Shortest Paths and Experimental Evaluation of Algorithms

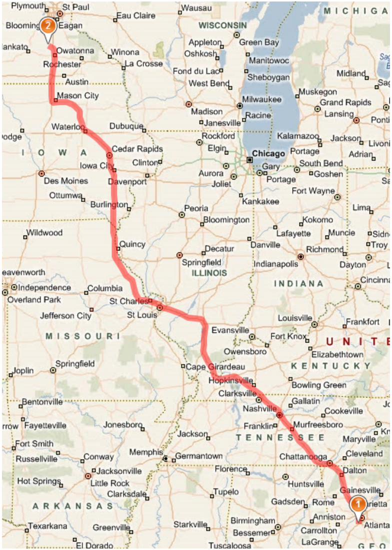
Renato F. Werneck

Microsoft Research Silicon Valley

MIDAS, August 2010

The Shortest Path Problem

- Input:
  - directed graph $G = (V, A)$;
  - arc lengths $\ell(v, w) \geq 0$;
  - $|V| = n$, $|A| = m$;
  - source s, target t.
- Goal: find **shortest path** from s to t.
  - its length is denoted by $\mathrm{dist}(s, t)$.
- Our focus is on **road networks**:
  - V: intersections;
  - A: road segments;
  - $\ell(\cdot, \cdot)$: typically travel times.



Outline

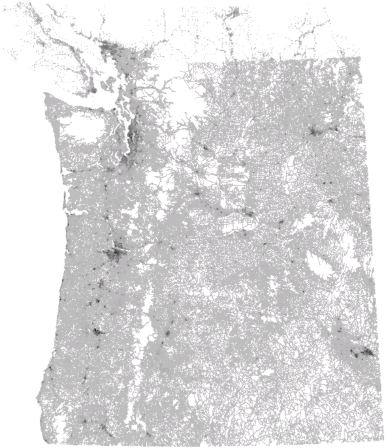
1. Dijkstra's algorithm
2. Basic data structures
3. Acceleration techniques:
   - A^* search and landmarks
   - reach-based routing
   - contraction hierarchies
   - arc flags
   - transit node routing
4. Highway dimension

Test Instance: USA Road Network

- $n = 24M$ vertices, $m = 58M$ arcs.
- Arc lengths represent travel times.

Test Instance: Northwestern USA

- $n = 1.65M$ vertices, $m = 3.78M$ arcs [GKW06];
- Arc lengths represent travel times.

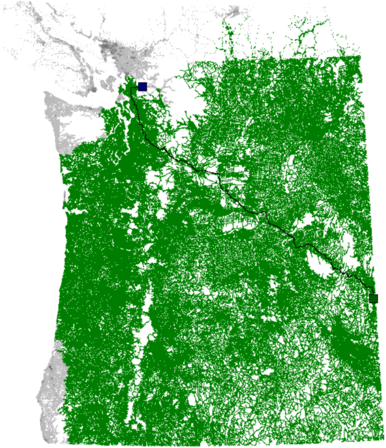


Dijkstra's Algorithm

- Intuition:
  - process vertices in increasing order of distance from the source;
  - stop when reaching the target.

Northwestern USA

- Dijkstra's algorithm:



Dijkstra's Algorithm

foreach $(v \in V)$ $d[v] \leftarrow \infty$;
$d[s] \leftarrow 0$;
`Q.Insert`$(s, 0)$; `// Q: priority queue`
while (!`Q.IsEmpty`()) {
    $v \leftarrow$ `Q.ExtractMin`(); `// v has smallest distance label`
    foreach (v, w){ `// scan vertex v`
        if $(d[w] > d[v] + \ell(v, w))${ `// found better path to w?`
            $d[w] \leftarrow d[v] + \ell(v, w)$;
            if $(w \in Q)$ **then** `Q.DecreaseKey`$(w, d[w])$;
            else `Q.Insert`$(w, d[w])$;
        }
    }
}

Dijkstra's Algorithm: Analysis

- Correctness:
  - we always take unscanned vertex with minimum $d(v)$;
  - edge lengths are nonnegative $\Rightarrow d(v)$ is exact (cannot be improved).
- Running time depends on priority queue operations:
  - $O(n)$ `Insert`;
  - $O(n)$ `ExtractMin`;
  - $O(m)$ `DecreaseKey` (one per arc).
- $O(m + n \log n)$ total time with Fibonacci heaps:
  - $O(\log n)$ time for `ExtractMin`;
  - $O(1)$ for `Insert` and `DecreaseKey`.

Dijkstra's Algorithm: *d*-heaps

- Binary heaps are good enough:
  - $O(\log n)$ time per operation:
  - $O(m \log n)$ time in total;
  - simpler than Fibonacci, often faster in practice.
- Road networks: $m = O(n)$ (almost planar).
- 4-heaps often work better:
  - similar to binary heaps, but each element has 4 children;
  - fewer levels, more elements per level;
  - better locality.

Dijkstra's Algorithm: Data Structures

- Dijkstra's algorithm on Europe with travel times:

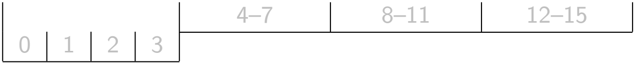
DATA STRUCTURE	SECONDS
2-heap (binary)	12.38
4-heap	11.53
8-heap	11.52

(Times on 2.4-GHz AMD Opteron with 16 MB of RAM.)

- Times are for building full trees:
  - about half for random *s*–*t* queries;
  - stop when *t* is about to be scanned.

Dijkstra's Algorithm: Multi-level Buckets

- Multi-level buckets (MLB):
  - put elements in **buckets** according to their values;
  - keep most elements in "wide" buckets (range of values);
  - first nonempty "wide" bucket split into "narrow" buckets as needed;
  - always remove from narrowest buckets;
    - ★ assumes `ExtractMin` is monotonic, as in Dijkstra's algorithm.



- The **caliber** acceleration for Dijkstra [Gol08]:
  - *caliber*(v): minimum incoming edge of v;
  - Let x be the latest vertex scanned by the algorithm;
  - Fact: if $d(v) < d(x) + \mathit{caliber}(v)$, $d(v)$ is exact.
    - ★ safe to scan v, even if not yet minimum!
  - MLB saves operations by identifying such vertices early.

Dijkstra's Algorithm: Data Structures

- Dijkstra's algorithm on Europe with travel times:

DATA STRUCTURE	SECONDS
2-heap (binary)	12.38
4-heap	11.53
8-heap	11.52
multi-level buckets	9.36
multi-level buckets + caliber	8.04

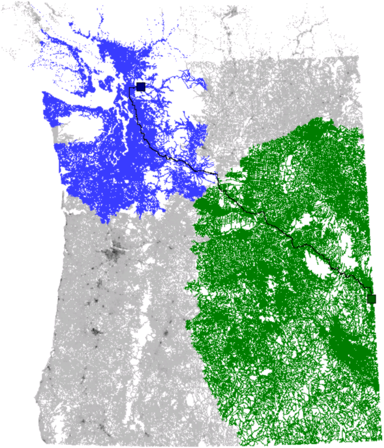
- Little hope for much better data structures:
  - MLB+caliber is within a factor of 2.5 of BFS [Gol08].

Bidirectional Dijkstra

- Bidirectional Dijkstra:
  - run **forward** Dijkstra from s with distance labels $d_f(v)$;
  - run **reverse** Dijkstra from t with distance labels $d_r(v)$.
  - alternate in any way.
- Keep track of best path μ seen so far:
  - path minimizing $d_f(v) + \ell(v, w) + d_r(w)$.
- Stop when some vertex x is about to be scanned twice.
  - return μ.

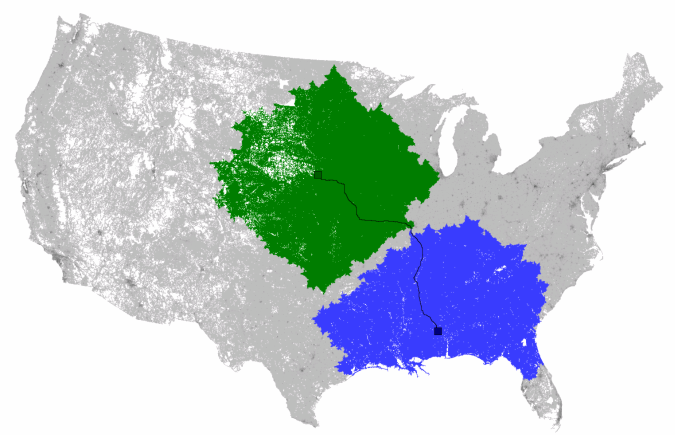
Northwestern USA

- Bidirectional Dijkstra:



USA Map

- Bidirectional Dijkstra:



Two-Stage Algorithms

- On road networks:
  - Bidirectional Dijkstra only twice as fast as Dijkstra;
  - we would like to do (much) better.
- We consider two-stage algorithms:
  - Preprocessing:
    - executed once for each graph;
    - may take a lot of time;
    - outputs some auxiliary data.
  - Query:
    - may use the preprocessed data;
    - executed once for each (s,t) pairs;
    - should be **very** fast (real time).

Two-Stage Algorithms

- Running Dijkstra:
  - preprocessing: do nothing;
  - query: run Dijkstra.
- Full precomputation:
  - preprocessing: compute $n \times n$ distance table:
    - time: $n \times$ *Dijkstra* (Europe: about 5 years);
    - space: $n \times n$ distance table (Europe: about 1 petabyte);
  - query: one table lookup.
- Both cases are too extreme.

Two-Stage Algorithms

- We want something in between:
  - preprocessing in minutes/hours;
  - linear amount of preprocessed data;
  - queries in real time.
- Lots of research in the past decade:
  - algorithm engineering;
  - we'll study the main ideas.

A^* Search

A* Search

- Take any **potential function** $\pi(v)$ mapping vertices to reals.
- It defines a **reduced cost** for each arc:
  - $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$;
- Fact: replacing ℓ by ℓ_π does not change shortest paths.
  - Take any path $P = (s = v_0, v_1, v_2, v_3, \ldots, v_k, t = v_{k+1})$:

$$
\begin{aligned}
\ell_\pi(P) &= \ell_\pi(s, v_1) + \ell_\pi(v_1, v_2) + \ell_\pi(v_2, v_3) + \ldots + \ell_\pi(v_k, v_t) \\
&= \ell(s, v_1) - \pi(s) + \pi(v_1) + \\
&\quad\ \ell(v_1, v_2) - \pi(v_1) + \pi(v_2) + \\
&\quad\ \ell(v_2, v_3) - \pi(v_2) + \pi(v_3) + \\
&\quad\ \ldots + \\
&\quad\ \ell(v_k, t) - \pi(v_k) + \pi(t) \\
&= \ell(P) - \pi(s) + \pi(t)
\end{aligned}
$$

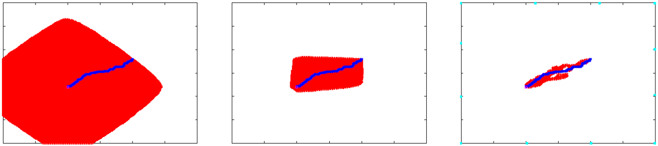
  - lengths of all s–t paths change by same amount $(-\pi(s) + \pi(t))$.

A* Search

- A* search ≡ Dijkstra on graph G_π:
  - same as G, with $\ell(\cdot,\cdot)$ replaced by **reduced cost** $\ell_\pi(\cdot,\cdot)$.
  - on each step, picks v minimizing $\ell(P_{sv}) - \pi(s) + \pi(v)$.
  - $\pi(s)$ is the same for all v!
- Equivalent: use $\ell(\cdot,\cdot)$, scanning **most promising** vertices first:
  - increasing order of $k(v) = d(v) + \pi(v)$.
  - $k(v)$: estimated length of shortest s–t path through v.
  - $d(v)$: estimate on $\text{dist}(s, v)$;
  - $\pi(v)$: estimate on $\text{dist}(v, t)$.
- Correctness requires $\ell_\pi \geq 0$:
  - potential function is **feasible**;
  - gives **lower bounds** if $\pi(t) \leq 0$.
- Effect: **goal-directed** search.

A* Search: Performance

- $\pi(v)$ gives lower bounds on $\mathrm{dist}(v, t)$.
- Performance:
  - Worst case: $\pi(v) = 0$ for all v (same as Dijkstra);
  - Best case: $\pi(v) = \mathrm{dist}(v, t)$ for all v:
    - $\ell_\pi(v, w) = 0$ if on shortest s–t path, positive otherwise;
    - search visits only the shortest path.
  - Theorem [GH05]: tighter lower bounds $\rightarrow$ fewer vertices scanned.
- Could use Euclidean-based lower bounds, for example.
  - we will see better methods shortly.



