Efficient Point-to-Point Shortest Path Algorithms

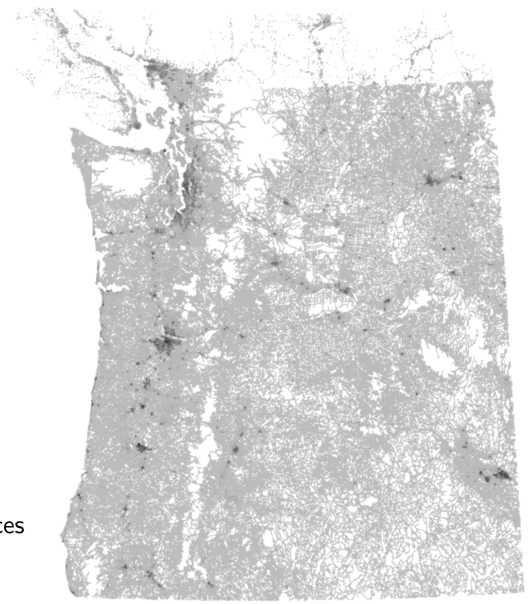
Andrew V. Goldberg (Microsoft Research)
Chris Harrelson (Google)
Haim Kaplan (Tel Aviv University)
Renato F. Werneck (Princeton University)

Shortest Paths

- Point-to-point shortest path problem (P2P):
  - Given:
    - directed graph with nonnegative arc lengths $\ell(v, w)$;
    - source vertex s;
    - target vertex t.
  - Goal: find shortest path from s to t.

- Our study:
  - Large road networks:
    - 330K (Bay Area) to 30M (North America) vertices.
  - Algorithms work in two stages:
    - preprocessing: may take hours, outputs linear amount of data;
    - query: should take milliseconds, uses the preprocessed data.

Example Graph



Northwest
n = 1.6M vertices
m = 3.8M arcs

Obvious Algorithm

- Precompute all shortest paths and store distance matrix.
- Will not work on large graphs (n = 30M).
  - $O(n^2)$ space: ∼26 PB.
  - $\tilde{O}(nm)$ time: years (single Dijkstra takes ∼10s).

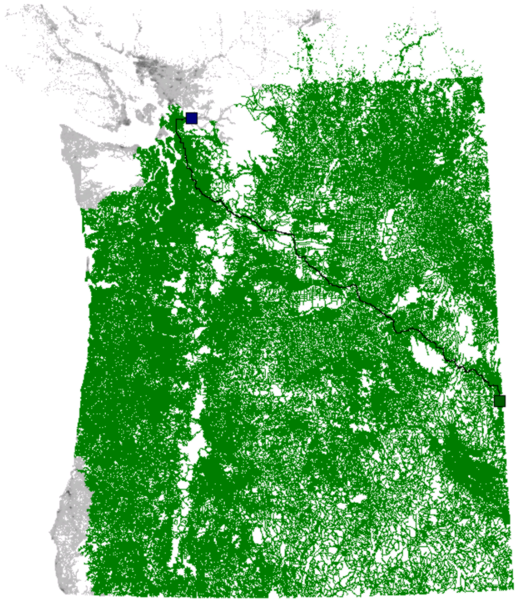
(All times on a 2.4 GHz AMD Opteron with 16 GB of RAM.)

Dijkstra's Algorithm

- Vertices processed in increasing order of distance:
  - maintains a distance label $d(v)$ for each vertex:
    - upper bound on dist(s, v);
    - initially, $d(s) = 0$ and $d(v) = \infty$ for all other vertices.
  - In each iteration:
    - Pick unscanned vertex v with smallest $d(\cdot)$ (use heap).
    - Scan v:
      - For each edge (v, w), check if $d(w) > d(v) + \ell(v, w)$.
      - If it is, set $d(w) \leftarrow d(v) + \ell(v, w)$.
  - Stop when the target t is about to be scanned.
  - [Dijkstra'59, Dantzig'63].

- Intuition:
  - grow a ball around s and stop when t is scanned.

