Computing Point-to-Point Shortest Paths from External Memory

Andrew V. Goldberg (Microsoft Research)
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:
  - data is preprocessed to avoid looking at the whole graph:
    - #vertices visited by the algorithm;
    - efficiency: $\frac{\#\text{vertices on the shortest path}}{\#\text{vertices scanned}}$
  - road networks;
  - target architecture: Pocket PC (works on PCs also).

Target Architecture

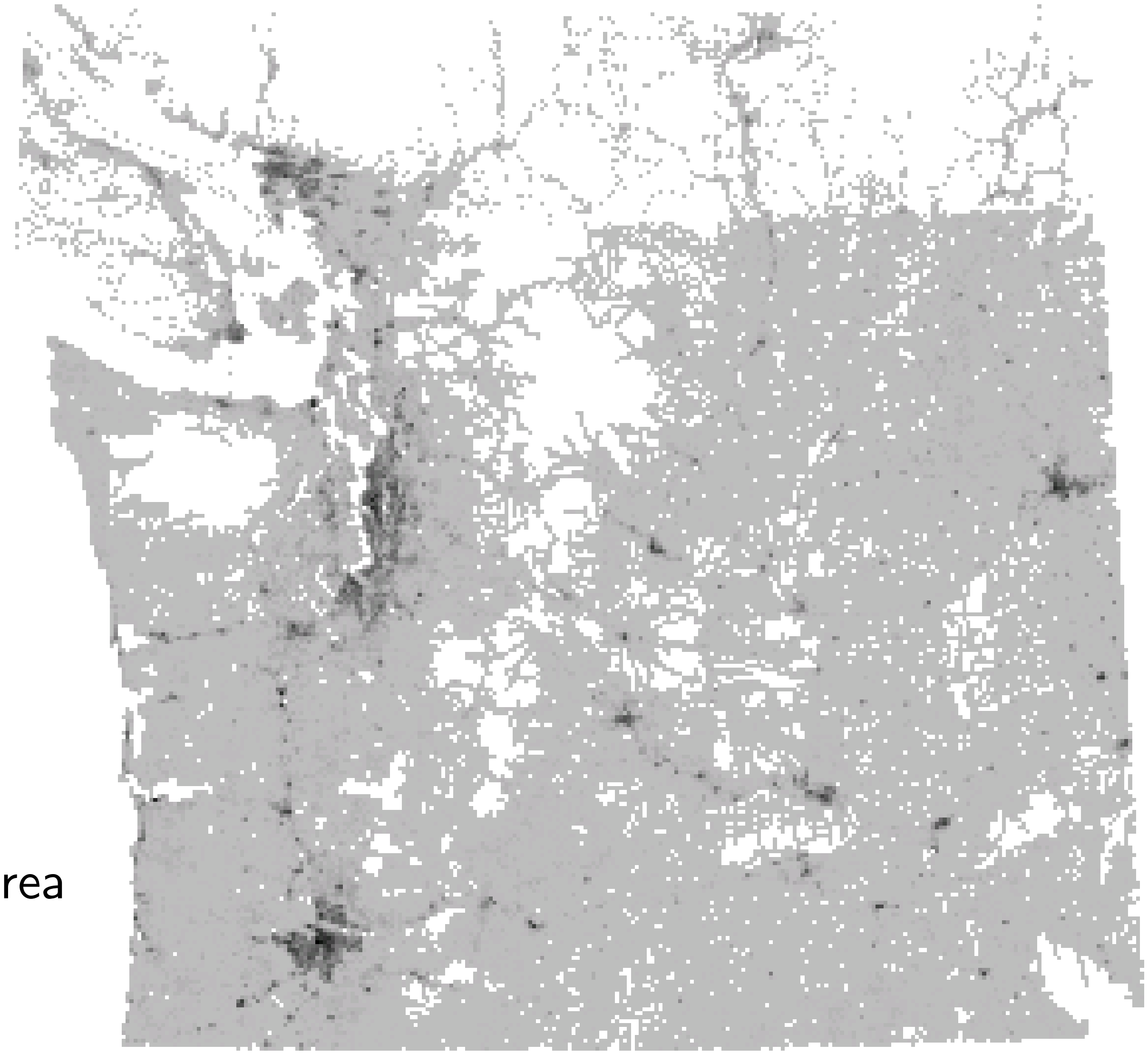
- Pocket PC:
  - Windows Mobile 2003
  - 400 MHz ARM processor
  - 128 MB of RAM
  - data read from external memory:
    - Compact Flash (4 GB, FAT32).