A^* Search: Bidirectional Version

- Bidirectional A^* search needs two potential functions:
  - forward: $p_f(v)$ estimates $\text{dist}(v, t)$;
  - reverse: $p_r(v)$ estimates $\text{dist}(s, v)$.

- Problem: different forward and reverse costs!
  - $\ell_{p_f}(v, w) = \ell(v, w) - p_f(v) + p_f(w)$ (arc scanned from v);
  - $\ell_{p_r}(v, w) = \ell(v, w) - p_r(w) + p_r(v)$ (arc scanned from w);
  - We need $\ell_{p_f}(v, w) = \ell_{p_r}(v, w)$:
    - must have $p_f(w) + p_r(w) = p_f(v) + p_r(v) =$ **constant**;
    - functions are **consistent**.

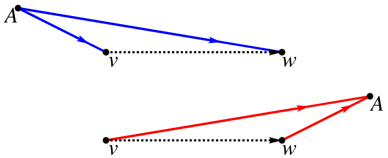
- Solution: use **average** potential function [IHI+94] instead:
  - $\pi_f(v) = \frac{p_f(v) - p_r(v)}{2}$
  - $\pi_r(v) = \frac{p_r(v) - p_f(v)}{2} = -\pi_f(v)$.
  - Now $\pi_r(u) + \pi_f(u) = 0$ for every u.

A^* Search: The ALT Algorithm

- Two-phase algorithm.
- Preprocessing:
  1. pick a few (e.g., 16) vertices as **landmarks**;
  2. compute distances between landmarks and **all** vertices;
  3. store these distances.
- Query: A^* search with landmarks using triangle inequality.
- $\mathbf{A}^*$ + **L**andmarks + **T**riangle inequality: **ALT** [GH05, GW05]

ALT Queries

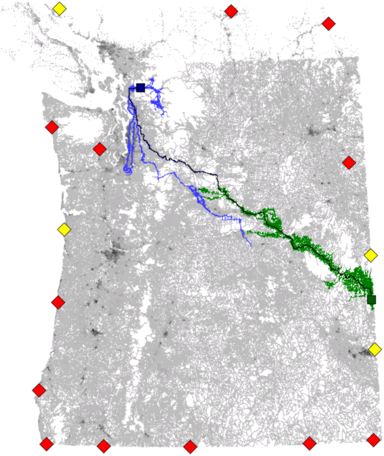
- Use triangle inequality for lower bounds:
  - $\text{dist}(v, w) \geq \text{dist}(A, w) - \text{dist}(A, v)$
  - $\text{dist}(v, w) \geq \text{dist}(v, A) - \text{dist}(w, A)$
  - $\text{dist}(v, w) \geq \max\{\text{dist}(A, w) - \text{dist}(A, v), \text{dist}(v, A) - \text{dist}(w, A)\}$.



- More than one landmark: pick best (maximum) over all.
  - more landmarks $\Rightarrow$ better bounds, more memory
- A good landmark appears "before" v or "after" w.

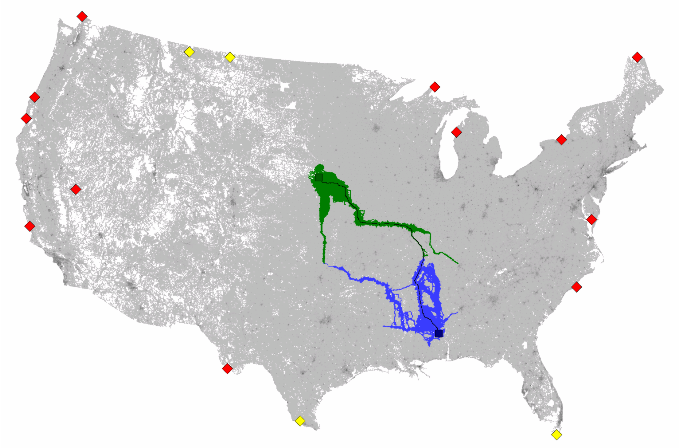
Northwestern USA

- ALT: 16 landmarks, 4 (yellow) used for this search.



ALT: An Example

- ALT:

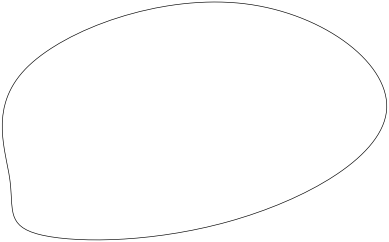


ALT: Selecting Landmarks

- A good landmark for an s–t query appears “before” s or “after” t.
- We must pick landmarks that are OK for all queries.
- Picking landmarks around the border seems reasonable.
- Several techniques have been tested:
  - random;
  - planar;
  - avoid;
  - maxcover.

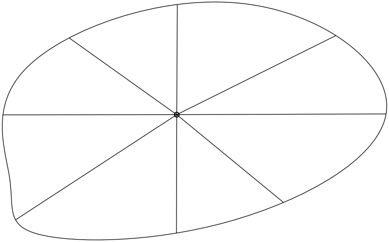
ALT: Planar Landmarks

- The **planar** algorithm:



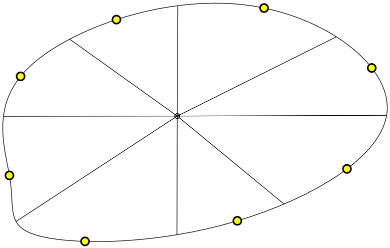
ALT: Planar Landmarks

- The **planar** algorithm:
  1. divide map into k equal-sized slices;



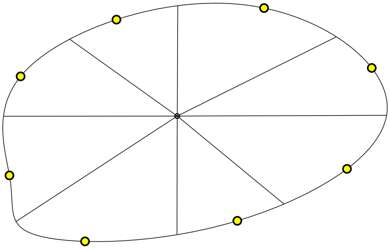
ALT: Planar Landmarks

- The **planar** algorithm:
  1. divide map into k equal-sized slices;
  2. pick farthest vertex in each slice as a landmark



ALT: Planar Landmarks

- The **planar** algorithm:
  1. divide map into k equal-sized slices;
  2. pick farthest vertex in each slice as a landmark
- Works well only if map is well-shaped:
  - not very good in practice.



ALT: Avoid Landmarks

- The **avoid** selection method (informally):
  - adds one landmark at a time;
  - prefers regions badly covered by previous landmarks.

- To pick a new landmark (less informally):
  1. pick a root r at random;
  2. build the shortest path tree T_r from r;
  3. pick a subtree T_w of T_r such that:
     - ★ T_w does not contain a landmark;
     - ★ T_w has many vertices;
     - ★ for $v \in T_w$, existing landmarks give bad bounds on $\text{dist}(r, v)$;
  4. pick a leaf of T_w as the new landmark.

ALT: Avoid Landmarks

- The **avoid** selection method (informally):
  - adds one landmark at a time;
  - prefers regions badly covered by previous landmarks.
- To pick a new landmark (almost formally):
  1. pick a root r at random;
  2. build the shortest path tree T_r from r;
  3. define for each node v:
     - $LB(v)$: lower bound on $\text{dist}(r, v)$ using landmarks already picked;
     - $weight(v)$: $\text{dist}(r, v) - LB(v)$;
     - $size(v)$: sum of the weights of v's descendants in T_r (or zero if there's a landmark among the descendants).
  4. let w be the vertex maximizing $size(w)$;
  5. starting at w, go down T_r following the maximum-sized child;
  6. pick the leaf at the end of this path as the new landmark.

ALT: Landmark Generation

- ALT algorithm on Europe with travel times, 16 landmarks:

METHOD	PREP.	SCANS
random	6	343440
avoid	12	84740

PREP.: preprocessing time in minutes.

SCANS: average number of scans for random queries.

ALT: Maxcover Landmarks

- To pick 16 landmarks, generate 64 and pick the "best" 16:
  - look for subset minimizing query times.
- Measure quality of subset S based on individual arcs.
  - Landmark L covers arc (v, w) if v is on shortest path from L to w:
    - Formally: $\ell(v, w) = \text{dist}(L, w) - \text{dist}(L, v)$.
    - Intuition: L gives lower bounds on paths containing (v, w).
  - If one landmark in S covers (v, w), then S covers (v, w).
- The **maxcover** landmark selection algorithm:
  - Pick the subset that covers the most arcs.
  - NP-hard, but local search works well.

ALT: Landmark Generation

- ALT algorithm on Europe with travel times, 16 landmarks:

METHOD	PREP.	SCANS
random	6	343440
avoid	12	84740
maxcover	79	71508

Results

- USA, travel times, random pairs [GKW09]:

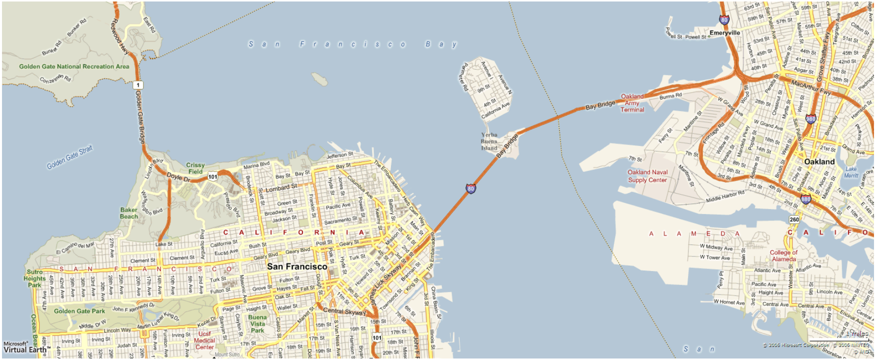
	PREPROCESSING		QUERY	
METHOD	minutes	MB	scans	ms
Dijkstra	—	536	11 808 864	5440.49
ALT(16)	18	2563	187 968	295.44

- 1% of nodes visited on average, 10% in bad cases.
- Can we do better?

Reach

Treasure Island

- Intuition: don't visit "local" roads when far from both s and t.
  - Search from San Francisco to Oakland should not visit Treasure Island.



Reach

- Let v be a vertex on the shortest path P between s and t:

- Reach of v with respect to P:

$$r(v, P) = \min\{\text{dist}(s, v), \text{dist}(v, t)\}$$

- Reach of v with respect to the whole graph:

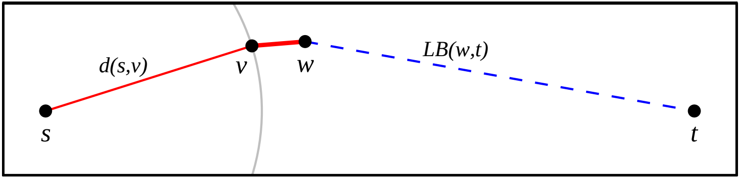
$$r(v) = \max_P r(v, P),$$

  over all **shortest** paths P that contain v [Gut04].

- Intuition:
  - a high-reach vertex is close to the middle of some long shortest path;
  - vertices on highways have high reach;
  - local intersections have low reach.

Reach Queries

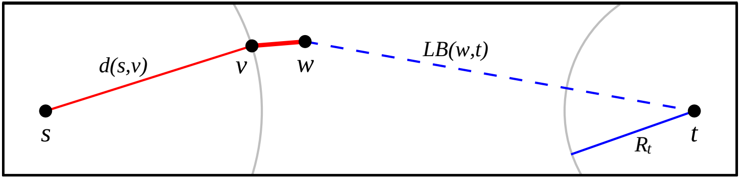
- Reaches can prune the search during an s-t query.
  - Intuition: don't visit small-reach vertices when far from s **and** t.
- When processing edge (v, w):
  - prune w if $r(w) < \min\{d(s, v) + \ell(v, w), LB(w, t)\}$:



  - If $P = (s, \ldots, v, w, \ldots t)$ were a shortest path, $r(w)$ would be higher.
- How can we find the lower bound $LB(w, t)$?
  - Explicitly: Euclidean distances [Gut04], landmarks.
  - Implicitly: make the search bidirectional.

Reach Queries

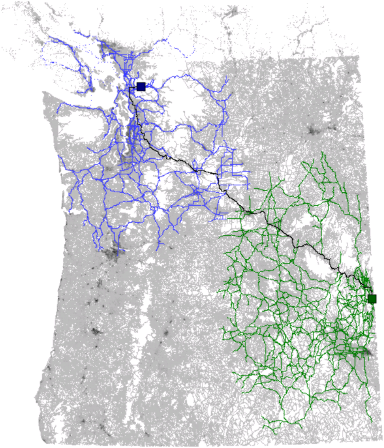
- Radius R_t of the opposite search is lower bound:
  - if w not visited by reverse direction, $d(w,t) \geq R_t$.
- When processing edge (v,w):
  - prune w if $r(w) < \min\{d(s,v) + \ell(v,w), R_t\}$:



- For best results, balance forward and reverse searches.

