Customization Route Planning
算法介绍

CRP算法

由微软硅谷研究院三位科学家于 2013 年提出，应用于 Bing 地图路径规划

CRP算法的设计目标

- 应用于大陆级别的实际路网数据。
- 支持任意类型 metric（度量，可以理解为 cost function）。
- 响应时间满足实时查询需求。
- 快速的路况更新以及定制化 metric 更新。

CRP算法的主要思想

- CRP 算法将实际路网划分为拓扑结构和 metric 属性两部分。
- 拓扑结构由道路的一系列静态属性组成，包括道路长度，转向类型，车道数，道路类型，最大速度等。
- metric 属性代表经过一条道路或转向时的实际权重。
- 我们认为路网拓扑结构是各个 metric 通用的并且很少变化，metric 属性可能会经常变化并且可以是用户定制的。

CRP算法的主要思想

- CRP算法分为三阶段：

1. Metric-Independent Processing
   - 预处理路网拓扑结构，生成分割图（partition-based overlay graph）。
   - 这一过程计算复杂度较高，但运行频率低。
2. Metric Customization
   - 处理每个 metric 的时候都必须运行，要求执行速度快。
3. Query Stage
   - 响应时间需满足实时查询。

分割图的定义

- 设原图为 $G=(V,E)$，V 的一个划分是一个 cell 的集合 $S=\{C_0,C_1,\dots,C_k\}$，V 中的每个顶点 u 属于且仅属于一个 cell，设定 U 表示最大 cell 的大小。
- V 的多层划分是一个划分族 $\{S^0,S^1,\dots,S^L\}$，U^l 为 S^l 的最大 cell 大小。我们规定 $U^0=1$，$S^{L+1}=\{V\}$。
- 我们使用嵌套多层划分，即对于 $l\le L$ 的每个 cell $C_i^l\in S^l$，都存在 $C_j^{l+1}\in S^{j+1},C_i^l\subseteq C_j^{l+1}$，我们称 C_j^{l+1} 是 C_i^l 的 supercell（超单元），反之为 subcell（子单元）。
- 我们用 $c_l(u)$ 表示层级 l 中顶点 u 所在的 cell。

分割图的一些说明

- 简单来说，分割图包含若干层，每层包含若干单元。
- 第 0 层每个顶点表示一个单元，高层的一个单元包含低层的若干单元，顶层只有一个单元，包含所有顶点。
- 一个顶点属于且仅属于同一层的唯一一个单元。
- 每一层单元中的顶点数小于一个设定的参数 U（每层不同）。
- 处于单元边界的顶点记为**边界点**，同一层连接不同 cell 的边记为**边界弧**。

分割图的剪枝

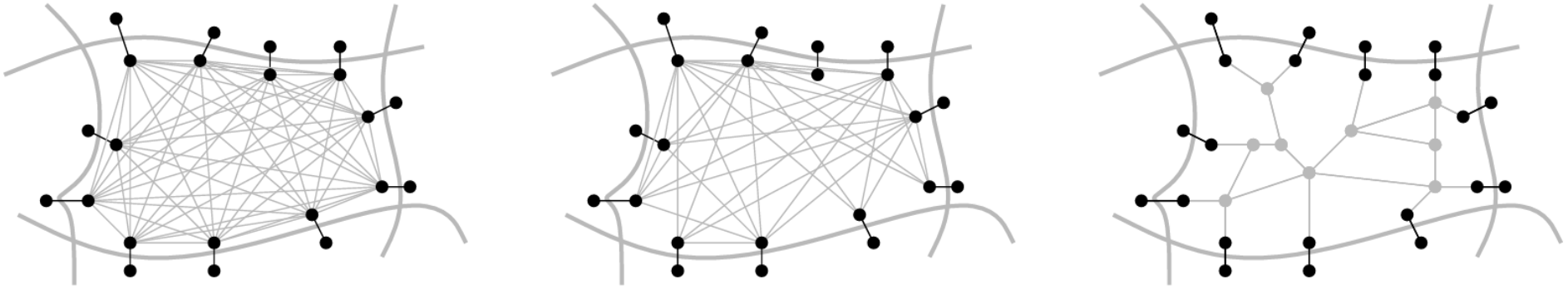

Fig. 1. Three possible ways of preserving distances within the overlay graph. Storing full cliques (left), performing arc reduction on the clique arcs (middle), and storing a skeleton (right).