- Flash is bottleneck:
  - minimum block size: 512 bytes;
  - throughput: $\sim$200 KB/sec for random accesses.

Data

- North America: 30M vertices.
- Five partial graphs with 330K to 1M vertices:
  - San Francisco Bay Area
  - Los Angeles
  - St Louis
  - Dallas
  - Washington State and vicinity
- Data does not fit in RAM.

Example Graph



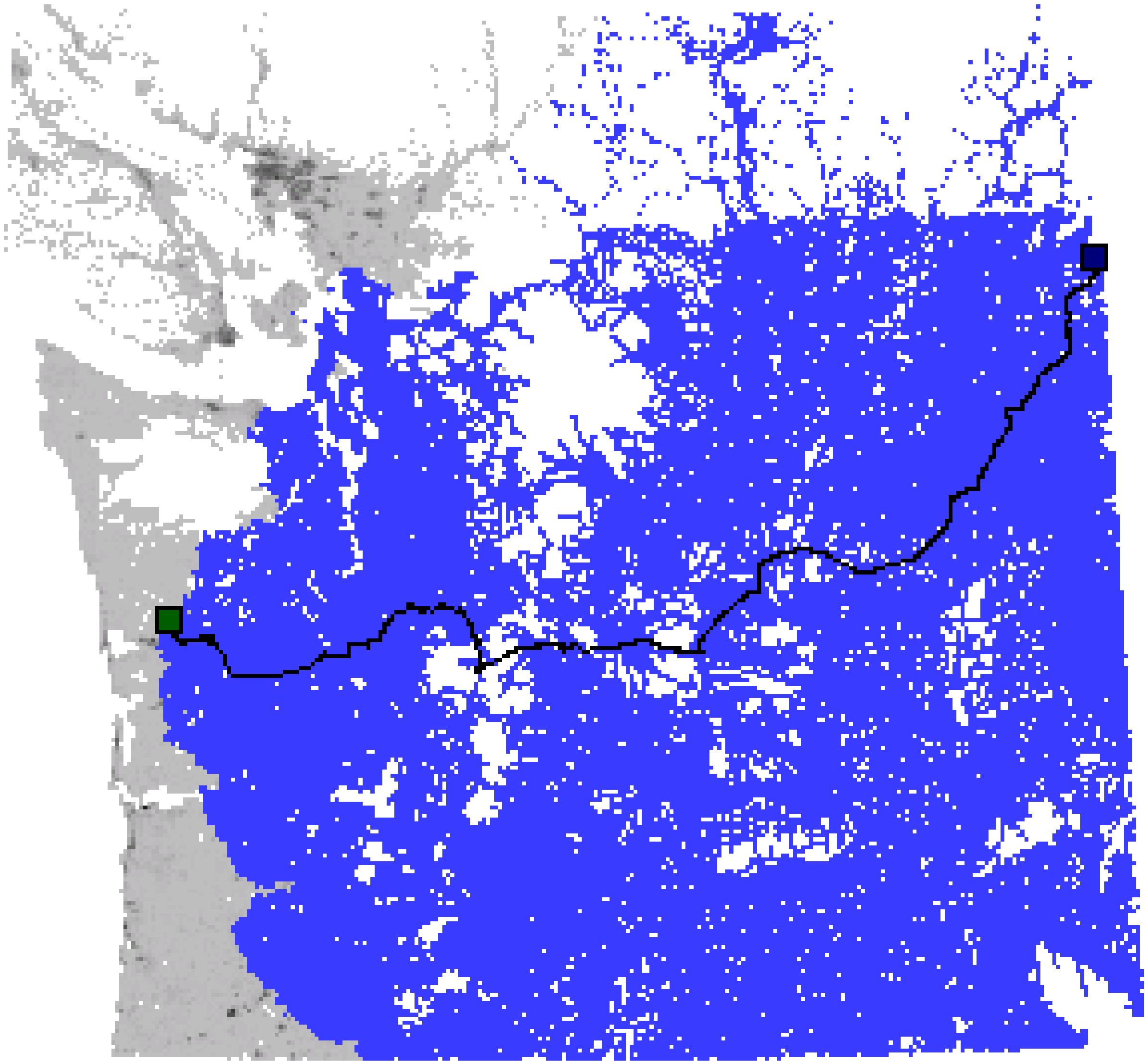
Washington Area
1M vertices
2.3M arcs

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, $d(v) = \infty$ for all vertices, except $d(s) = 0$.
  - Select unscanned vertex with smallest $d(\cdot)$.
  - Scan it, updating estimates for neighbors.
  - Stop when target is selected.
  - [Dijkstra'59, Dantzig'63].

- Intuition:
  - grows a ball around s;
  - radius is dist(s, t).