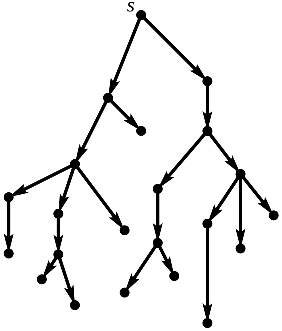
Northwestern USA

- Reach:



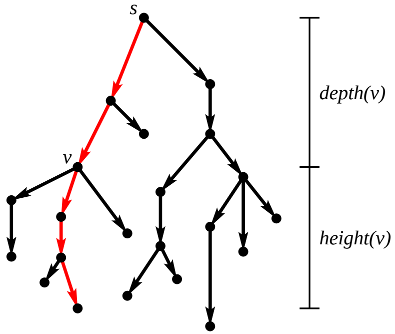
Preprocessing: Computing Reaches

1. Initialization: set $r(v) \leftarrow 0$, for all v;
2. For every vertex $s \in V$:
   - for each vertex v in shortest path tree T_s:
     - take longest shortest path P_{st} containing v.
     - $d_s(v)$ (**depth**): distance from s;
     - $h_s(v)$ (**height**): distance to farthest descendant;
     - $r_s(v)$ (**reach** within T_s) $= \min\{d_s(v), h_s(v)\}$.
     - set $r(v) \leftarrow \max\{r(v), r_s(v)\}$.



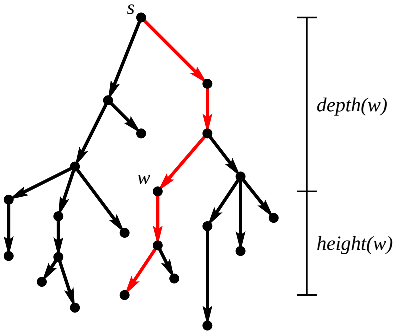
Preprocessing: Computing Reaches

1. Initialization: set $r(v) \leftarrow 0$, for all v;
2. For every vertex $s \in V$:
   - for each vertex v in shortest path tree T_s:
     - take longest shortest path P_{st} containing v.
     - $d_s(v)$ (**depth**): distance from s;
     - $h_s(v)$ (**height**): distance to farthest descendant;
     - $r_s(v)$ (**reach** within T_s) $= \min\{d_s(v), h_s(v)\}$.
     - set $r(v) \leftarrow \max\{r(v), r_s(v)\}$.



Preprocessing: Computing Reaches

1. Initialization: set $r(v) \leftarrow 0$, for all v;
2. For every vertex $s \in V$:
   - for each vertex v in shortest path tree T_s:
     - take longest shortest path P_{st} containing v.
     - $d_s(v)$ (**depth**): distance from s;
     - $h_s(v)$ (**height**): distance to farthest descendant;
     - $r_s(v)$ (**reach** within T_s) $= \min\{d_s(v), h_s(v)\}$.
     - set $r(v) \leftarrow \max\{r(v), r_s(v)\}$.



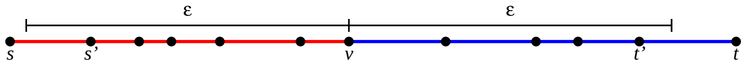
Preprocessing: Computing Reaches

1. Initialization: set $r(v) \leftarrow 0$, for all v;
2. For every vertex $s \in V$:
   - for each vertex v in shortest path tree T_s:
     - take longest shortest path P_{st} containing v.
     - $d_s(v)$ (**depth**): distance from s;
     - $h_s(v)$ (**height**): distance to farthest descendant;
     - $r_s(v)$ (**reach** within T_s) $= \min\{d_s(v), h_s(v)\}$.
     - set $r(v) \leftarrow \max\{r(v), r_s(v)\}$.

- Running time $= n \times$ *Dijkstra*
  - Too slow: 12 hours on Bay Area ($n = 330K$), years on USA.
- We need something faster!

Preprocessing: Computing Small Reaches

- Suppose we only need to find vertices with **small reach** ($< \epsilon$).
- Fact: $r(v) \geq \epsilon \Rightarrow$ there is a path $P \ni v$ with $r(v, P) \geq \epsilon$.
- A shortest path $P_{st} = (s, s', \ldots, v, \ldots, t', t)$ is ϵ-minimal w.r.t. v if
  - $\text{dist}(s, v) \geq \epsilon$ and $\text{dist}(s', v) < \epsilon$;
  - $\text{dist}(v, t) \geq \epsilon$ and $\text{dist}(v, t') < \epsilon$.



Preprocessing: Computing Small Reaches

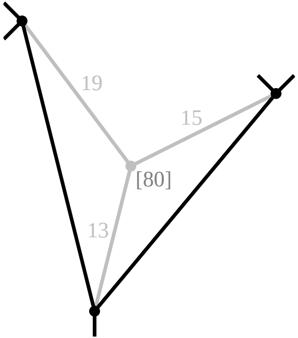
- Fact: $r(v) \geq \epsilon \Rightarrow$ there is an ϵ-**minimal** path $P \ni v$ with $r(v, P) \geq \epsilon$.
- Algorithm:
  - compute reach bounds $r'(\cdot)$ using only (all) ϵ-minimal paths;
  - if $r'(v) < \epsilon$, the bound is correct $(r(v) = r'(v))$;
  - if $r'(v) \geq \epsilon$, we can say nothing $(r(v) \geq r'(v))$.
- It suffices to consider **partial trees**:
  - shortest path trees grown to depth about 2ϵ.

Preprocessing: Bounding Reaches

- Algorithm:
  1. Set $G' \leftarrow G$ and $\epsilon \leftarrow \epsilon_0$ (some small value).
  2. while G' is not empty:
     - ★ use partial trees to find vertices with reach $\geq \epsilon$;
     - ★ remove from G' the remaining vertices (their reach is $< \epsilon$);
     - ★ set $\epsilon \leftarrow 3\epsilon$.

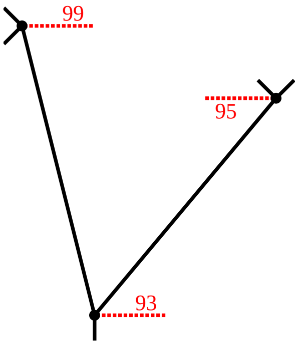
Penalties

- Problem: must consider shortest paths starting at discarded vertices.
- Solution is to add **penalties**:
  - upper bound on the length of longest path into "discarded" area.



Penalties

- Problem: must consider shortest paths starting at discarded vertices.
- Solution is to add **penalties**:
  - upper bound on the length of longest path into "discarded" area.
- When growing partial trees:
  - "extend" all paths s–t using penalties at s and t.
- Reaches are no longer exact!
  - valid **upper bounds** $\bar{r}(\cdot)$;
  - query algorithm still correct;
  - query performance slightly worse.

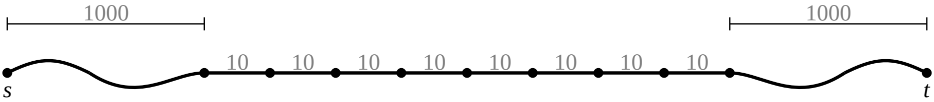


Preprocessing: Bounding Reaches

- Algorithm:
  1. Set $G' \leftarrow G$ and $\epsilon \leftarrow \epsilon_0$ (some small value).
  2. while G' is not empty:
     - ★ use partial trees to find vertices with reach $\geq \epsilon$;
     - ★ remove from G' the remaining vertices (their reach is $< \epsilon$);
     - ★ set $\epsilon \leftarrow 3\epsilon$.
- Preprocessing time:
  - trees get deeper as ϵ increases;
  - G' gets smaller: fewer trees than G, each with fewer vertices.
- This helps somewhat, but not much:
  - Small graph ($n = 330K$): 12h exact, 1h approximate.
  - Still too slow for large graphs (weeks).

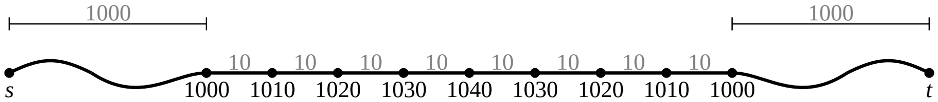
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:



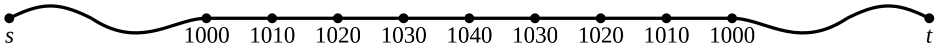
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.



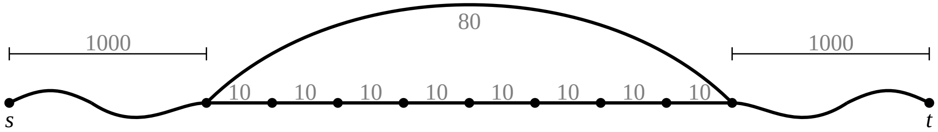
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut [SS05, SS06]:
  - single edge bypassing a path (with same length).



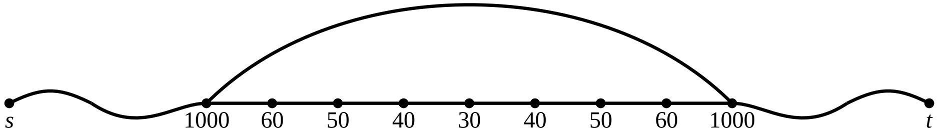
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut [SS05, SS06]:
  - single edge bypassing a path (with same length).
- Assume ties are broken by taking path with fewer nodes:



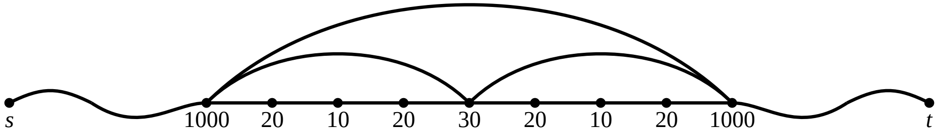
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut [SS05, SS06]:
  - single edge bypassing a path (with same length).
- Assume ties are broken by taking path with fewer nodes:
  - some reaches are reduced!



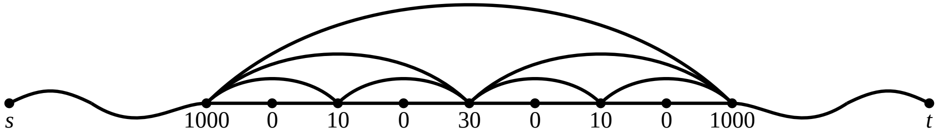
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut [SS05, SS06]:
  - single edge bypassing a path (with same length).
- Assume ties are broken by taking path with fewer nodes:
  - some reaches are reduced!
- More shortcuts can be added recursively.



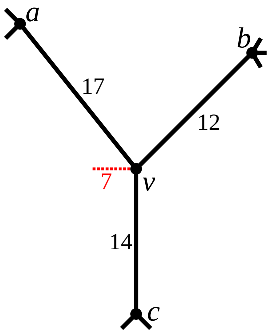
Reach with Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut [SS05, SS06]:
  - single edge bypassing a path (with same length).
- Assume ties are broken by taking path with fewer nodes:
  - some reaches are reduced!
- More shortcuts can be added recursively.



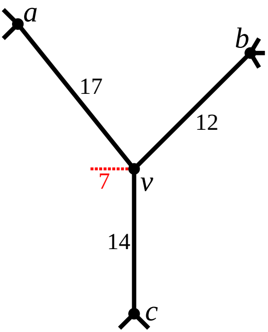
Reach with Shortcuts

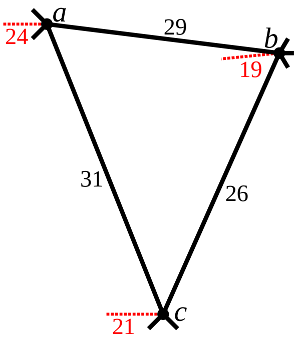
- Shorcuts are actually added heuristically during preprocessing:
  - in each iteration, bypass some low-degree (≤ 4) vertices:



Reach with Shortcuts

- Shorcuts are actually added heuristically during preprocessing:
  - in each iteration, bypass some low-degree (≤ 4) vertices:
    - ★ neighbors connected directly by shortcuts;
    - ★ penalties added to the neighbors;
    - ★ bound the reach of eliminated vertex (below: $\bar{r}(v) = 7$);





Reach with Shortcuts

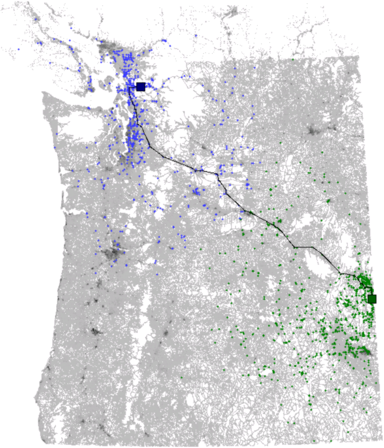
- Benefits of shortcuts:
  - speeds up preprocessing (graph G' shrinks faster);
  - speeds up queries (pruning more effective);
  - requires slighly more space (graph has $\sim 50\%$ more arcs).