- 对于每个单元，我们只保留其边界点，并在任意两边界点之间连边，边权为只经过单元内点的最短路。
- 左：保留完整的图。
- 中：若只经过单元内点的最短路不是原图最短路则删去该边。
- 右：求出以每个边界点为根，以其他边界点为叶子的的最小生成树，保留其并集。

针对 turn（转向）的建模

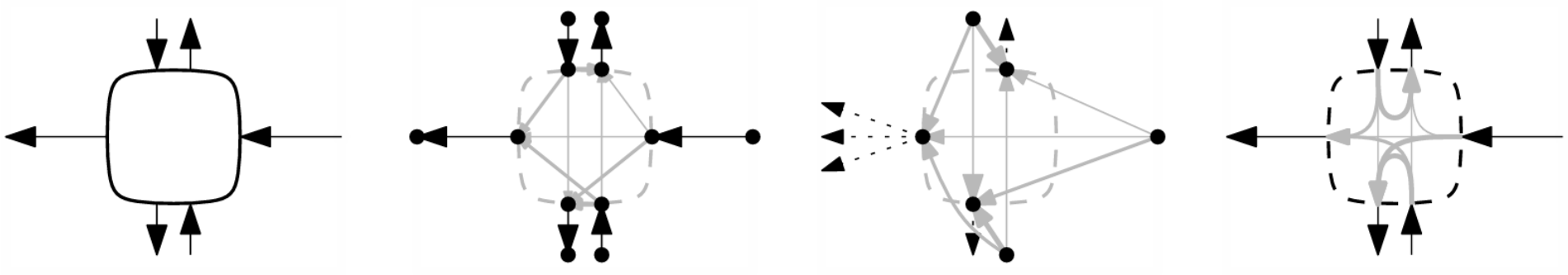

Fig. 2. Turn representations (from left): none, fully expanded, arc-based, and compact.

- 在路网中，顶点一般是一个路口，需要考虑转向的成本。
- None: 无转向。
- Fully expanded: 对每个出入口添加节点，按转向加边。
- Arc-based: 保留路口周围每条边的起点，新加的边表示一段道路加一次转向。
- Compact: 构建一个 $p \times q$ 的权值矩阵（p 代表入口个数，q 代表出口个数），若没有特殊说明，下文中的最短路均使用这种方案。

Metric-Independent Processing

对路网进行多级分区（partitioning），构建覆盖图的拓扑结构（overlay topology），并建立供定制化阶段使用的辅助数据结构。

Partitioning

- 将原图划分成一个若干层的分割图。
- CRP算法性能依赖于**边界弧**数量，因此目标是使**边界弧**的个数尽量少。
- 目前存在许多做法，如 METIS，SCOTCH，planar separators，grid partitions 等，但是这些算法不是为路网设计的，划分质量很差。
- 这里使用 PUNCH 算法，利用路网中的自然分割（河流，山川，高速公路）作为启发因子进行划分。

Overlay Topology 的过程

- 对原图 G 完成划分后，我们要构建覆盖图 H。
- 覆盖图的结构不一定与原图相同，但两点间最短路相等。
- 对于每个边界弧 (u, v)，根据对 turn 的建模，将原图 u, v 对应为出口点 u'_H 和入口点 v''_H，在覆盖图 H 中添加边 (u'_H, v''_H)。