Dijkstra's Algorithm

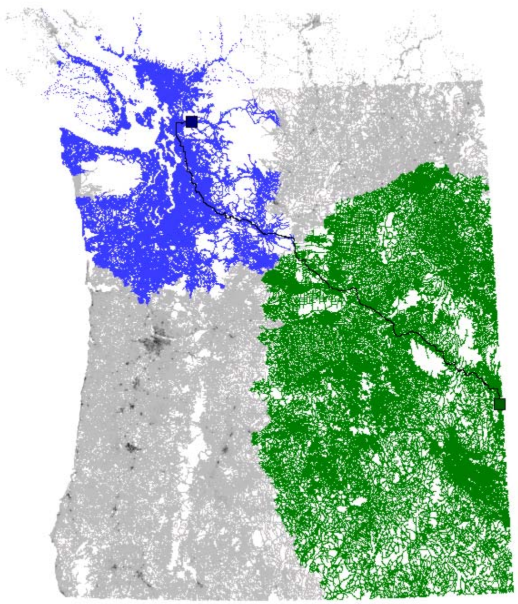


Bidirectional Dijkstra's Algorithm

- Bidirectional Dijkstra's algorithm:
  - forward search from s with labels d_f:
    - performed on the original graph.
  - reverse search from t with labels d_r:
    - performed on the reverse graph;
    - same set of vertices, each arc (v, w) becomes (w, v).
  - alternate in any way.

- Intuition: grow a ball around each end (s and t) until they "meet".

Bidirectional Dijkstra's Algorithm



Bidirectional Dijkstra's Algorithm

- Possible stopping criterion:
  - a vertex v is about to be scanned a second time:
    - once in each direction;
  - v may not be on the shortest path.

- We must maintain the length μ of the best path seen so far:
  - initially, $\mu = \infty$;
  - when scanning an arc (v, w) in the forward search and w is scanned in the reverse search, update μ if $d_f(v) + \ell(v, w) + d_r(w) < \mu$.
  - similar procedure if scanning an arc in the reverse search.

Bidirectional Dijkstra's Algorithm

- Stronger stopping condition:
  - Let top_f and top_r be the top heap values (forward and reverse).
  - Stop when $\text{top}_f + \text{top}_r \geq \mu$.
  - Previous stopping criterion is a special case.

- Why does it work?
  - Suppose there exists an s-t path P with length less than μ.
  - There must be an arc (v, w) on this path such that:
    - $\text{dist}(s, v) < \text{top}_f$ and
    - $\text{dist}(w, t) < \text{top}_r$.
  - Both v and w have been scanned already.
  - When the second of these was scanned, it would have found the P.
    - Contradiction: P cannot exist.

Part I: A* Search

A* Search

- Define potential function $\pi(v)$ and modify lengths:
  - $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$
  - $\ell_\pi(v, w)$: reduced cost of arc (v, w).
- All s-t paths change by same amount: $\pi(t) - \pi(s)$.

- A* search:
  - Equivalent to Dijkstra on the modified graph:
    - correct if $\ell_\pi(v, w) \geq 0$ (π feasible).
  - Vertices scanned in increasing order of $k(v) = d(v) + \pi(v)$:
    - $\pi(v)$: estimate on $\text{dist}(v, t)$;
    - $k(v)$: estimated length of shortest s-t path through v.
  - If $\pi(t) = 0$ and π feasible, $\pi(v)$ is a lower bound on $\text{dist}(v, t)$.
- All we need are good feasible lower bounds (e.g., Euclidean).

A* Search

- Why is A* equivalent to Dijkstra on the modified graph?
  - Dijkstra picks vertices with increasing (modified) distance from s:
    - $\text{dist}_\pi(s,v) = \text{dist}(s,v) - \pi(s) + \pi(v)$
  - A^* search picks vertices with increasing key:
    - $k(v) = \text{dist}(s,v) + \pi(v)$
  - $\pi(s)$ is constant: these orders are the same.

- Why is $\pi(v)$ a lower bound on $\text{dist}(v,t)$ when π is feasible and $\pi(t) = 0$?
  - Take the shortest path from v to t.
  - Two ways of computing its reduced cost:
    1. $\text{dist}(v,t) - \pi(v) + \pi(t) = \text{dist}(v,t) - \pi(v)$ (since $\pi(t) = 0$);
    2. sum of the reduced costs of all arcs:
       - must be nonnegative, since π is feasible.
  - Combining them: $\text{dist}(v,t) - \pi(v) \geq 0 \Rightarrow \pi(v) \leq \text{dist}(v,t)$.