Preprocessing: Bounding Reaches

- Revised algorithm:
  1. Set $G' \leftarrow G$ and $\epsilon \leftarrow \epsilon_0$ (some small value).
  2. while G' is not empty:
     - **add shortcuts**;
     - use partial trees to find vertices with reach $\geq \epsilon$;
     - remove from G' the remaining vertices (their reach is $< \epsilon$);
     - set $\epsilon \leftarrow 3\epsilon$.

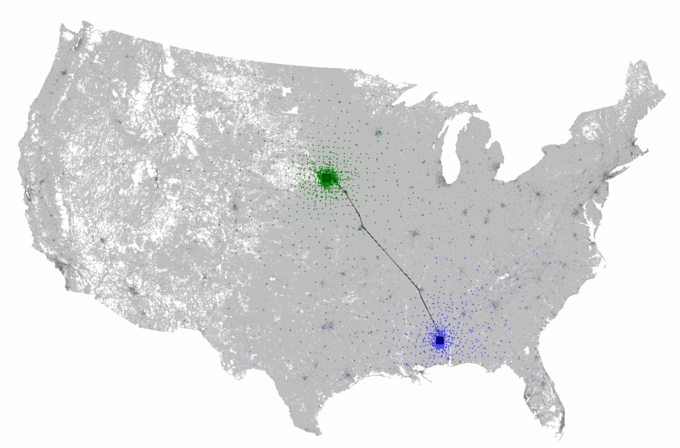
Northwestern USA

- Reach with shortcuts (RE):



Reach with Shortcuts

- Reach with shortcuts (RE):



Engineering Reach

- Preprocessing:
  - keep in-penalties and out-penalties;
  - compute **arc** reaches instead of vertex reaches;
  - recompute the top reaches explicitly;
  - careful stopping criterion when growing partial trees.
  - relax criteria for shortcutting in later rounds.
- Queries:
  - rearrange vertices to improve locality;
  - high-reach vertices together in memory;
  - implicitly skip low-reach arcs out of each vertex.

Results

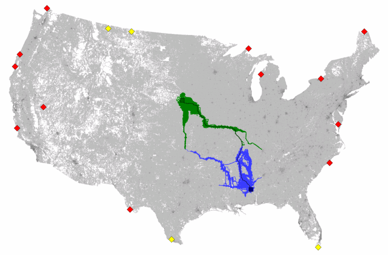
- USA, travel times, random pairs [BDS+10]:

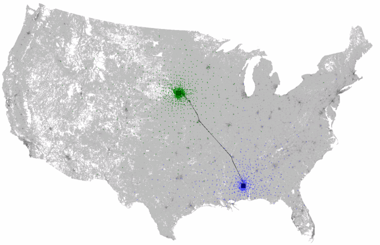
	PREPROCESSING		QUERY	
METHOD	minutes	MB	scans	ms
Dijkstra	—	536	11 808 864	5440.49
ALT(16)	18	2563	187 968	295.44
RE	28	893	2 405	1.77

Reach for A*

REAL

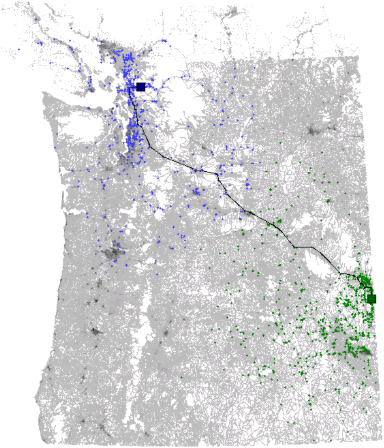
- A* search with landmarks can use reaches:
  - A* gives direction to the search;
  - reaches make the search space sparser.
- Landmarks have dual purpose:
  - guide the search;
  - provide lower bound for reach pruning.





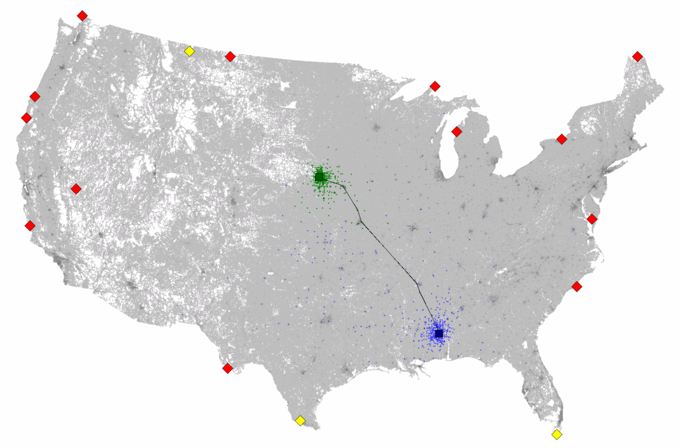
Northwestern USA

- Reach + ALT (REAL):



REAL

- REAL: Reach + ALT.



Results

- USA, travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	MB	scans	ms
Dijkstra	—	536	11 808 864	5440.49
ALT(16)	18	2563	187 968	295.44
RE	28	893	2 405	1.77
REAL(16)	45	3032	592	0.80

Partial Landmarks

- Fact: search visits low-reach vertices only in the beginning.
- Can lower memory usage as follows:
  - store landmark distances only for vertices with reach $\geq R$.
  - start search without A* (only reaches);
  - use A* (with reaches) when radii are greater than R.
- Example: 64 landmarks, with distances to only $n/16$ top vertices:
  - same memory as four "full" landmarks.

Results

- USA, travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	MB	scans	ms
Dijkstra	—	536	11 808 864	5440.49
ALT(16)	18	2563	187 968	295.44
RE	28	893	2 405	1.77
REAL(16)	45	3032	592	0.80
REAL(64,16)	114	1579	538	0.86

Other Graphs

- Partial A^* (REAL(64,16)), 1000 random pairs:

	PREPROCESS		QUERY	
METHOD	minutes	MB	avgscan	ms
USA (times)	113	1579	538	0.86
USA (distances)	120	1575	670	1.22
Europe (times)	102	1037	610	0.91
Europe (distances)	76	1084	562	0.91

(Europe has 18.0 M vertices and 42.6 M arcs.)

- Grids: A^* is just as good, reaches not so much (no hierarchy).
- Random graphs: both methods are useless.

Results

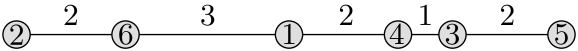
- Europe (18M vertices), travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91

Contraction Hierarchies (CH)

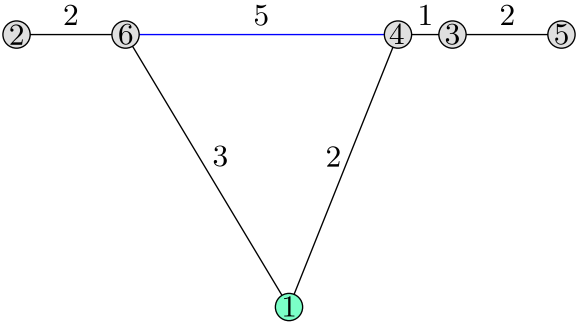
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;



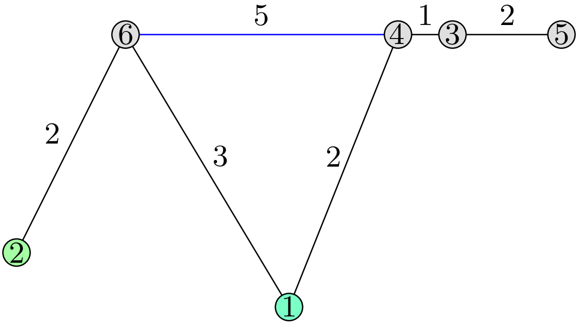
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;
  2. add shortcuts to preserve distances.



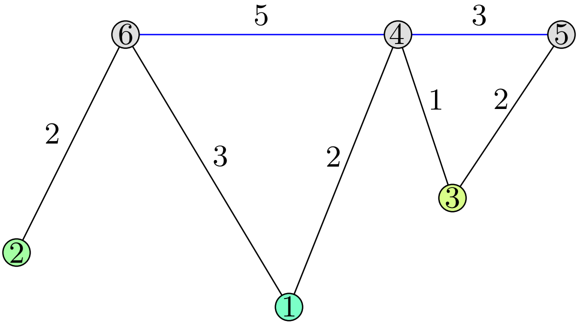
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;
  2. add shortcuts to preserve distances.



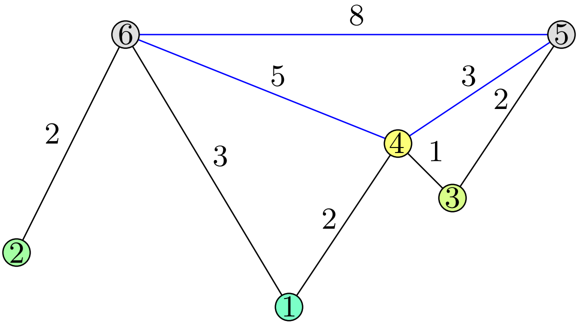
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;
  2. add shortcuts to preserve distances.
- Invariant: distances between remaining vertices are preserved;
  - last k vertices induce an **overlay graph**, for every k.



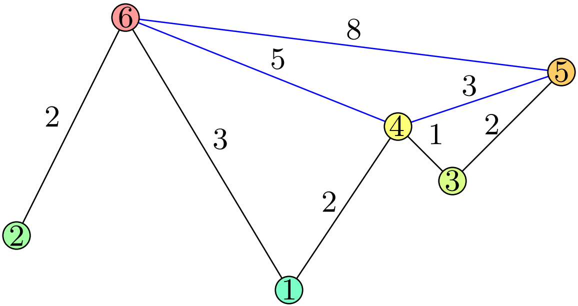
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;
  2. add shortcuts to preserve distances.
- Invariant: distances between remaining vertices are preserved;
  - last k vertices induce an **overlay graph**, for every k.



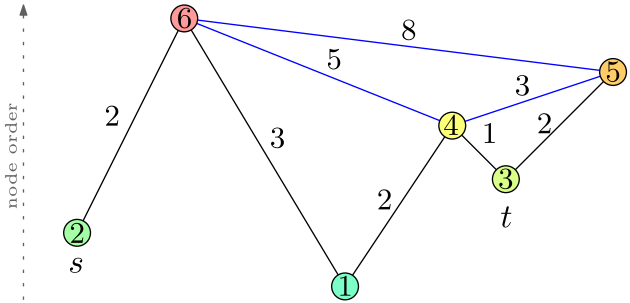
Contraction Hierarchies: Preprocessing

- CH Preprocessing [GSSD08]:
  1. eliminate vertices one by one, in some order;
  2. add shortcuts to preserve distances.
- Invariant: distances between remaining vertices are preserved;
  - last k vertices induce an **overlay graph**, for every k.
- Output: augmented graph + node order.



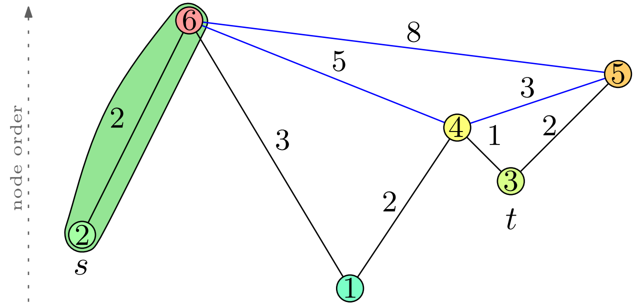
Contraction Hierarchies: Query

- Bidirectional search following only **upward** arcs.



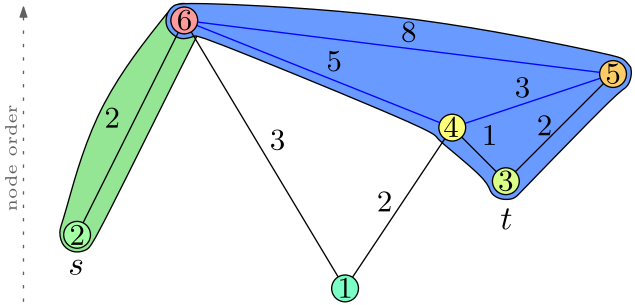
Contraction Hierarchies: Query

- Bidirectional search following only **upward** arcs.



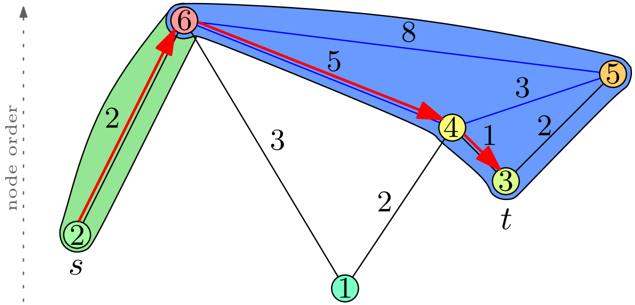
Contraction Hierarchies: Query

- Bidirectional search following only **upward** arcs.