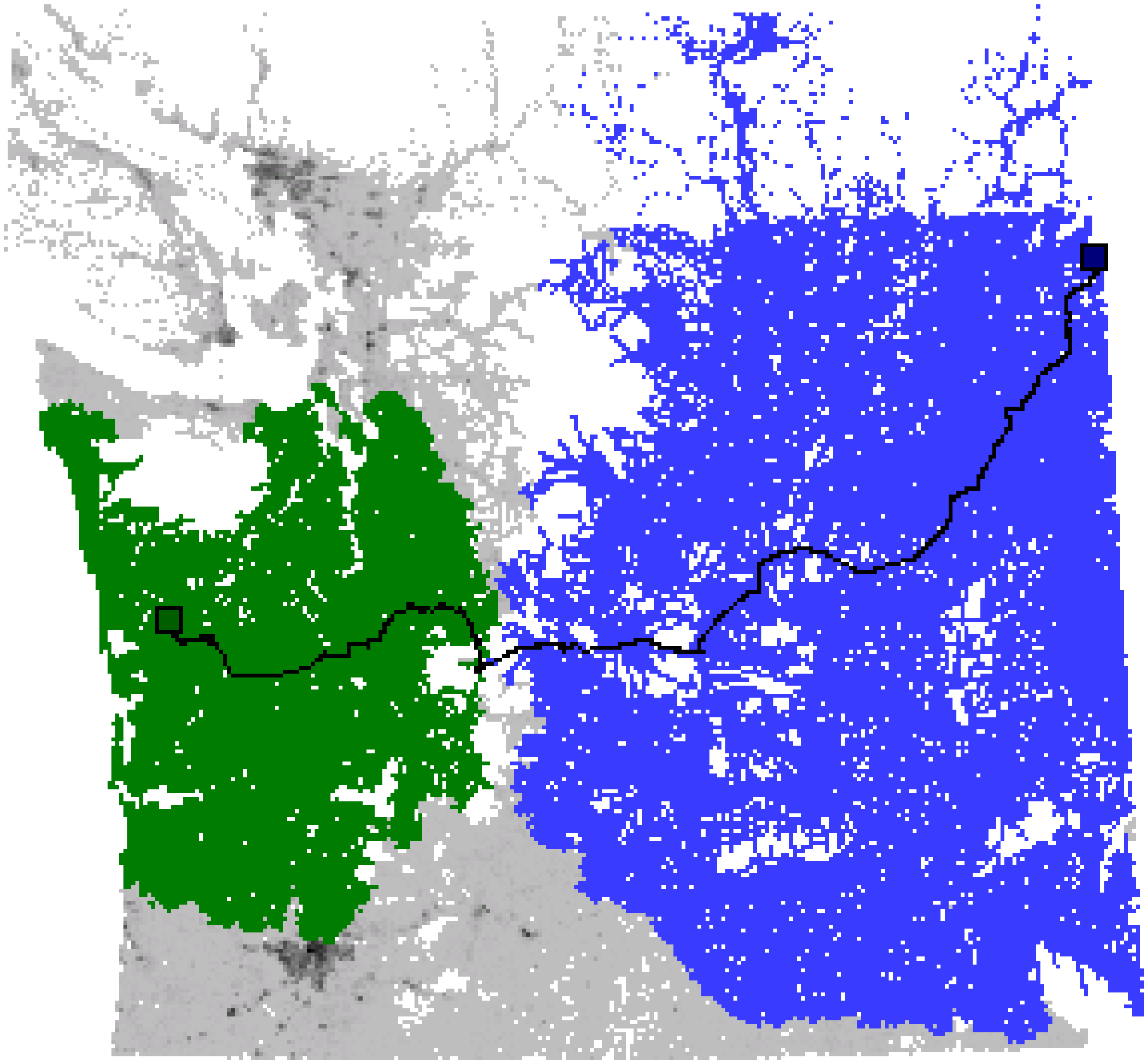
Dijkstra's Algorithm



Bidirectional Dijkstra's Algorithm

- Perform a forward search from s, as before.
- Also perform a reverse search from t:
  - similar, but on the reverse graph.
- Stop when the two searches meet.
- Intuition: grows one ball from each side.

Bidirectional Dijkstra's Algorithm



A* Search

- Similar to Dijkstra's algorithm:
  - Uses potentials $\pi(v)$, estimates on $\text{dist}(v, t)$.
  - Vertices scanned in increasing order of $k(v) = d(v) + \pi(v)$.
    - $k(v)$: estimate on length of shortest s-t path through v.

- Equivalent do Dijkstra's algorithm on graph with modified weights:
  - $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$
  - $\ell_\pi(v, w)$: reduced cost of arc (v, w).

- A* is optimal if $\ell_\pi(v, w) \geq 0$ (π feasible).

- If $\pi(t) = 0$ and π feasible, $\pi(v)$ is a lower bound on $\text{dist}(v, t)$.