Bidirectional A* Search

- Bidirectional search needs two potential functions:
  - $\pi_f(v)$: estimate on $\text{dist}(v,t)$.
  - $\pi_r(v)$: estimate on $\text{dist}(s,v)$.
- Reduced cost of arc (v,w):
  - Forward: $\ell_f(v,w) = \ell(v,w) - \pi_f(v) + \pi_f(w)$.
  - Reverse: $\ell_r(w,v) = \ell(v,w) - \pi_r(w) + \pi_r(v)$.
    - the arc appears as (w,v) in the reverse graph.
- These values must be consistent:

$$\begin{aligned} \ell_f(v,w) &= \ell_r(w,v) \\ \ell(v,w) - \pi_f(v) + \pi_f(w) &= \ell(v,w) - \pi_r(w) + \pi_r(v) \\ \pi_f(w) + \pi_r(w) &= \pi_f(v) + \pi_r(v) \end{aligned}$$

- This must be true for all pairs (v,w), i.e., $(\pi_f + \pi_r) = \text{constant}$.

Bidirectional A* Search

- Must use consistent potential functions.
- In general, two arbitrary feasible functions π_f and π_r are not consistent.
- Their average is both feasible and consistent [Ikeda et al. 94]:
  - $p_f(v) = \frac{1}{2}(\pi_f(v) - \pi_r(v))$
  - $p_r(v) = \frac{1}{2}(\pi_r(v) - \pi_f(v)) = -p_f(v)$
- To make the algorithm more intuitive, we make:
  - $p_f(v) = \frac{1}{2}(\pi_f(v) - \pi_r(v)) + \frac{\pi_r(t)}{2}$
  - $p_r(v) = \frac{1}{2}(\pi_r(v) - \pi_f(v)) + \frac{\pi_f(s)}{2}$
  - Added terms are constant: functions still feasible and consistent.
  - When π_f and π_r are lower bounds, $p_f(t) = 0$ and $p_r(s) = 0$.
- p usually provides worse bounds than π:
  - still worth it in practice.

Bidirectional A* Search

- Standard bidirectional Dijkstra:
  - stop when $\text{top}_f + \text{top}_r \geq \mu$.
    - top_f: length of the path from s to top element of forward heap.
    - top_r: length of (reverse) path from t to top element of reverse heap.
    - μ: best s-t path seen so far.
- Bidirectional A* search: same, but on the modified graph:
  - Let v_f and v_r be the top elements in each heap;
  - Length of path s-v_f is $d_f(v_f) + p_f(v_f) - p_f(s) = \text{top}_f - p_f(s)$.
  - Length of reverse path t-v_r is $d_r(v_r) + p_r(v_r) - p_r(t) = \text{top}_r - p_r(t)$.
  - Stopping criterion:

$$[\text{top}_f - p_f(s)] + [\text{top}_r - p_r(t)] \geq [\mu - p_f(s) + p_f(t)]$$

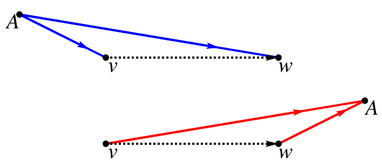
  - Simplifying and using $p_f(t) = 0$:

$$\text{top}_f + \text{top}_r \geq \mu + p_r(t).$$

Lower Bounds

- Preprocessing:
  - select a constant number of landmarks (we use 16);
  - for each landmark, precompute distance to and from every vertex.