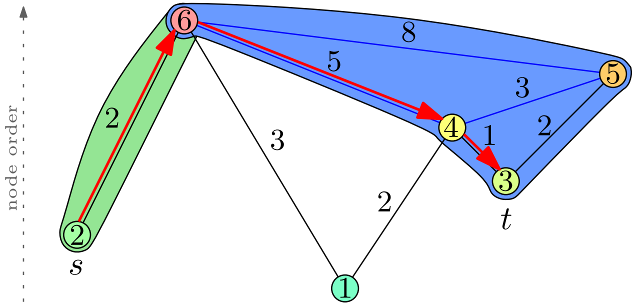
Contraction Hierarchies: Query

- Bidirectional search following only **upward** arcs.
- In general, several nodes will be visited by both searches:
  - Shortest path uses the node minimizing the sum of its distance labels.



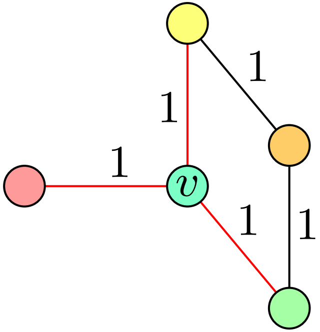
Contraction Hierarchies: Correctness

- Distances are preserved by the augmented graph $G^+ = (V, E^+)$.
- Claim: for every $\{s, t\}$, G^+ has a shortest path $P_{st} \ni v$ such that:
  - the subpath P_{sv} is increasing;
  - the subpath P_{vt} is decreasing.
- In other words, P_{st} has **no interior local minima**.
- Forward search will find P_{sv}, backward search will find P_{vt}.

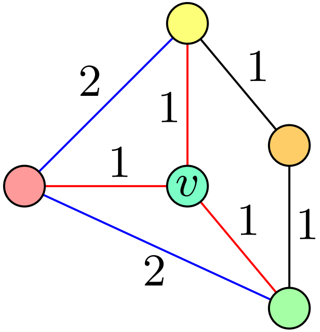


Contraction Hierarchies: Witness Search

- When bypassing v, for every two neighbors u and w:
  - Add edge (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$...

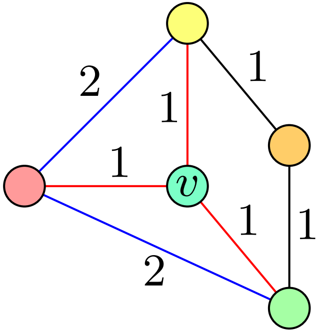


- When bypassing v, for every two neighbors u and w:
  - Add edge (u, w) with $\ell(u,w) = \ell(u,v) + \ell(v,w)$...
  - ...but only if there is no **witness path** P_{uw}:
    - length at most $\ell(u,v) + \ell(v,w)$;
    - does not contain v.



Contraction Hierarchies: Witness Search

- When bypassing v, for every two neighbors u and w:
  - Add edge (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$...
  - ...but only if there is no **witness path** P_{uw}:
    - length at most $\ell(u, v) + \ell(v, w)$;
    - does not contain v.
- Must perform **witness searches** to find such paths:
  - Dijkstra between neighbors;
  - essential for good query performance;
  - avoids explosion in the size of the graph;
  - somewhat expensive:
    - run when shortcutting;
    - run when computing priorities.

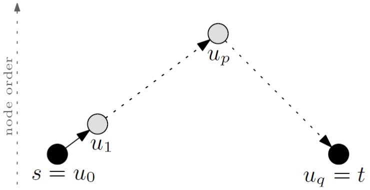


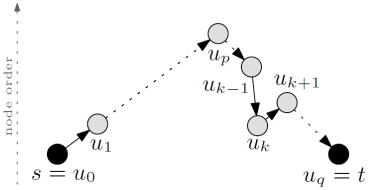
Contraction Hierarchies: Correctness

Theorem

The augmented graph has a shortest s–t path with no interior local minima.

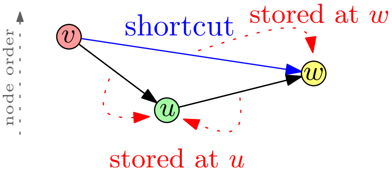
- Take a SP P with local minima $M_P = \{u_i : u_{i-1} > u_i < u_{i+1}\}$; let $u_k = \min M_P$;
- when u_k was contracted, (u_{k-1}, u_k) and (u_k, u_{k+1}) were in the overlay graph;
- either the edge $e = (u_{k-1}, u_{k+1})$ was added, or there was a witness path;
- the subpath (u_{k-1}, u_k, u_{k+1}) can be replaced:
  - either M_P becomes empty or $\min M_P$ increases.
- $\min M_P$ can increase at most n times $\Rightarrow$ M_P is eventually empty.





Contraction Hierarchies: Representation

- Each arc (a, b) stored only at $\min\{a, b\}$:
  - used by forward search (from a) if $a < b$;
  - used by backward search (from b) if $b < a$.
- Can save memory, particularly on undirected edges.
  - stored twice for Dijkstra, once for CH.



Contraction Hierarchies: Elimination Order

- Elimination order actually determined online.
  - each candidate vertex has a **priority**;
  - keep all vertices in a heap;
  - neighbors updated after contraction.
- Priority may depend on several terms:
  - edge difference (removed − inserted).
  - uniformity (nodes should be spread around the graph);
  - cost of contraction (search spaces during witness search);
  - ...
- Any order is correct, but performance varies:
  - space (number of shortcuts);
  - preprocessing time;
  - query times.

Contraction Hierarchies: Elimination Order

- Europe (18M vertices), travel times, random pairs [GSSD08]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
CH(E)	245	−1.6	1791	0.67
CH(ES)	91	**−3.5**	614	0.24
CH(EDS1235)	**10**	0.6	459	0.22
CH(EVSQWL)	32	−3.0	**359**	**0.15**

- Key:
  - E: edge difference
  - S: size of search space (cost of contraction)
  - D: deleted neighbors (uniformity)
  - 1235: hop limits on witness search
  - EVSQWL: six different heuristic measures (see [GSSD08]).

Results

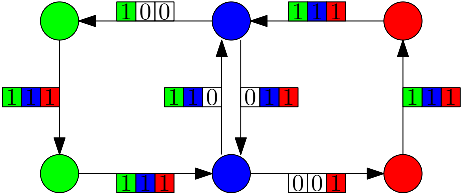
- Europe (18M vertices), travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18

Arc Flags

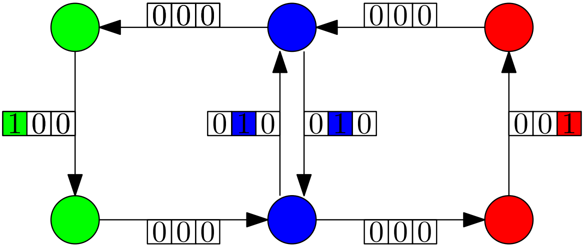
Arc Flags

- Originally Lauther [Lau04], later improved [MSS+06, HKMS09].
- Preprocessing:
  - partition graph into k regions;
  - store a k-bit **flag** with each arc (v, w):
    - bit i indicates if there is a shortest path from v to region i using (v, w).
- Query from s to t:
  - set $R \leftarrow \text{region}(t)$;
  - run modified Dijkstra's algorithm from s:
    - when scanning v, skip arc (v, w) if ths R-th bit of its flag is 0 (arc (v, w) is not on any shortest path to region r).



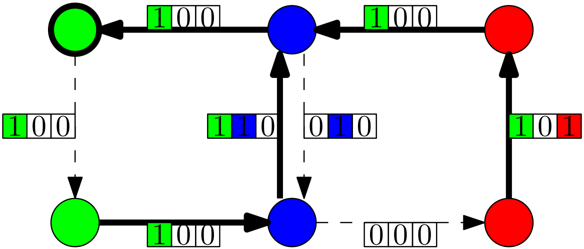
Preprocessing: Arc Flags

- Basic preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.



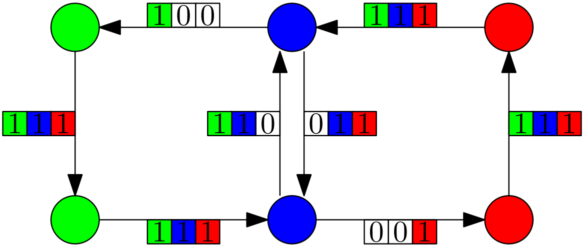
Preprocessing: Arc Flags

- Basic preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



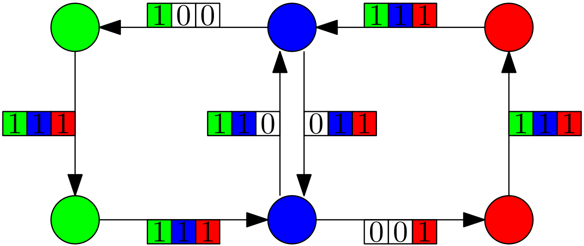
Preprocessing: Arc Flags

- Basic preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



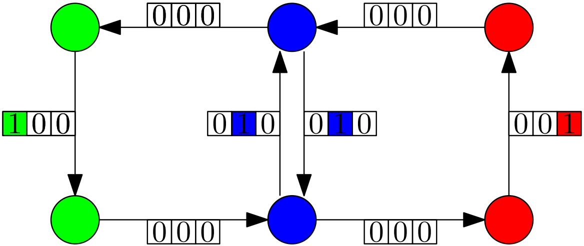
Preprocessing: Arc Flags

- Basic preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.
- Cost: $n \times$ Dijkstra.



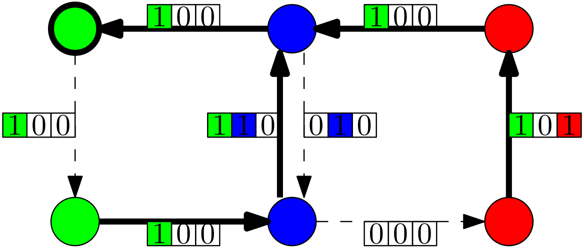
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



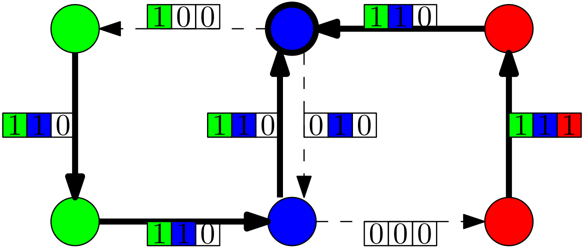
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



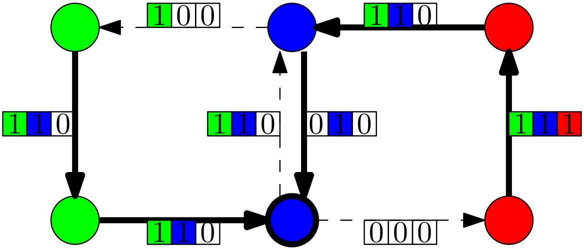
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



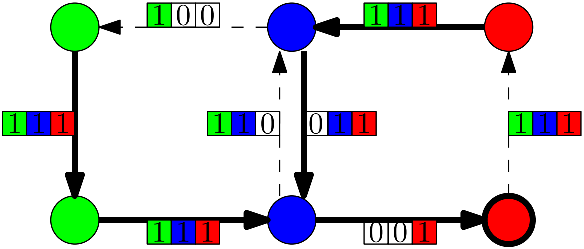
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



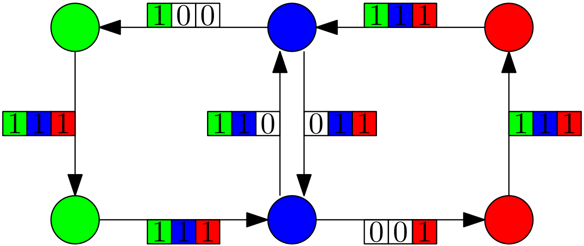
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



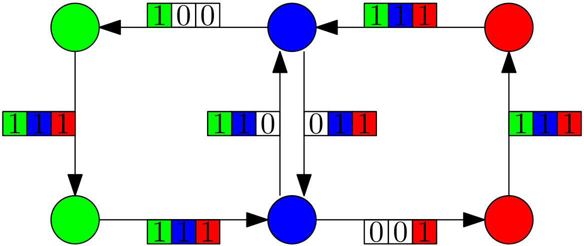
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.



