,  $(0, 0, r_0^{\rightarrow}(s))$ ); insert( $\overleftarrow{Q}$ ,  $t$ ,  $(0, 0, r_0^{\leftarrow}(t))$ );
3  while ( $\overrightarrow{Q} \cup \overleftarrow{Q} \neq \emptyset$ ) do {
4      select direction  $\Leftarrow \in \{\rightarrow, \leftarrow\}$  such that  $\overleftarrow{\overleftarrow{Q}} \neq \emptyset$ ;
5       $u := \text{deleteMin}(\overleftarrow{\overleftarrow{Q}})$ ;
6      if  $u$  has been settled from both directions then
           $d' := \min(d', \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u))$ ;
7      if  $\text{gap}(u) \neq \infty$  then  $\text{gap}' := \text{gap}(u)$  else  $\text{gap}' := r_{\ell(u)}^{\Leftarrow}(u)$ ;
8      foreach  $e = (u, v) \in \overrightarrow{E}$  do {
9          for ( $\ell := \ell(u)$ ,  $\text{gap} := \text{gap}'$ ;  $w(e) > \text{gap}$ ;
               $\ell++$ ,  $\text{gap} := r_{\ell}^{\Leftarrow}(u)$ ); // go "upwards"
10         if  $\ell(e) < \ell$  then continue; // Restriction 1
11         if  $u \in V_{\ell}' \wedge v \in B_{\ell}$  then continue; // Restriction 2
12          $k := (\delta(u) + w(e), \ell, \text{gap} - w(e))$ ;
13         if  $v$  has been reached then decreaseKey( $\overleftarrow{\overleftarrow{Q}}$ ,  $v$ ,  $k$ );
            else insert( $\overleftarrow{\overleftarrow{Q}}$ ,  $v$ ,  $k$ );
14     }
15 }
16 return  $d'$ ;
```

- Line 4: 算法的正确与否和这里选择正向队列或反向队列的策略无关，但是一个好的策略是能够加快算法的速度。
- Line 7: 如果一个点的gap为 ∞ ，那么这个点要么是一个不在core中的点要么是第一个进入core中的点。所以不管该点是哪一种，这样的处理都能正确的计算该点的gap。
- Line 9: 有可能这一步将网络层次上升几层。
- Line 13: 如果改变了 $\delta(u)$ ，则优先队列会改变，如果没有改变 $\delta(u)$ 仅仅改变了 $\ell(u)$ 和 $\text{gap}(u)$ ，则优先队列不变。