- Lower bounds use the triangle inequality:



$$\text{dist}(v,w) \geq \text{dist}(A,w) - \text{dist}(A,v)$$

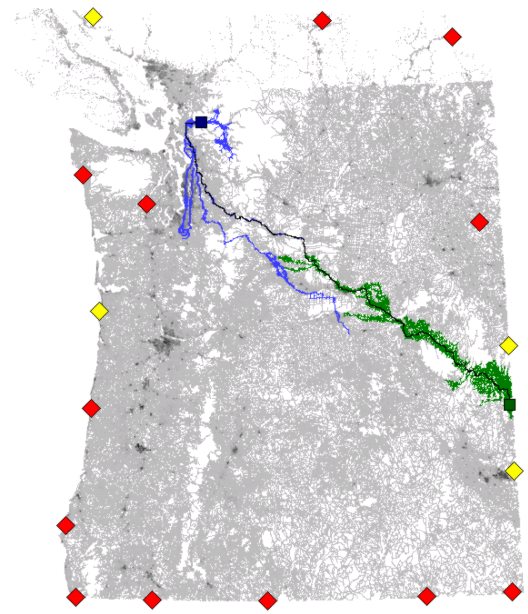
$$\text{dist}(v,w) \geq \text{dist}(v,A) - \text{dist}(w,A)$$

$$\text{dist}(v,w) \geq \max\{\text{dist}(A,w) - \text{dist}(A,v), \text{dist}(v,A) - \text{dist}(w,A)\}$$

- A good landmark appears "before" v or "after" w.

- More than one landmark: pick maximum (still feasible).

Query with Landmarks



Experimental Results

- Northwest (1 649 045 vertices), 1000 random pairs:

	PREPROCESSING		QUERY		
METHOD	minutes	MB	avgscan	maxscan	ms
Bidirectional Dijkstra	—	28	518 723	1 197 607	340.74
Landmarks	4	132	16 276	150 389	12.05

- Vertices scanned: ∼1% on average, ∼10% on bad cases.

Landmark Selection

- Landmark selection happens in two stages.

- Preprocessing:
  - Pick a small number of landmarks (we use 16).
    - ∗ more landmarks: better queries, more space.
  - Store on disk distances to and from each landmark.

- Query (s and t known):
  - using all available landmarks is expensive;
  - pick a small subset (2 to 6) that is good for the search.

Landmark Selection during Preprocessing

- Ultimate goal:
  - There should be a landmark "behind" every s-t pair.
  - Graphs are big, cannot evaluate this exactly: use heuristics.
    - All methods are quasi-linear.
- Algorithms:
  - Simple methods: random, farthest, planar;
  - avoid: adds landmarks "behind" regions not currently covered;
  - maxcover: avoid + local search:
    - goal: maximize #arcs with zero reduced cost.
- Best in practice is maxcover:
  - queries ~3 times as fast as random;
  - preprocessing ~15 times slower.

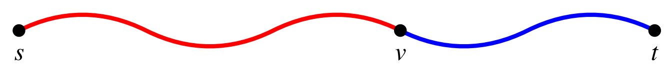
Landmark Selection at Query Time

- Use only an active subset:
  - prefer landmarks that give the best lower bound on $\text{dist}(s, t)$.
- We use dynamic selection:
  - start with two landmarks (best forward + best reverse);
  - periodically check if a new landmark would help;
  - heaps rebuilt when landmarks added.
- Performance in practice:
  - picks only ~3 landmarks;
  - fewer nodes visited than with any fixed number of landmarks.

Part II: Reach

Reaches

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



- Reach of v with respect to P:

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

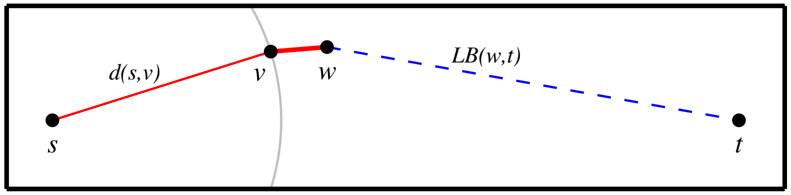
- Reach of v with respect to the whole graph:

$$\text{reach}(v) = \max_P\{\text{reach}(v, P)\},$$

  over all shortest paths P that contain v [Gutman'04].
- Intuition:
  - vertices on highways have high reach;
  - vertices on local roads have low reach.