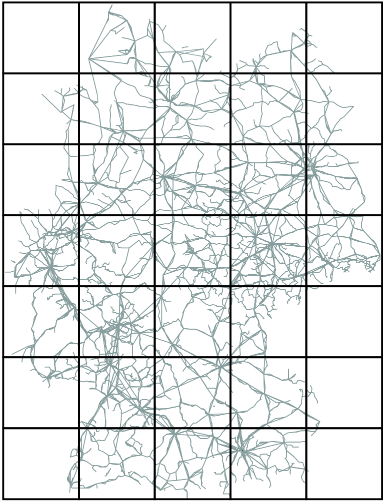
Preprocessing: Arc Flags

- **Faster** preprocessing algorithm:
  1. Initialize all flags to 0.
  2. Set self-region flags to 1.
  3. For each node v **with an incoming boundary arc**:
     1. let $R \leftarrow \text{region}(v)$.
     2. grow an **incoming** shortest path tree;
     3. for each **tree** arc, set R-th bit to 1.
- Cost: (number of boundary nodes)$\times$Dijkstra.
  - can be accelerated in practice.



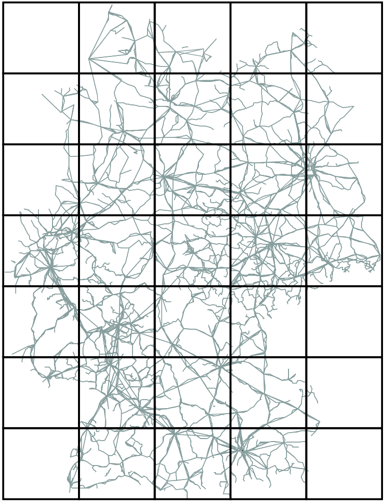
Arc Flags: Partitioning

- Must partition the graph into regions.
- Original suggestion [Lau04]: simple grid.



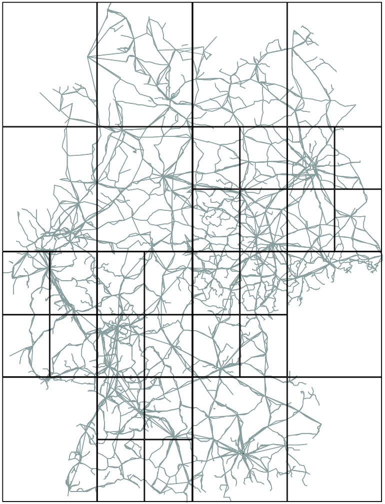
Arc Flags: Partitioning

- Must partition the graph into regions.
- Original suggestion [Lau04]: simple grid.
- Better performance if regions:
  - are roughly balanced in size;
  - are connected;
  - have few boundary arcs.



Arc Flags: Partitioning

- Must partition the graph into regions.
- Original suggestion [Lau04]: simple grid.
- Better performance if regions:
  - are roughly balanced in size;
  - are connected;
  - have few boundary arcs.
- Several methods tried [MSS$^+$06]:
  - quad-trees;



Arc Flags: Partitioning

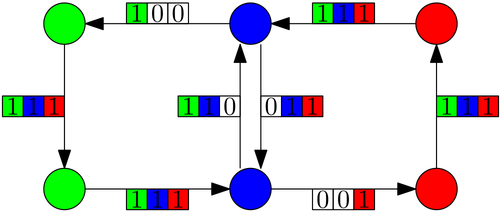
- Must partition the graph into regions.
- Original suggestion [Lau04]: simple grid.
- Better performance if regions:
  - are roughly balanced in size;
  - are connected;
  - have few boundary arcs.
- Several methods tried [MSS$^+$06]:
  - quad-trees;
  - kd-trees;

Arc Flags: Partitioning

- Must partition the graph into regions.
- Original suggestion [Lau04]: simple grid.
- Better performance if regions:
  - are roughly balanced in size;
  - are connected;
  - have few boundary arcs.
- Several methods tried [MSS+06]:
  - quad-trees;
  - kd-trees;
  - multiway cut.

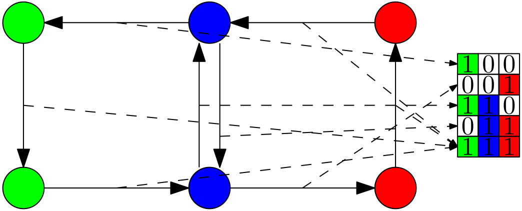
Arc Flags: Compression

- Fact: several arcs have the exact same k-bit flags.



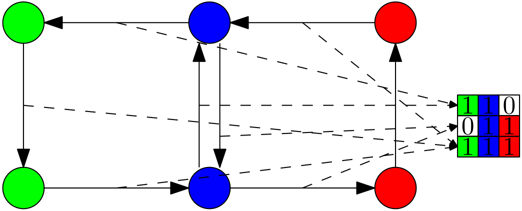
Arc Flags: Compression

- Fact: several arcs have the exact same *k*-bit flags.
- Trick: keep a table with all **different** flags seen:
  - each arc keeps index of a table entry;
  - typical savings: 80%.



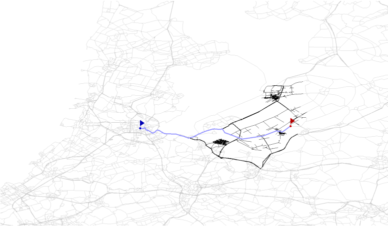
Arc Flags: Compression

- Fact: several arcs have the exact same *k*-bit flags.
- Trick: keep a table with all **different** flags seen:
  - each arc keeps index of a table entry;
  - typical savings: 80%.
- Can even merge some flags to save more space:
  - still correct if some 0s become 1s.
  - performance may suffer.

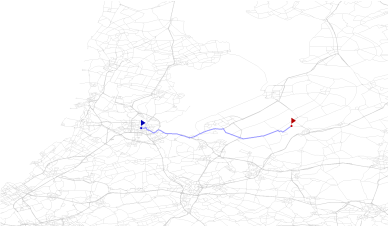


Arc Flags: Coning

- The "problem of cones":
  - flags fail close to the destination;
  - no pruning within target region.

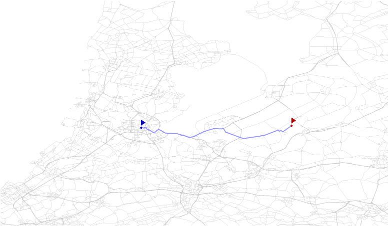


Arc Flags: Coning



- The "problem of cones":
  - flags fail close to the destination;
  - no pruning within target region.
- **Bidirectional** arc flags are better:
  - keep forward and reverse flags;
  - double space and preprocessing time;
  - searches meet far from source and destination!
  - Europe, 128 regions [Del10]:
    - unidirectional: 92 545 scans;
    - bidirectional: 2 764 scans.

Arc Flags: Coning



- The "problem of cones":
  - flags fail close to the destination;
  - no pruning within target region.
- **Bidirectional** arc flags are better:
  - keep forward and reverse flags;
  - double space and preprocessing time;
  - searches meet far from source and destination!
  - Europe, 128 regions [Del10]:
    - ⋆ unidirectional: 92 545 scans;
    - ⋆ bidirectional: 2 764 scans.
- Also: use more regions.

Arc Flags: Number of Regions

- Europe (18M vertices), travel times, random pairs [Hil07]:

	PREPROCESSING		QUERY	
REGIONS	minutes	B/node	scans	ms
0	0	0	9 114 385	5591.6
200	1028	19	2369	1.6
600	1723	21	1700	1.1
1000	2156	25	1593	1.1

- More regions:
  - fewer vertices scanned, but...
  - ...more space required.

Arc Flags: Results

- Europe (18M vertices), travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18
Arc Flags	2156	25	1 593	1.10

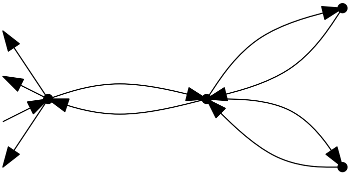
SHARC

SHARC

- Issues with arc flags:
  - very expensive preprocessing;
  - (unidirectional) queries could be faster.
- Possible improvements [BD09]:
  - contraction (shortcuts);
  - multilevel flags.

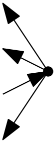
SHARC: Tree Elimination

1. Before partition, remove **attached trees**:
   - repeatedly remove vertices of degree 1 (about 1/3 of Europe);
   - the remaining 2/3 are the **core**.



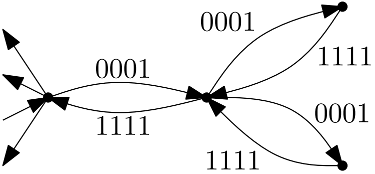
SHARC: Tree Elimination

1. Before partition, remove **attached trees**:
   - repeatedly remove vertices of degree 1 (about 1/3 of Europe);
   - the remaining 2/3 are the **core**.
2. Compute partition and arc flags of the core.



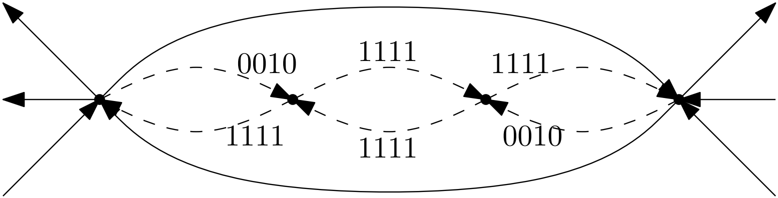
SHARC: Tree Elimination

1. Before partition, remove **attached trees**:
   - repeatedly remove vertices of degree 1 (about 1/3 of Europe);
   - the remaining 2/3 are the **core**.
2. Compute partition and arc flags of the core.
3. Set flags on attached trees:
   - arcs towards the core: all flags set to 1;
   - arcs away from the core: only R-th bit set to 1.
     - R: region containing the root of the attached tree.



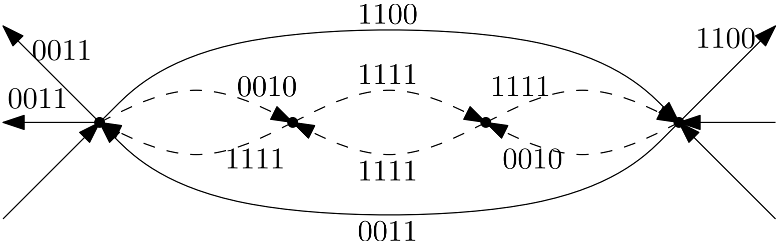
SHARC: Contraction

- Idea can be generalized for arbitrary contractions:
  1. add shortcuts to bypass a contracted component;
     - ★ arcs entering the component have only own-region bit set;
     - ★ all other arcs are set to 1.



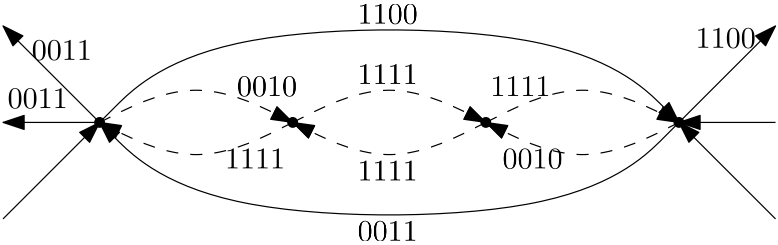
SHARC: Contraction

- Idea can be generalized for arbitrary contractions:
  1. add shortcuts to bypass a contracted component;
     - ★ arcs entering the component have only own-region bit set;
     - ★ all other arcs are set to 1.
  2. compute arc flags of the core.



SHARC: Contraction

- Idea can be generalized for arbitrary contractions:
  1. add shortcuts to bypass a contracted component;
     - ★ arcs entering the component have only own-region bit set;
     - ★ all other arcs are set to 1.
  2. compute arc flags of the core.
  3. further refinement of component flags is possible.

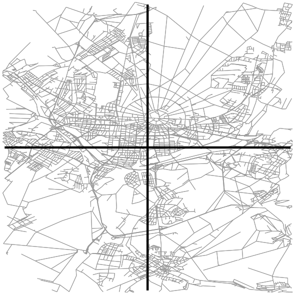


SHARC: Multilevel Flags

- Compute **multilevel partition**:

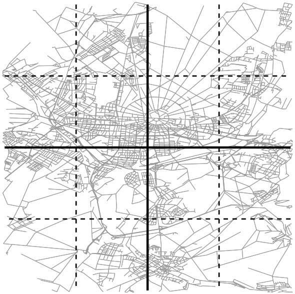
SHARC: Multilevel Flags

- Compute **multilevel partition**:
  - cells (region) at level i subdivide cells at $i + 1$.



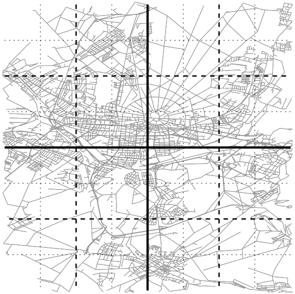
SHARC: Multilevel Flags

- Compute **multilevel partition**:
  - cells (region) at level i subdivide cells at $i+1$.