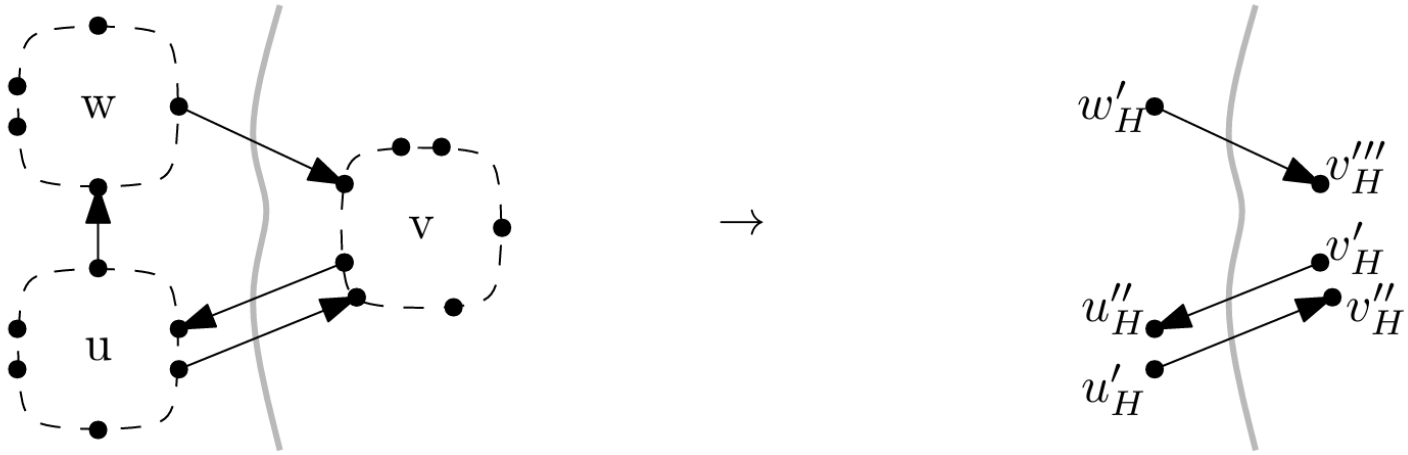

Fig. 3. Building the overlay graph from the cut arcs. Each cut arc (u, v) yields two vertices u'_H,v''_H in the overlay graph.

Overlay Topology 的一些说明

- u'_H, v''_H 是在覆盖图中的点，记为覆盖点，那么高层覆盖点一定包含在低层覆盖点中。
- 覆盖图只由覆盖点和覆盖边组成。
- 需要通过哈希实现原图与覆盖图顶点的双向映射。
- 对于每个单元，设有 p 个入口点，q 个出口点，存储一个 $p \times q$ 的矩阵 W，表示两点间最短路。
- 进一步优化，可以将矩阵存成一维的数组。

Overlay Topology 的存储

- 对于每个单元，在其 supercell 中分配一个唯一的连续标识符，由于层数和 supercell 中的单元数都较少，所以顶点 u 的所有标识符可以存储到一个 64 位整数 $pv(u)$ 中。
- 为进一步节省空间，可以只对第一层的每个单元存储 pv，单元内顶点共用，查询时先查找 u 在第一层属于哪个单元。
- 此数据结构需要 $4 \cdot n + 8 \cdot |C_1|$ 字节。

Metric Customization

度量定制阶段，会在 Query 的同时进行，因此需要足够快。

Metric Customization 的过程

- 定制阶段需要计算前文提到的 W 矩阵。
- 我们采用自底向上的方式逐层计算这些距离，设 H_i 为第 i 层分割图的覆盖图。
- 考虑 H_1 中的每个单元，对每个入口点，在原图中跑 Dijkstra（只在单元内），求出到每个出口点的最短路。
- 对于 $H_i(i > 1)$，在跑 Dijkstra 时，不在原图上，而是在 H_{i-1} 中对应于单元的子图。子单元内可以直接使用计算好的结果。

Metric Customization 的优化

1. 提高局部性：

   处理第 i 层的单元时，我们在整个覆盖图 H_{i-1} 计算，但将搜索范围限制在单元内的顶点。 事实上可以将子图提取出来，存储一个临时副本。这简化了搜索，并使内存访问连续。