Using Reaches

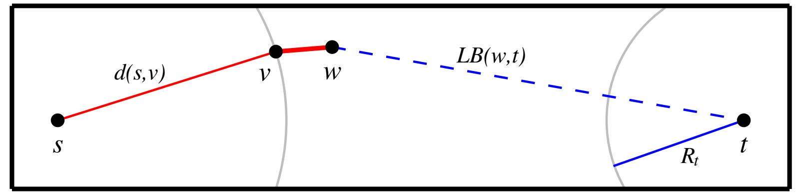
- Reaches can be used to prune the search during an s-t query.
- While scanning an edge (v, w):
  - If reach$(w) < \min\{d(s,v) + \ell(v,w), \mathrm{LB}(w,t)\}$, then w can be pruned.



- How do we obtain lower bounds?
  - Explicitly: Euclidean distances (Gutman's suggestion), landmarks.
  - Implicitly: make the search bidirectional.

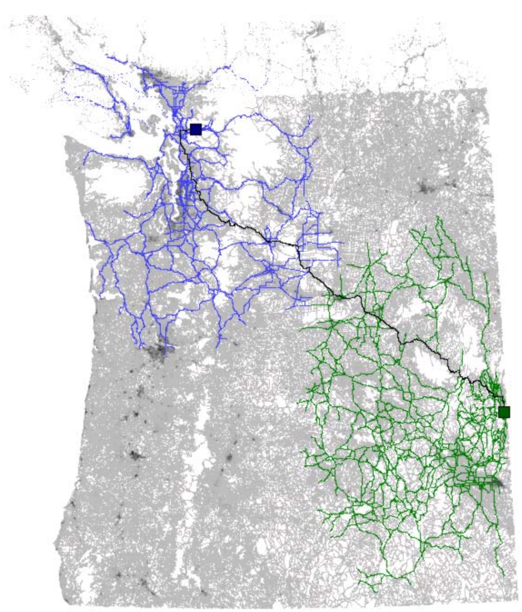
Implicit Bounds: Bidirectional Search

- Let R_t be the radius of the reverse search:
  - R_t is the value of the top element in the reverse heap;
  - if w not labeled in the reverse direction, then $d(w,t) \geq R_t$.



- Pruning test: reach$(w) < \min\{d(s,v) + \ell(v,w), R_t\}$
  - for best results, balance the forward and reverse searches by radius.

Queries with Reaches



Experimental Results

- Northwest (1 649 045 vertices), 1000 random pairs:

	PREPROCESSING		QUERY		
METHOD	minutes	MB	avgscan	maxscan	ms
Bidirectional Dijkstra	—	28	518 723	1 197 607	340.74
Landmarks	4	132	16 276	150 389	12.05
Reaches	1100	34	53 888	106 288	30.61

Computing Reaches

- Trivial algorithm:
  - compute every s-t path;
  - determine reach of each vertex on each path.

- Implementation:
  - Build shortest path tree T_r from each vertex r;
  - Determine reach of each vertex v within the tree:

$$\text{reach}(v, T_r) = \min\{\text{depth}(v), \text{height}(v)\}$$

  - Take maximum over all r.

- Runs in $\tilde{O}(nm)$ time:
  - overnight on Bay Area, years on North America.

Computing Reaches

- Query still correct with upper bounds on reaches.

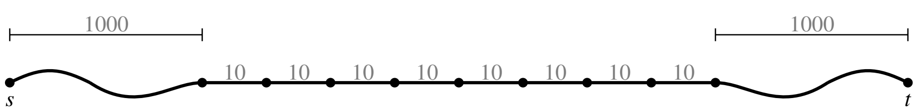
- We use iterative algorithm:
  1. find vertices with reach at most ϵ;
     - look only at partial shortest path trees (depth $\sim 2\epsilon$).
  2. eliminate vertices with small reach;
     - if no vertices remain, stop;
     - otherwise, increase ϵ and start another iteration.

- Use penalties to account for vertices already eliminated:
  - reaches no longer exact, but valid upper bounds

- Works well if many vertices are eliminated between iterations.