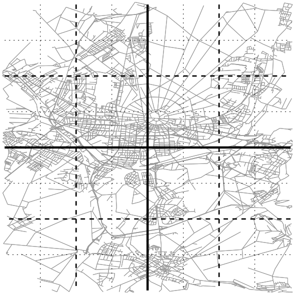
SHARC: Multilevel Flags

- Compute **multilevel partition**:
  - cells (region) at level i subdivide cells at $i + 1$.



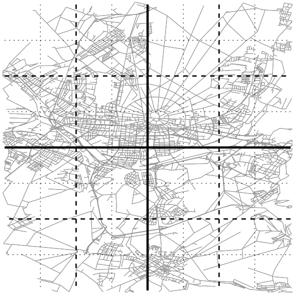
SHARC: Multilevel Flags

- Compute **multilevel partition**:
  - cells (region) at level i subdivide cells at $i+1$.
- When query looks at (v, w):
  - use flags for highest level i with $\text{cell}_i(v) \neq \text{cell}_i(t)$;



SHARC: Multilevel Flags

- Compute **multilevel partition**:
  - cells (region) at level i subdivide cells at $i+1$.
- When query looks at (v, w):
  - use flags for highest level i with $\text{cell}_i(v) \neq \text{cell}_i(t)$;
- Advantages:
  - performance: bottom-level cells are tiny.
  - saves space: level-i flags defined only for cells with same parent.



SHARC

- SHARC [BD09]: SHortcuts + ARC flags.
- Preprocessing:
  1. Build multilevel partition.
  2. For each level $0 \dots L$:
     - perform contractions within each cell;
     - compute appropriate arc flags.
- Query: similar to arc flags, on graph with shortcuts.
- Unidirectional!

SHARC: Tuning

- Unidirectional SHARC queries:

PARTITION									PREPRO		QUERY	
#cells per level								#total cells	time [h:m]	space [B/n]	#settled nodes	time [μs]
-	-	-	-	-	-	-	128	128	1:52	6.0	78 429	23 306
-	-	-	-	-	-	8	120	960	1:14	9.8	11 362	3 049
-	-	-	-	-	4	8	116	3 712	1:25	10.8	3 459	983
-	-	-	-	4	4	8	112	14 336	1:14	11.6	1 320	441
-	-	-	4	4	8	8	104	106 496	1:18	13.7	700	294
-	-	**4**	**4**	**4**	**4**	**8**	**104**	**212 992**	**1:21**	**14.5**	**654**	**290**
-	4	4	4	4	8	8	96	1 572 864	1:46	19.3	637	294
4	4	4	4	4	4	8	96	3 145 728	1:56	20.0	627	293

- Can be made bidirectional:
  - best performace with two levels;
  - double preprocessing time;
  - queries about 5 times faster.

Results

- Europe (18M vertices), travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18
Arc Flags	2156	25	1 593	1.10
SHARC (uni)	81	15	654	0.29
SHARC (bi)	212	21	125	0.065

CHASE

CHASE

- CHASE [BDS$^+$08, BDS$^+$10]: Contraction Hierarchies + Arc Flags.
- Preprocessing:
  1. Run CH preprocessing to create G'.
  2. Partition G' into k regions.
  3. Compute arc flags for all arcs in G'.
- Query is bidirectional Dijkstra, double-pruned:
  - CH: only follow arcs going "up" in the hierarchy;
  - Arc flags: only follow arcs on a shortest path to the target region.
- Practical improvement:
  - compute flags only for arcs between the top 5% vertices;
  - much faster preprocessing, query times barely affected.

Results

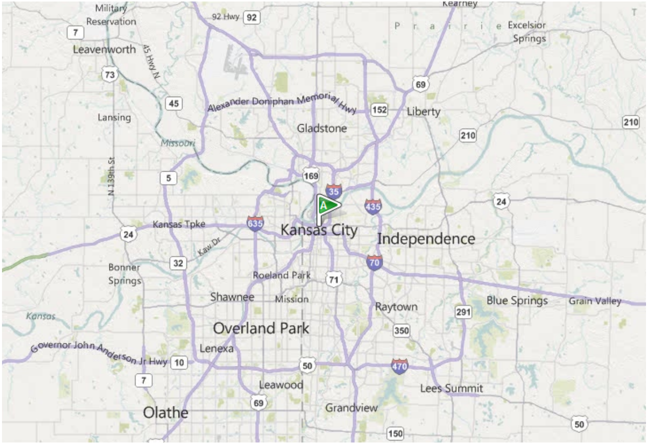
- Europe (18M vertices), travel times, random pairs [BDS+10]:

METHOD	PREPROCESSING minutes	B/node	QUERY scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18
Arc Flags	2156	25	1 593	1.10
SHARC (uni)	81	15	654	0.29
SHARC (bi)	212	21	125	0.065
CHASE	99	12	45	0.017

Transit-Node Routing (TNR)

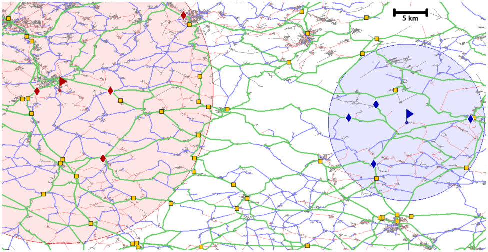
Transit Nodes

- Intuition: when driving "far away" from a small region (on a shortest path), you must pass through one of very few **access** roads.



Transit Nodes

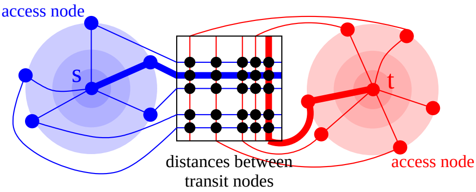
- USA has small ($\sim 10\,000$) set $\mathcal{T}$ of **transit nodes** s.t. [BFM+07]:
  - the shortest path between any two "far away" nodes passes through $\mathcal{T}$:
    - path has at least one node (vertex) from $\mathcal{T}$;
  - there are very few ways of driving "far away" from any vertex s:
    - v is an **access node** for s if it's the first node in $\mathcal{T}$ on a long shortest path from s;
    - on average, any vertex s has about 10 access nodes.



[BFM+07]

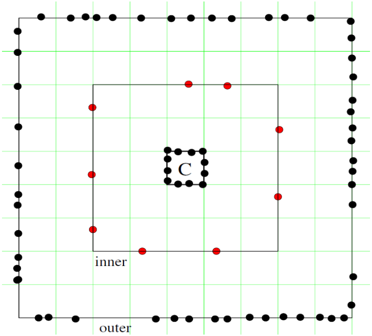
Transit Node Routing

- Preprocessing:
  - find set of **transit nodes** $\mathcal{T} \subset V$;
  - store full $|\mathcal{T}| \times |\mathcal{T}|$ **distance table**;
  - for every node v, store distances v to $\overrightarrow{A}(v)$ and from $\overleftarrow{A}(t)$:
    - ★ forward and backward access nodes.
- Query from s to t:
  - if s and t are "sufficiently far," do $|\overrightarrow{A}(s)| \times |\overleftarrow{A}(t)|$ table lookups;
    - ★ $\text{dist}(s,t) = \min\{\text{dist}(s,u) + \text{dist}(u,v) + \text{dist}(v,t) : u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)\}$
  - otherwise, use another algorithm (e.g., CH).



Transit Nodes: Grid Implementation

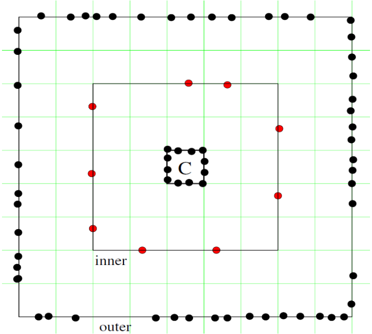
- Use a $k \times k$ grid to partition the graph.
- Define two squares centered at each cell C:
  - **inner** (5×5) and **outer** (9×9).
- Access nodes for C: vertices in B_5 on shortest paths from B_1 to B_9.
  - B_i: boundary vertex of $i \times i$ square centered at C.



[BFM+07]

Transit Nodes: Grid Implementation

- Query algorithm:
  - If s and t are > 4 cells apart, do table lookups.
  - Otherwise, do simplified reach-pruned Dijkstra.



[BFM$^+$07]

Transit Nodes: Grid Implemtation

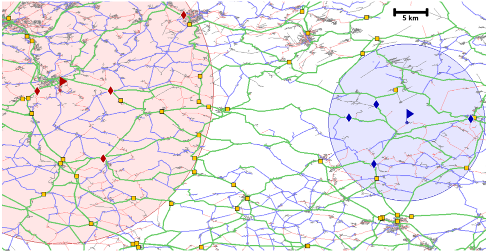
- USA graph [BFM+07]:

grid size	$\lvert\mathcal{T}\rvert$	$\lvert A(v)\rvert$	global queries	prep. time (minutes)
64×64	2 042	11.4	91.7%	498
128×128	7 426	11.4	97.4%	525
256×256	24 899	10.6	99.2%	638
512×512	89 382	9.7	99.8%	859
1024×1024	351 484	9.1	99.9%	964

- Tuned algorithm:
  - Two levels (top with 128×128, bottom with 256×256).
    - ⋆ bottom (hash) table only stores distances not covered by top table.
  - with some compression techniques, needs 21 bytes/node.
  - queries: 12 μs global (99%), 5112 μs local (1%).
  - average: 63μs

Transit Nodes: Hierarchical Version

1. Perform hierarchy-based preprocessing (e.g., CH).
2. Pick ∼10 000 most important nodes as transit nodes;
   - compute distance table for them.
3. Store with each vertex $v \in V$:
   - $\overrightarrow{A}(v)$ and $\overleftarrow{A}(v)$ (forward and reverse access nodes):
     - run CH searches from v to find them;
   - *radius*(v): (Euclidean) distance to farthest access node.



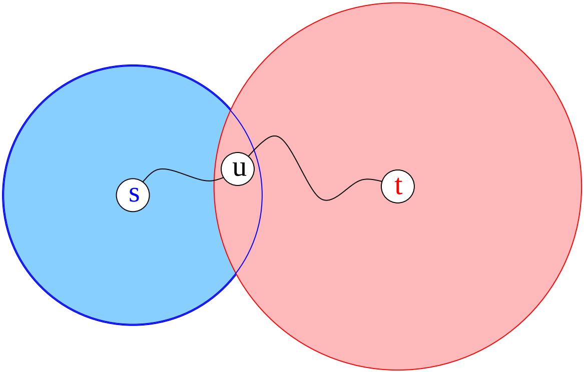
[BFM+07]

Transit Nodes: Hierarchical Version

- Query from s to t: check locality first.
- Let $\text{geo}(s, t)$ = Euclidean distance between s and t.

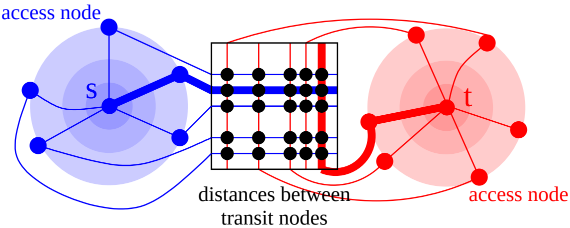
Transit Nodes: Hierarchical Version

- Query from s to t: check locality first.
- Let $\text{geo}(s, t)$ = Euclidean distance between s and t.
  - $radius(s) + radius(t) \leq \text{geo}(s, t)$: run CH query.



Transit Nodes: Hierarchical Version

- Query from s to t: check locality first.
- Let $\text{geo}(s,t)$ = Euclidean distance between s and t.
  - $\textit{radius}(s) + \textit{radius}(t) \leq \text{geo}(s,t)$: run CH query.
  - $\textit{radius}(s) + \textit{radius}(t) > \text{geo}(s,t)$: do table lookups.



Transit Nodes: Hierarchical Version

- Query from s to t: check locality first.
- Let $\text{geo}(s, t)$ = Euclidean distance between s and t.
    - $\mathit{radius}(s) + \mathit{radius}(t) \leq \text{geo}(s, t)$: run CH query.
    - $\mathit{radius}(s) + \mathit{radius}(t) > \text{geo}(s, t)$: do table lookups.
- Engineering: use three levels for best peformance.

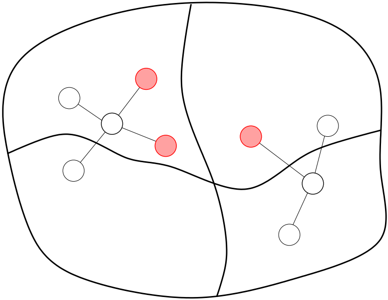
Results

- Europe (18M vertices), travel times, random pairs [BDS+10]:

	PREPROCESSING		QUERY	
METHOD	minutes	B/node	scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18
Arc Flags	2156	25	1 593	1.10
SHARC (uni)	81	15	654	0.29
SHARC (bi)	212	21	125	0.065
CHASE	99	12	45	0.017
Transit Node Routing	112	204	—	0.003

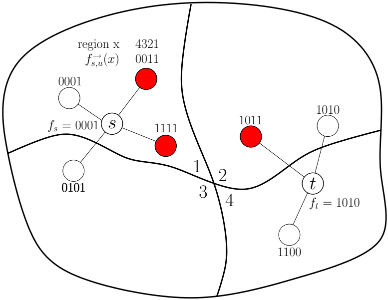
TNR+AF

- TNR considers $\overrightarrow{A}(s) \times \overleftarrow{A}(t)$ table entries:
  - no directionality!