2. 改变算法：

   可用 Bellman-Ford 算法替代 Dijkstra 进一步加速。在小规模图中（如定制阶段处理的图），Bellman-Ford 性能更加优秀。

Metric Customization 的优化

3. 搜索图剪枝：

处理第 $i(i > 1)$ 层覆盖图的单元时，我们可以将其 subcell 的出口点进行收缩，因其出度为 1，可以删去该点和出边，将所有入边加上其边权。

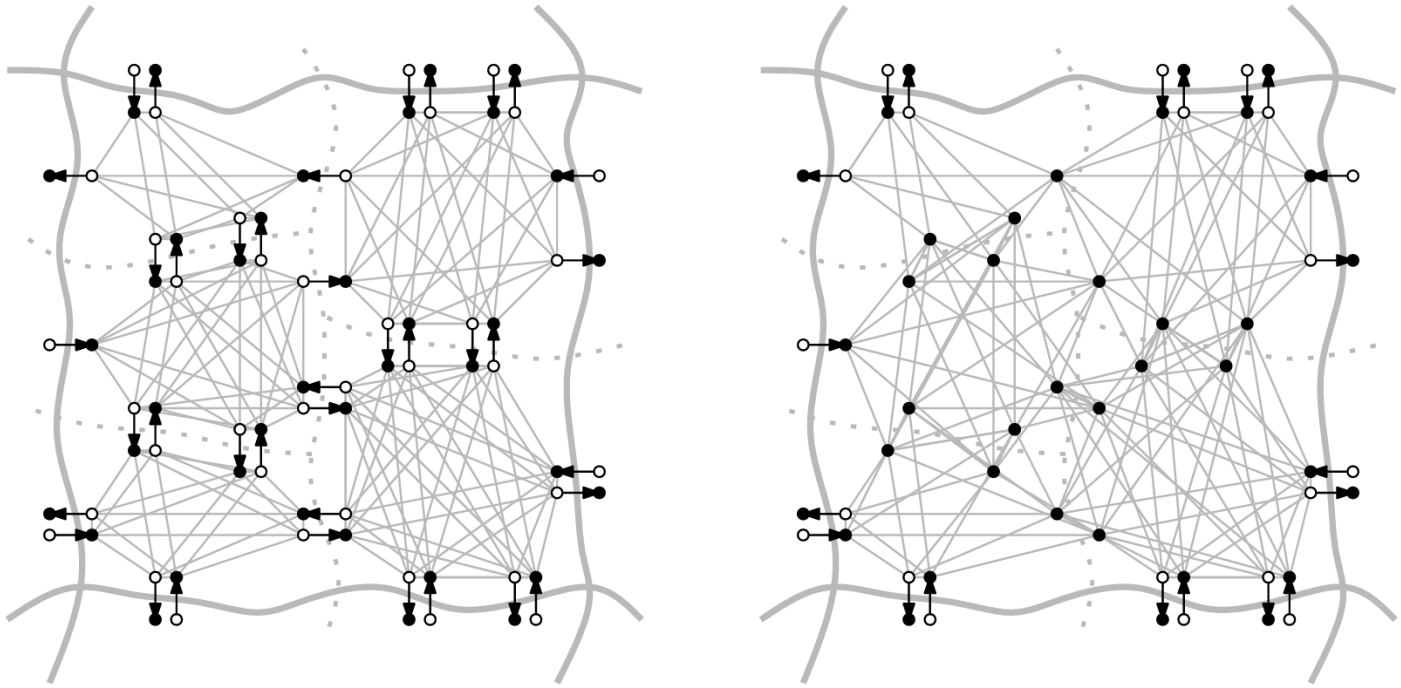

Fig. 4. The overlay graph before (left) and after (right) pruning.

Metric Customization 的优化

4. 多源并行执行：

   通过将多次 Dijkstra 运行合并为单次执行可加速这一过程，即同时从 k 个源点 $s_1, s_2, \ldots, s_k$ 进行搜索时，为每个顶点维护 k 个距离标签，表示所有源点到该点的最短路。此思想也可以应用于 Bellman-Ford。

5. 并行化：

   - 现代 CPU 支持 SIMD（单指令多数据）指令集，SSE 指令可操作 128 位寄存器，允许对四个 32 位字并行执行加法或比较。因此四个源点可以用一条 SSE 指令。
   - 核心级并行，将源点分组分配给 CPU 的不同核心。

Metric Customization 的优化

6. 虚设层级：

   实验表明，增加层级可以在一定程度内加速定制，但会增加空间开销，并且不会加速查询。

   因此引入虚设层级，仅用于加速定制过程，但不保留于查询过程。

7. 增量更新：

   在线地图服务需持续接收实时交通信息，在更新覆盖图时，若只有少数边的成本发生变化，可以仅识别所有受到影响的顶点所在的单元，并对这些单元重新进行定制。

Metric Customization 进一步优化

- 将预处理与度量定制相分离，使得 CRP 能够在现代服务器上以大约 10 秒钟的时间在大陆级道路网络中引入新的成本函数。
- 这对于实时交通而言已经足够，但仍不足以实现用户在线定制成本函数。
- 为进一步加速，就需要加快其基本操作：计算每个单元内的最短路。
- 引入 CH 算法的核心操作——收缩 (contraction)。