Bidirectional A*

- Two searches, as in bidirectional Dijkstra's algorithm.

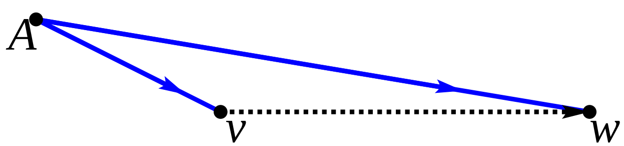
- Uses two potential functions:
  - $\pi_f(v)$: estimate on dist(v,t), for forward search.
  - $\pi_r(v)$: estimate on dist(s,v), for reverse search.

- The pair must be consistent:
  - An arc must have the same reduced cost in both searches.
  - Not true for arbitrary feasible functions.
  - True for their *average* [Ikeda et al. 94]:
    - $p_f(v) = \frac{1}{2}(\pi_f(v) - \pi_r(v))$
    - $p_r(v) = \frac{1}{2}(\pi_r(v) - \pi_f(v)) = -p_f(v)$
  - In general, p provides worse bounds than π.

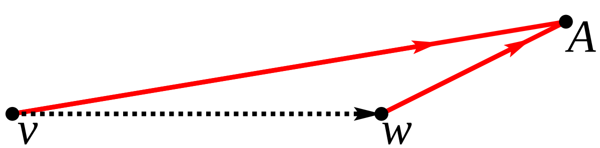
Lower Bounds

- Preprocessing:
  - Select a constant number of *landmarks*;
  - For each landmark, precompute distance to and from every vertex.

- Lower bounds use the *triangle inequality*:



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



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

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

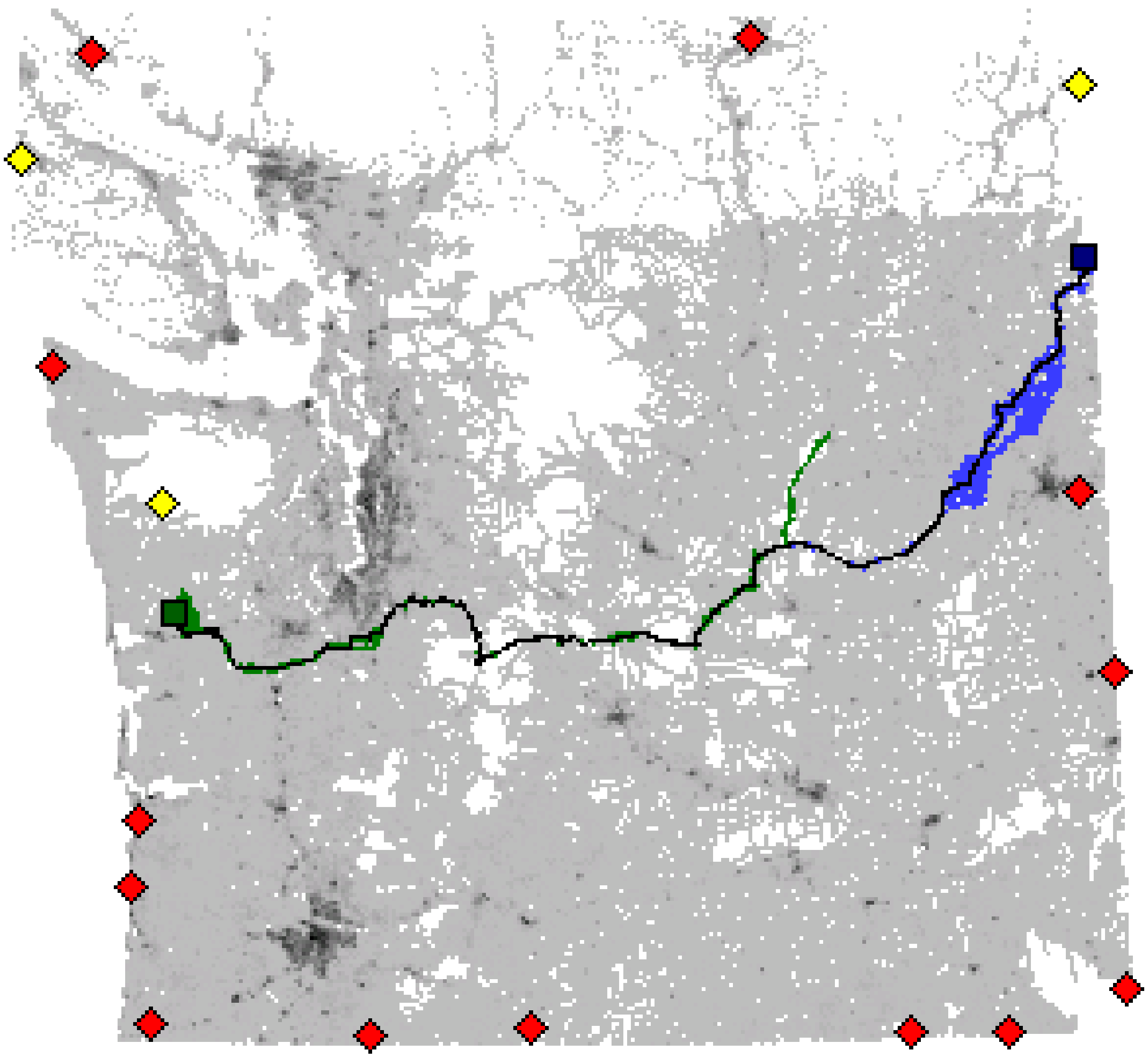
- A good landmark appears "before" v or "after" w.

- More than one landmark: pick maximum.

ALT Algorithm

- ALT = A^* search + Landmarks + Triangle inequality.
- Goldberg and Harrelson (SODA'05).

ALT Algorithm



Dealing with External Memory

- Immutable data (read-only):
  - forward and reverse graphs (adjacency lists and arc lengths);
  - distances to and from each landmark.

- Mutable data (changes during the algorithm):
  - distance labels;
  - parent pointers;
  - heap position.

- Immutable data in external memory, mutable data in RAM:
  - only visited vertices in RAM ("mutable nodes");
  - hash table keeps track of them.

Dealing with External Memory

- Caching:
  - Landmark files and graphs are cached.
  - Data read in pages:
    - good locality (neighbors have similar IDs).