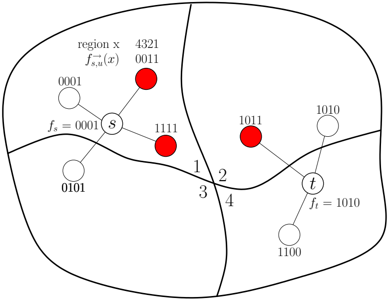
TNR+AF

- TNR considers $\overrightarrow{A}(s) \times \overleftarrow{A}(t)$ table entries:
  - no directionality!
- TNR+AF [BDS$^+$08, BDS$^+$10]:
  - Perform TNR preprocessing, find transit nodes $\mathcal{T}$.
  - Partition overlay graph $G_{\mathcal{T}} = (\mathcal{T}, E_{\mathcal{T}})$ into k regions.
  - Compute k-bit flags on the arcs (s, u), for all $s \in V$ and $u \in \overrightarrow{A}(s)$.
    - R-th bit is 1 if there is a shortest path from s to R through u;
    - same for reverse direction.
- Query only looks at relevant entries.



TNR+AF

- TNR considers $\overrightarrow{A}(s) \times \overleftarrow{A}(t)$ table entries:
  - no directionality!
- TNR+AF [BDS+08, BDS+10]:
  - Perform TNR preprocessing, find transit nodes $\mathcal{T}$.
  - Partition overlay graph $G_{\mathcal{T}} = (\mathcal{T}, E_{\mathcal{T}})$ into k regions.
  - Compute k-bit flags on the arcs (s, u), for all $s \in V$ and $u \in \overrightarrow{A}(s)$.
    - R-th bit is 1 if there is a shortest path from s to R through u;
    - same for reverse direction.
- Query only looks at relevant entries.
- Lookups on Europe: $40.9 \rightarrow 3.1$.



Results

- Europe (18M vertices), travel times, random pairs [BDS+10]:

METHOD	PREPROCESSING minutes	B/node	QUERY scans	ms
Dijkstra	—	—	8 984 289	4365
ALT(16)	13	93	82 348	120.1
RE	45	38	4 371	3.06
REAL(16)	58	109	714	0.89
REAL(64,16)	75	60	610	0.91
Contraction Hierarchies	25	−3	355	0.18
Arc Flags	2156	25	1 593	1.10
SHARC (uni)	81	15	654	0.29
SHARC (bi)	212	21	125	0.065
CHASE	99	12	45	0.017
Transit Node Routing	112	204	—	0.003
TNR + Arc Flags	229	321	—	0.002

Highway Dimension

Highway Dimension

- All these algorithms work well on road networks. Why?
- Intuitively, road networks have nice properties:
  - natural hierarchy (few verticess/arcs are really important);
  - small number of access nodes.
- We can try to formalize this.
- Assumptions [AFGW10]:
  - undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$;
  - positive, integer arc lengths;
  - diameter D;
  - constant maximum degree.

Highway Dimension

- $B_{u,d}$: ball of radius d around u (i.e., all v with $\text{dist}(u, v) \leq d$).
- $P(v, w)$: shortest path between v and w.

Definition

The **highway dimension** of $G = (V, E)$ is the smallest h such that:

- for every distance $r > 0$ and vertex $u \in V$, there exists a set S s.t.:
  - $|S| \leq h$ (S is small); and
  - $S \subseteq B_{u,4r}$ (S is a subset of a ball); and
  - S hits every shortest path $P(v, w) \subseteq B_{u,4r}$ with $|P(v, w)| > r$.

In English

For any vertex u and any distance r, there are h vertices that "hit" every long ($> r$) shortest path belonging to the ball of radius $4r$ around u.

Highway Dimension

Shortest Path Cover (SPC)

A set of vertices C is an (r, k)-**SPC** of $G = (V, E)$ iff:

- C hits every shortest path P with $r < |P| \leq 2r$ in G; and
- $|C \cap B_{u,2r}| \leq k$ for every $u \in V$.

Theorem

If G has highway dimension h, there exists an (r, h)-SPC for any r.

- Can find $(r, O(h \log n))$-SPC in polynomial time.
- Will assume we can find (r, h)-SPCs for simplicity.

Preprocessing Algorithm

1. Pick a series of $\log D$ SPCs:
   - $S_0 = V$;
   - S_i is an $(2^i, h)$-SPC, for $i > 0$.
2. Define $level(v) = i$ iff $v \in S_i$ but not higher.
   - "important" nodes have higher levels;
   - at most $\log D$ levels.
3. Perform CH-like preprocessing:
   - contract level 0, then level 1, ..., then level $\log D$.
   - arbitrary order within each level.

Theorem

This algorithm produces a graph $G^+ = (V, E \cup E^+)$ with maximum degree $O(h \log D)$.

Analysing Reach

Lemma

If v has level i, then $reach(v) \leq 2^{i+1}$ in G^+.

- If $reach(v) > 2^{i+1}$, there would be a shortest path P_{svt} s.t.:
  - $\ell(P_{sv}) \geq 2^{i+1}$
  - $\ell(P_{vt}) \geq 2^{i+1}$
- Both subpaths have nodes at level $i+1$; call them u and w.
- During preprocessing, v was eliminated before u and w.
- There would be a shortcut (u, w) bypassing v.
  - shortest s–t path would use it.

Other Algorithms

Theorem

A reach-pruned query on G^+ takes $O((h \log D)^2)$ time.

- A vertex v of reach 2^{i+1} is only scanned if $v \in B_{s,2\cdot 2^i}$.
- S_i is a $(2^i, h)$-SPC: there are at most $O(h)$ such vertices.
- $O(h \log D)$ total scans, each with $O(h \log D)$ degree.

- The same bound holds for CH.
- Can prove a bound of $O(h \log D)$ for a variant of TNR.

Extensions

Many-to-Many Computation

- **Many-to-many shortest path problem**:
  - Input: Weighted graph $G = (V, A)$, two sets $S \subseteq V$ and $T \subseteq V$.
  - Output: $|S| \times |T|$ distance table (from each $s \in S$ to each $t \in T$).
- Possible solutions:
  - run Dijkstra's algorithm $|S|$ times;
  - run $|S| \cdot |T|$ point-to-point queries.
  - can one do better?

Many-to-Many Computation: Algorithm

1. Run CH (or similar) preprocessing [KSS$^+$07].
2. Set $D[s,t] \leftarrow \infty$ for all $(s,t) \in S \times T$
3. Compute backward CH searches for each target $t \in T$:
   - a few hundred nodes visited for each t;
   - store search spaces as triples $(v, t, \text{dist}(v,t))$;
4. Partition triples into **buckets**:
   - bucket $B(v)$ has all triples of the form $(v, \cdot, \cdot)$.
5. Compute forward CH searches from each source $s \in S$:
   - When scanning v, check all triples $(v, t, \text{dist}(v,t))$ in $B(v)$:
     - set $D[s,t] \leftarrow \min\{\text{dist}(s,t), \text{dist}(s,v) + \text{dist}(v,t)\}$.

Many-to-Many Computation: Results

- Random $10^4 \times 10^4$ table [GSSD08].
  - Dijkstra $\times\, 10^4$ (full trees): $\sim$14 hours.
  - CH $\times\, 10^8$ (point-to-point): $\sim$5 hours.
  - Many-to-many with CH: 10.2 seconds;
    - plus preprocessing.

External Memory

- Implementations on portable devices [GW05, SSV08].
- Basic idea:
  - keep immutable (preprocessed) data on flash/disk;
    - bring relevant parts to RAM as needed;
  - mutable data (distance labels, heaps, ...) kept in RAM.
- Must minimize data transfer:
  - rearrange data;
  - compress data;
  - create well-defined blocks.
- CH has good performance:
  - Europe: 140 MB of flash, 69 ms query (330MHz ARM).

Handling Traffic: Dynamic Graphs

- Traffic: some edge weights increase temporarily.
- Shortest paths change in arbitrary ways.
- Solutions:
  1. Rerun full preprocessing algorithm.
  2. Rerun partial preprocessing algorithm:
     - ALT: keep landmarks, recompute distances.
     - CH: keep node ordering, recompute shortcuts.
  3. Keep preprocessing, more effort at query time:
     - ALT: lower bounds are still lower bounds (but worse).
     - CH: allow “down” moves close to changed edges.

Time-Dependent Routing

- Lengths $\ell(v, w, \tau)$ are **functions** of time:
  - how long it takes to traverse the edge if arriving at time τ;
  - usually piecewise linear.
- Dijkstra works if FIFO (non-overtaking) property holds:
  - If B leaves after A, B cannot arrive before A.
- Problem for acceleration techniques:
  - Cannot do simple bidirectional search:
    - unknown arrival time!
    - there are workarounds.
  - (Unidirectional) SHARC-based algorithm works well [Del09].

Acknowledgements

- Several pictures in this presentation were created by researchers at the University of Karlsruhe. Special thanks to Daniel Delling and Peter Sanders for allowing their use.

References

I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck.
Highway dimension, shortest paths, and provably efficient algorithms.
In *Proc. 21st SODA*, pages 782–793, 2010.

R. Bauer and D. Delling.
SHARC: Fast and robust unidirectional routing.
ACM Journal of Experimental Algorithmics, 14:2.4, 2009.

R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner.
Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm.
In C. C. McGeoch, editor, *Proc. 7th WEA*, volume 5038 of *LNCS*, pages 303–318. Springer, June 2008.

R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner.
Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm.
ACM Journal of Experimental Algorithmics, 15:2.3, 2010.

H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes.
In Transit to Constant Shortest-Path Queries in Road Networks.
In *Proc. 9th ALENEX*, pages 46–59. SIAM, 2007.

D. Delling.
Time-dependent SHARC-routing.
Algorithmica, July 2009.

D. Delling, 2010.
Personal Communication.

A. V. Goldberg and C. Harrelson.
Computing the shortest path: A* search meets graph theory.
In *Proc. 16th SODA*, pages 156–165, 2005.

A. V. Goldberg, H. Kaplan, and R. F. Werneck.
Reach for A*: Efficient point-to-point shortest path algorithms.
In *Proc. 8th ALENEX*, pages 129–143. SIAM, 2006.

A. V. Goldberg, H. Kaplan, and R. F. Werneck.

Reach for A*: Shortest path algorithms with preprocessing.
In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS*, pages 93–140. AMS, 2009.

A. V. Goldberg.
A practical shortest path algorithm with linear expected time.
SIAM J. Computing, 37:1637–1655, 2008.

R. Geisberger, P. Sanders, D. Schultes, and D. Delling.
Contraction hierarchies: Faster and simpler hierarchical routing in road networks.
In C. C. McGeoch, editor, *Proc. 7th WEA*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

R. J. Gutman.
Reach-based routing: A new approach to shortest path algorithms optimized for road networks.
In *Proc. 6th ALENEX*, pages 100–111. SIAM, 2004.

A. V. Goldberg and R. F. Werneck.
Computing point-to-point shortest paths from external memory.
In *Proc. 7th ALENEX*, pages 26–40. SIAM, 2005.

M. Hilger.
Accelerating point-to-point shortest path computations in large scale networks.
Master's thesis, Technische Universität Berlin, 2007.

M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling.
Fast point-to-point shortest path computations with arc-flags.
In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS*, pages 41–72. AMS, 2009.

T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh.
A fast algorithm for finding better routes by AI search techniques.
In *Proc. Vehicle Navigation and Information Systems Conference*, pages 2037–2044. IEEE, 1994.

S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner.
Computing many-to-many shortest paths using highway hierarchies.
In *Proc. 9th ALENEX*, pages 36–45. SIAM, 2007.

U. Lauther.
An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background.
In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm.
Partitioning graphs to speedup dijkstra's algorithm.
ACM Journal of Experimental Algorithmics, 11:2.8, 2006.

P. Sanders and D. Schultes.
Highway Hierarchies Hasten Exact Shortest Path Queries.
In *Proc. 13th ESA*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

P. Sanders and D. Schultes.
Engineering Highway Hierarchies.
In *Proc. 14th ESA*, LNCS, pages 804–816. Springer, 2006.

P. Sanders, D. Schultes, and C. Vetter.
Mobile Route Planning.
In *Proc. 16th ESA*, volume 5193 of *LNCS*, pages 732–743. Springer, September 2008.