收缩操作介绍

- 我们不再显示用 Dijkstra 计算最短路，而是通过不断删除单元内的顶点，并添加新边来保持距离。最终保留下来的边就是两边界点之间的最短路。
- 当收缩顶点 v 时，假设存在边 (u,v) 和 (v,w)，那么移除这两条边，新边 (u,w)，边权为原来两边之和。
- 但只有当新边 (u,w) 为单元内去掉点 v 后唯一的最短路时才会添加。
- 为了加速收缩过程，我们需要预计算顶点收缩顺序，对于不同的 metric 使用同样的顺序。

收缩顺序

- 最小化新增最短路数量的收缩顺序是 NP-Hard 问题。
- 实现中使用在线启发式策略：根据顶点局部属性动态选择下一个收缩顶点。
- 我们测试了多种贪心函数，考虑参数入边数 $ia(v)$，出边数 $oa(v)$，收缩时新增边数 $sc(v)$。
- 我们发现选择使 $h(v) = 100sc(v) - ia(v) - oa(v)$ 最小的顶点效果较好

收缩顺序

- 还可以通过结合图拓扑信息可以获得更优顺序。
- 利用划分指导收缩顺序，在预处理阶段创建额外**引导层级**，将标准划分向下延申至更小单元。
- 我们将层级 1 的单元（最大为 U）递归划分为大小为 $\frac{U}{\sigma_i}(i = 1,2,\dots)$ 的嵌套子单元，直至单元过小，其中 $\sigma > 1$ 为**引导步长**。
- 对层级 1 单元内每个顶点 v，设 $g(v)$ 为使其成为引导层级边界点的最小 i（理论上更重要的点 $g(v)$ 更小）。
- 若 $g(v) > g(w)$，则先收缩 v，否则比较 $h(v), h(w)$。

微指令（Microinstructions）

- 虽然收缩顺序再预处理阶段确定，但实际收缩操作需要在定制阶段进行。即使顺序已知，执行过程复杂度依然很高。
- 每次计算最短路显然不可接受，因此我们假设所有新边都需要被添加，即不再计算最短路。
- 收缩的基本操作十分简单，读取两条边成本，求和，与第三条边成本比较并更新。可完全由三元组 (a, b, c) 描述，表示存储三条边成本的内存地址，读取地址 a, b 的值，将和写入 c。
- 由于操作序列对任意 cost function 都相同，我们可以在预处理阶段为每个单元设置**指令数组**。

收缩过程说明

- 在这一过程中，图本身被抽象化了，我们完全没有顶点和边的概念。
- 我们只会执行指令数组，并在内存中进行相应的操作。
- 这样比在实际图上操作更加快速，也更简单。
- 将收缩与 Bellman-Ford 结合起来，时间可以缩短到一秒以内，但空间开销会增大。

Query Stage

查询阶段：双向 dijkstra 算法

- 在查询 s 到 t 的最短路时，传统的单源最短路算法 dijkstra 过程中极有可能会访问到大量无意义的点，所以我们采用双向 dijkstra 来代替传统的 dijkstra 算法。
- 算法流程：从 s 和 t 同时开始跑最短路，记录 $d_s(x)$ 表示从 s 到 x 的最短路，$d_t(x)$ 表示从 x 到 t 的最短路，也就是把边的方向全都改变后，从 t 到 x 的最短路。两个过程分别记为前向搜索和后向搜索。维护两个优先队列，同时在全局记录目前的最优解 μ。每次更新的时候，两个队列同时取队头，并更新所有的邻居，每次一个点 x 的最短路被更新时，令 $\mu \leftarrow \min\left(\mu, d_s(x) + d_t(x)\right)$。当两个队头的最短路之和超过 μ 时停止执行。