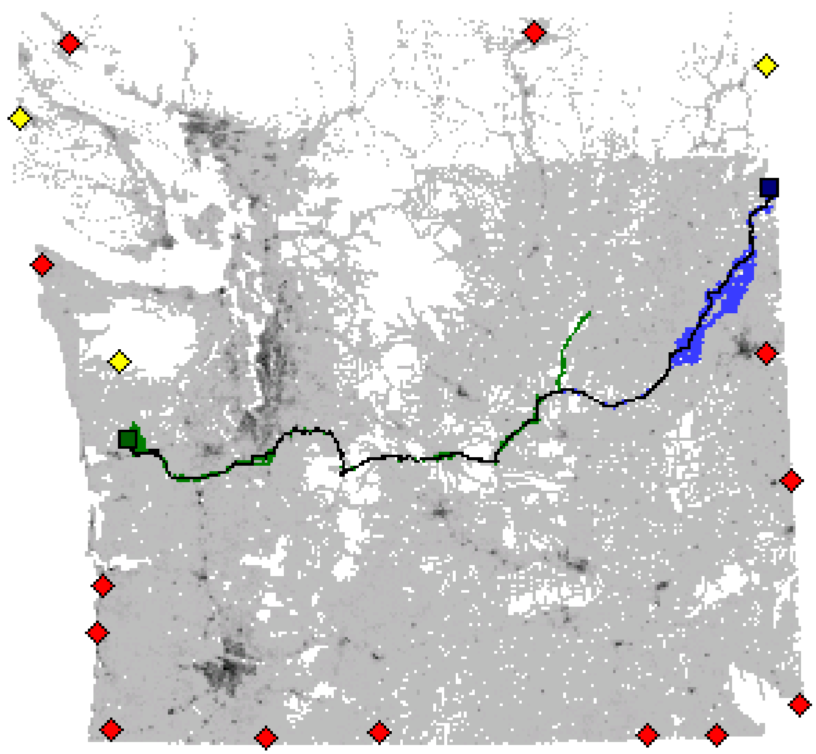
- Compression:
  - Landmark data compressed by almost 50%:
    - faster reading;
    - more landmarks fit in flash.

Important Goals

- Make the search as efficient as possible:
  - limit number of mutable nodes in RAM;
  - read as little as possible from external memory.
- We propose improvements to the ALT algorithm.

Landmarks

- Landmark selection occurs in two levels:
  - During preprocessing (PC), pick some vertices to be landmarks;
  - During the actual search (Pocket PC), pick a small subset to be active:
    - less data to read;
    - bad landmarks are not helpful anyway.



Landmark Selection During Preprocessing

- Ultimate goal:
  - For every pair s-t, there should be a landmark "behind" it.
  - Graphs are big, cannot evaluate this exactly: use heuristics.
    - All methods are $\tilde{O}(n)$.

- Two new methods:
  - *avoid*: adds landmarks "behind" regions not currently covered;
  - *maxcover*: *avoid* + local search:
    - tries to minimize the number of arcs with zero reduced cost.

- Improvements with *maxcover* (over best method in [GH05]):
  - Partial graphs: ∼25% fewer nodes visited;
  - North America: ∼50% fewer nodes visited.

Active Landmarks

- Goldberg and Harrelson [GH05] propose static selection:
  - pick the landmarks that give the best bound on dist(s, t);
  - use them during the whole search.

- We propose dynamic selection:
  - start with two landmarks (the best for each search);
  - periodically check if a new landmark will help;
  - new landmarks change the potential function:
    - we propose a new stopping criterion to handle this.

- Dynamic selection is better:
  - On average, picks only $\sim$3 landmarks;
  - Visits fewer nodes than with any fixed number of static landmarks.

Pocket PC Runs

- 100 random pairs, 23 *maxcover* landmarks.

Measure	Partial graphs	North America
Nodes visited (avg)	$\sim$1%	$\sim$1%
Nodes visited (max)	$\sim$10%	$\sim$10%
Average efficiency	29%–45%	15%
Average time	5–10 seconds	6 minutes
Data read	500–700 KB	22 MB

- Our improvements did help:
  - original implementation had efficiency $< 10\%$ for partial graphs.
- Current bottleneck: reading the data.
  - Preloading makes the algorithm 13 times faster on Bay Area.

Pocket PC Runs

- BFS distribution:
  - vertices are 50 hops away from each other in this distribution;
  - simulates “local” queries, typical in practice.

- Results for 100 pairs using 23 *maxcover* landmarks:

Nodes visited (avg)	300–700
Nodes visited (max)	1000–4000
Average efficiency	26%–43%
Average time	1–2 seconds
Data read	50–100 KB

- Graph size is not an important factor.

Final Thoughts

- Our contributions:
  - improved landmark selection (*avoid*, *maxcover*);
  - dynamic selection of active landmarks;
  - new stopping criterion;
  - external memory implementation.

- Future work:
  - direct access to flash;
  - even better landmark selection;
  - reusing active nodes;
  - proper in-memory implementation;
  - theoretical justification.

Thank You