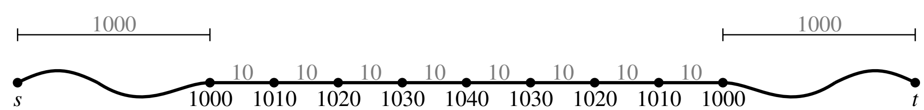
Shortcuts

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



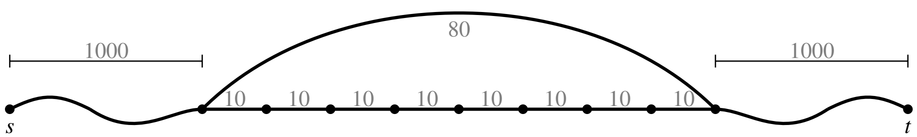
Shortcuts

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



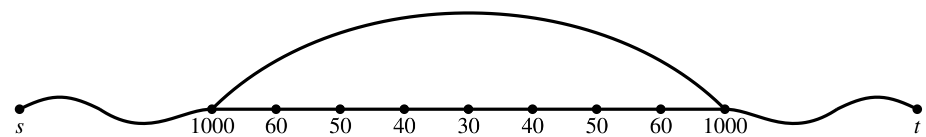
Shortcuts

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



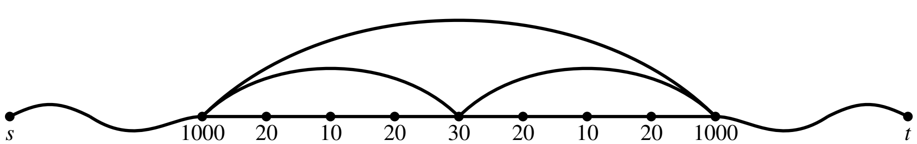
Shortcuts

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



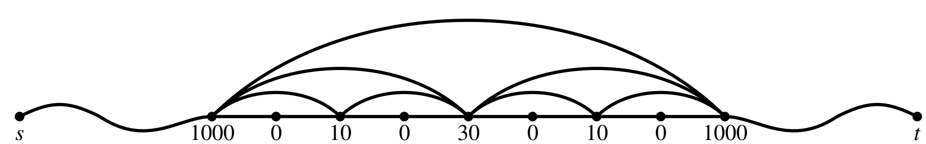
Shortcuts

- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut:
  - single edge bypassing a path (with same length).
  - assume ties are broken by taking path with fewer nodes.
- More shortcuts can be added recursively.



Shortcuts

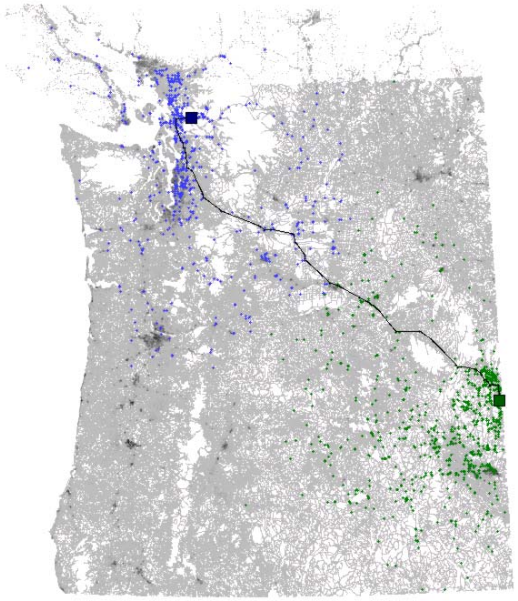
- Consider a sequence of vertices of degree two on the path below:
  - they all have high reach.
- Add a shortcut:
  - single edge bypassing a path (with same length).
  - assume ties are broken by taking path with fewer nodes.
- More shortcuts can be added recursively.



Shortcuts

- Adding shortcuts during preprocessing:
  - speeds up queries (pruning more effective);
  - speeds up preprocessing (graph shrinks faster);
  - requires slightly more space (graph has more arcs).
- Shortcuts bypass vertices of degree two:
  - some have degree two in the original graph;
  - some acquire degree two as other vertices are eliminated.
- Sanders and Schultes [ESA'05]:
  - similar idea for hierarchy-based algorithm.