查询阶段：compact 图上的 dijkstra

- compact 图不能简单地为每一个顶点（路口）维护它到起点的距离，而是要对顶点的每一个入口点维护。
- 在优先队列中维护三元组 (v, i, d)，其中 v 为顶点编号，i 表示 v 的入口点的序数，即 v 的第 i 个入口点，d 表示距离。
- 这样的维护方式相比于简单地对每个顶点维护其最短路要慢很多。

查询阶段：compact 图上的 dijkstra

- 考虑优化，为每个入口维护隐式最短路 $b_v(i)$，初值为 ∞。每次从队列中取出 (v,i,d) 时，对每个 k 更新其隐式最短路。
- $b_v(k) \leftarrow \min\big(b_v(k), d + \max(T_v[i,j] - T_v[k,j])\big)$
- 其中 $T_v[i,j]$ 表示从 v 的第 i 个入口点转到第 j 个出口点的成本。
- 之后当 (v,k,d') 即将被推入队列时，若 $d' \le b_v(k)$，则将其推入队列，否则不推入。
- 证明：若 $d' > b_v(k) \ge d + T_v[i,j] - T_v[k,j]$，即对于任意 j，$d' + T_v[k,j] \ge d + T_v[i,j]$，所以即便被推入队列，也不会起到任何作用。
- 预处理 $c_v[i,k] = \max(T_v[i,j] - T_v[k,j])$，更新每个入口点时可以 $O(1)$ 更新。

查询阶段

- 实际查询时，起点和终点通常并不是顶点，而是在一条边上的某个位置。
- 对于前向搜索，将起点所在边的尾顶点初始化为起点到它的距离，并将其推入前向搜索的队列中。同理，后向搜索只需要将终点所在边的头顶点初始化为两点间距离，并将其推入后向搜索的队列中。接下来正常运行双向 dijkstra 即可。

查询阶段

- 之前的部分属于基础知识，接下来才是介绍 CRP 中的查询阶段
- 我们要完成的问题是：给定起点 s 和终点 t，可能是顶点，也可能是道路的某个位置，求在一些给定度量下的最优路径，以及一些接近最优路径的替代路径。

查询阶段：基础算法

- 定义 $l_{st}(v)$ 表示最大的满足 v 与 s 和 t 都不在同一单元的层级。即 $l_{st}(v)$ 为满足 $C_i(v) \cap \{s,t\} = \emptyset$ 的最大的 i。
- 计算 $l_{st}(v)$，可以取 $pv(s)$ 和 $pv(v)$ 的最高不同位，记为 $l_s(v)$，表示最高的满足 v 与 s 不在同一单元的层级。同理取 $pv(t)$ 和 $pv(v)$ 的最高不同位记为 $l_t(v)$。$l_{st}(v) = \min\big(l_s(v), l_t(v)\big)$。
- 进行双向 dijkstra，每次从队列中取出对头时分为两种情况：
- 1. 非覆盖顶点，即原图上的某个入口点，此时运行 compact 图上的 dijkstra。
- 2. 覆盖顶点，则在 $H_{l_{st}(v)}$ 上运行不包含转向的 dijkstra。

查询阶段：基础算法

- 一些优化：
- 若覆盖图中一个顶点 u 只有一条出边 (u, v)，则更新 u 时不把 u 推入队列中，直接更新 v。
- 双向搜索时，前向搜索访问矩阵是按行访问的，速度不受影响，后向搜索则按列访问，可以存相应的转置矩阵以提升速度。不过空间开销极大，所以最终没有使用这一优化。

查询阶段：路径解包

- 算出最短路后，我们还需要找到对应方案。
- 采用递归的方式解包路径：对于第 i 层的一条边，在层级 $i-1$ 的对应单元内运行 dijkstra 找到对应的路径。
- 加速解包：在度量定制阶段，可以预处理出在 $i-1$ 层中，所有出现在第 i 层的边所对应的最短路中的边，在解包时只考虑这些边即可。

查询阶段：替代路径

- 在道路网络中，用户通常希望获得多条合理路径选项，而不仅仅是严格意义上的最短路径。例如，可能存在多条时间或成本相近的路线，或需要满足额外约束（如避开收费站、偏好风景路线）。
- 容许路径：一条路径 P 是 $(\alpha, \gamma, \epsilon) -$ 容许路径，需满足以下条件：