- 如果我们将 $u \in V_l'$ ， $v \in B_l$ 的边 (u, v) 的层次下降1，约束1就会自动包括约束2。



算法优化

- 重新排列顶点
- 预计算最高层网络
- 修改算法的终止条件



重新排列顶点

为了提高算法效率，我们可以根据顶点所在core的层次重新排列顶点，这样在高层次的core中搜索的效率会大大提升。

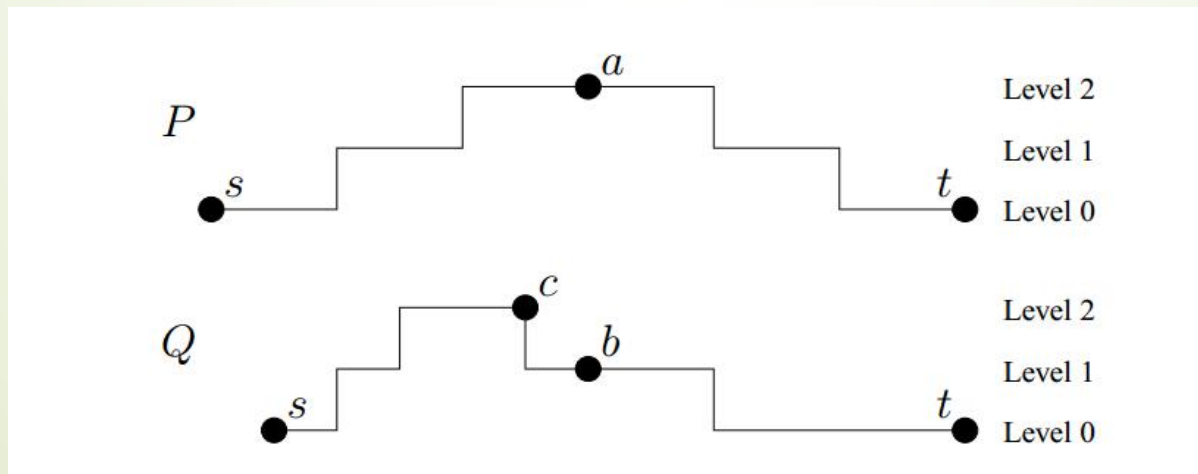


预计算最高层网络


- 从Highway Hierarchies的构造中，我们能看出查询的最短路大部分都会经过最高层网络，所以若我们预计算出最高层网络的每一条最短路，形成一个距离表，那么将会大大的提高算法的效率。
- 由于距离表的存在，我们不需要再最高层网络中搜索；当我们到达一个属于 V'_L 的点 u ，我们将它加入集合 \vec{I} 或 \overleftarrow{I} 中，而且我们也不松弛指向第 L 层的边。当所有 L 层的入口都搜索过之后，我们考虑 \vec{I} 和 \overleftarrow{I} 能形成的所有边 $(u, v) = \vec{I} \times \overleftarrow{I}$ ，那么起点到终点的最短距离即为 $\min(\vec{\delta}(u) + d_L(u, v) + \overleftarrow{\delta}(v), d')$ (PS: d' 是为了防止起点与终点有不经过 L 层的最短路)。

修改算法终止条件

- 在双向Dijkstra算法中，当算法的正向搜索和反向搜索相交时，我们就终止算法，所得的路径就是最短路径！而在上述的双向Highway Hierarchies搜索中，这样的结论并不成立。
- 采取一个保守的策略：当我们找到一条从起点到终点的路径 P' 后，所有 $\delta(u) \geq \omega(P')$ 的点 u 都不往下进行算法了。



上面的搜索路径时候对称的；下面的搜索路径不对称，从起点出发的路径到 c 点就停下了。



最短路径输出

- 基本算法只求出了两点之间的最短距离，要想求得两点之间的最短路径，只需在基本算法中稍加修改，将每个点的父节点都存储起来，形成一颗搜索树即可。
- 怎么将最高层网络中正向反向的路径连起来？（在优化了最高层网络距离表的情况下）
- 怎么将最短路中的捷径展开成原图中的路径？

最短路径输出

- 初始化 u 点为正向的入口， v 点为反向的入口。
- 遍历最高层网络距离表，对于每一条边 (u, ω) ，如果 $\omega = v$ ，则已经找到一条从 u 到 v 的最短路径，算法终止；如果 $d_L(u, \omega) + d_L(\omega, v) = d_L(u, v)$ ，则令 $u = \omega$ 。

最短路径输出

- 第二个问题可以直接做一次最短路算法，对于一条捷径 $(u, v) \in S_l$ ，我们直接在 G_l 上搜索 (u, v) 的最短路，注意 (u, v) 在 G_l 上的最短路可能包含低层次网络中的捷径，所以这个算法是递归的。
- 第二个问题还可以在每一条捷径中加入一定的信息，方便捷径的展开，这是一个以空间换取时间的例子。我们在每一条捷径中存储它经过的每一跳索引 (*hop indices*)，例如捷径经过边 (u, v) ，我们只需记录 (u, v) 是第几条以 u 为起点的边即可，当然这里面可能包含低层网络的捷径，所以最后的展开算法仍然是递归的！因为图中平均每点的度都不会很大，所以我们只需要用一个很小的空间就可以给每条捷径记录上述的信息。
- 因为最高层网络中的捷径很多时候都会被用到，所以我们可以将最高层网络中所有捷径的非递归信息都记录下来，最后展开最高层捷径时只需加以替换即可。