Reaches with Shortcuts



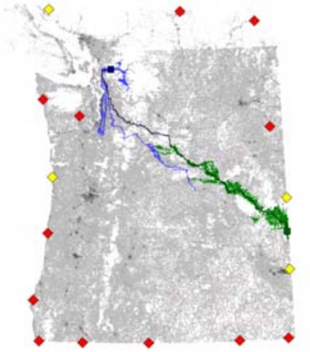
Experimental Results

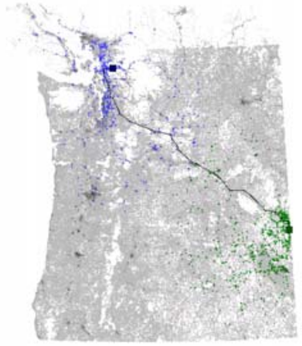
- Northwest (1 649 045 vertices), 1000 random pairs:

METHOD	PREPROCESSING minutes	PREPROCESSING MB	QUERY avgscan	QUERY maxscan	QUERY ms
Bidirectional Dijkstra	—	28	518 723	1 197 607	340.74
Landmarks	4	132	16 276	150 389	12.05
Reaches	1100	34	53 888	106 288	30.61
Reaches+Shortcuts	17	100	2 804	5 877	2.39

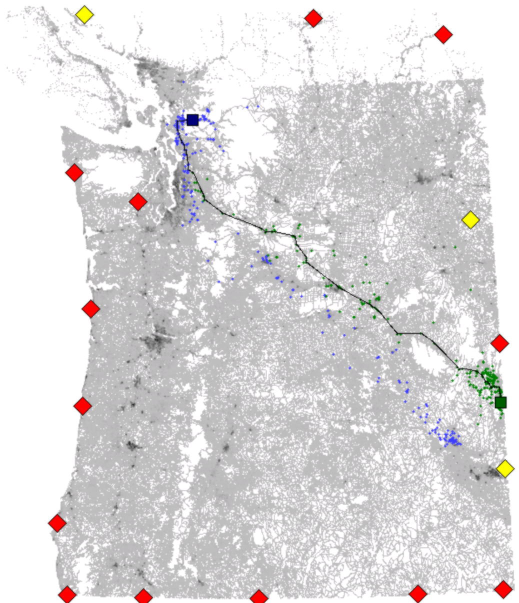
Reaches and Landmarks

- A* search with landmarks can use reaches:
  - A* gives the search a sense of direction.
  - Reaches make the search sparser.
- Landmarks have dual purpose:
  1. guide the search;
  2. provide lower bounds for reach-based pruning.





Reaches and Landmarks (with Shortcuts)



Experimental Results

- Northwest (1 649 045 vertices), 1000 random pairs:

	PREPROCESSING		QUERY		
METHOD	minutes	MB	avgscan	maxscan	ms
Bidirectional Dijkstra	—	28	518 723	1 197 607	340.74
Landmarks	4	132	16 276	150 389	12.05
Reaches	1100	34	53 888	106 288	30.61
Reaches+Shortcuts	17	100	2 804	5 877	2.39
Reaches+Shortcuts+Landmarks	21	204	367	1 513	0.73

Summary of Results

- North America (29 883 886 vertices), 1000 random pairs:

	PREPROCESS		QUERY		
METHOD	hours	GB	avgscan	maxscan	ms
Bidirectional Dijkstra	—	0.5	10 255 356	27 166 866	7 633.9
Landmarks	1.6	2.3	250 381	3 584 377	393.4
Reaches+Shortcuts	11.3	1.8	14 684	24 618	17.4
Reaches+Shortcuts+Landmarks	12.9	3.6	1 595	7 450	3.7

Future Directions

- Theory:
  - For which classes of graphs does each algorithm work?
  - How to find a good set of landmarks?
  - What is the best set of shortcuts for a given graph?
  - Is there a faster algorithm for computing exact reaches?
  - Is there a better algorithm for computing approximate reaches?
- Practice:
  - Reduce size of preprocessed data.
  - Make queries more cache-efficient.

References

- Goldberg, Harrelson, and Werneck (in preparation):
  - Goldberg and Harrelson (SODA'05):
    - "ALT algorithm" (A^* search + Landmarks + Triangle inequality).
  - Goldberg and Werneck (Alenex'05):
    - improved preprocessing and queries;
    - Pocket PC implementation.
- Goldberg, Kaplan, and Werneck (2005):
  - reach with shortcuts + A^* search.

http://www.cs.princeton.edu/~rwerneck/public.htm