1. 有限共享：P 与最短路径 Opt 的公共弧总成本不超过 $\gamma \times dist(s,t)$。

2. 局部最优性：P 中任何长度小于 $\alpha \times dist(s,t)$ 的路径必须是最短路径

3. 有界拉伸：$l(P) < (1+\epsilon) \times dist(s,t)$。

典型参数：$\alpha = 0.80, \gamma = 0.25, \epsilon = 0.3$。

查询阶段：替代路径

- 生成一条替代路径的核心思想是，通过经由点 v 生成候选路径，即 $s \to v \to t$，其中 $s \to v$ 和 $v \to t$ 都是最短路径。
- 算法流程：

1. 放宽双向搜索终止条件为，两队列对顶元素的距离和超过 $(1+\epsilon) \times \mu$（$\mu = dist(s,t)$）。

2. 计算每个点 v 的成本为 $l(v) = dist(s,v) + dist(v,t)$，与 Opt 的公共部分成本为 $\sigma(v)$，P_v 中所有满足 $dist(s,u) + dist(u,t)$ 的 u 构成的子路径的总成本 $pl(v)$。

3. 保留满足条件的 v：$l(v) < (1+\epsilon) \times \mu$，$\sigma(v) < \gamma \times \mu$，$pl(v) > \alpha \times \mu$。

4. $f(v) = 2l(v) + \sigma(v) - pl(v)$，取 $f(v)$ 最小的 v。

查询阶段：替代路径

- 有时可能需要不止一条替代路径。
- 生成多条替代路径步骤：

1. 取最短路径 Opt 为 P_1。

2. 选取替代路径 P_i 时，与 $P_1, \cdots, P_{i-1}$ 分别验证“有限共享”这一条件，即 P_i 与 P_j 的共享边成本和不超过 $\gamma \times l(p_j)$。

3. 不停操作直到达到所需数量或无满足条件的路径为止。

实验

- 下表为 Dijkstra，CH 和 CRP 三种算法在欧洲路网上运行的结果，共 1.8×10^7 个顶点，4.2×10^7 条边，测试机器运行Windows Server 2008 R2，配备96 GiB DDR3-1333内存和两颗6核Intel Xeon X5680 3.33 GHz CPU（每颗6×64 KB L1缓存、6×256 KB L2缓存、12 MB共享L3缓存）。

			Dijkstra		CH				CRP					
	graph		queries		prepro		queries		prepro		custom		queries	
metric	data structure	space [MiB]	scans [$\times 10^6$]	time [ms]	time [s]	space [MiB]	nmb. scans	time [ms]	time [s]	space [MiB]	time [s]	space [MiB]	nmb. scans	time [ms]
dist	no turns	408	9.3	1779	726	270	858	0.87	654	411.8	1.04	71.0	2942	1.91
time	no turns	408	9.4	2546	109	196	280	0.11	654	411.8	1.05	71.0	2766	1.65
	fully-blown	2739	42.7	13306	1967	2682	409	0.20	–	–	–	–	–	–
	arc-based	1620	21.6	7888	1392	1520	404	0.19	–	–	–	–	–	–
	compact	445	15.1	5582	1753	642	1998	2.27	654	411.8	1.10	71.0	3049	1.67

实验

- 观察实验数据得出如下结论：
- 对于 Dijkstra 和 CH 算法，引入转向成本会导致性能显著下降（Dijkstra 速度降低 2.2-5.2 倍，CH 预处理时间增加一个数量级），但 CRP 受其影响较小。
- CH 在距离下预处理和查询性能下降明显，而 CRP 在距离和时间下的查询均稳定在 2ms 以内。
- CRP 定制阶段仅需 1 秒，支持实时交通更新；CH 预处理需要 654 秒，无法满足动态场景需求。

总结

- CRP 算法在加入转向成本的情况下进行查询，速度几乎是 传统 Dijkstra 算法的 3000 倍。
- CRP 算法能够将度量无关预处理与度量定制阶段分离。这使得它能够在 1 秒内处理全新的度量，速度远超以往任何方法。这不仅实现了实时交通更新等关键功能，还支持个性化行车路线。
- 两大优势使得 CRP 算法成为地图软件的理想选择。

参考资料

- Delling, Daniel, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks*. 2013.
- Delling, Daniel, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. 2010.

感谢聆听