Scalable Network Distance Browsing in Spatial Databases

Hanan Samet

hjs@cs.umd.edu

Department of Computer Science

Center for Automation Research

Institute for Advanced Computer Studies

University of Maryland

College Park, MD 20742, USA

Joint work with Jagan Sankaranarayanan and Houman Alborzi

Scalable Network Distance Browsing in Spatial Databases, SIGMOD 2008 (best paper award), Vancouver, Canada, June 2008, pp. 43–54.

Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than “as the crow flies”
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices

Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than "as the crow flies"
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

Key to Nearest Neighbor Finding in Spatial Networks

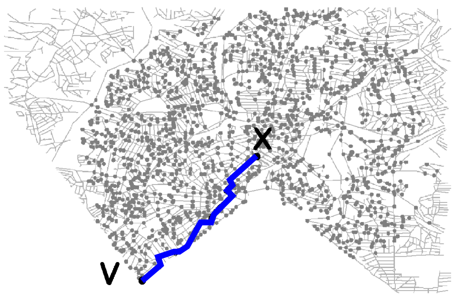
1. Use distance along a graph rather than “as the crow flies”
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of netwok
   - Implies no need to recompute shortest paths each time q or S change

Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than "as the crow flies"
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices

   - Can reduce to $O(N)$ by also using path coherence of source vertices
3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of netwok
   - Implies no need to recompute shortest paths each time q or S change

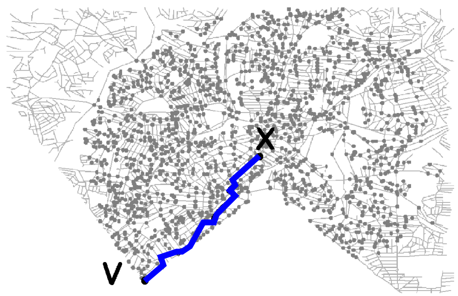


4. Avoids Dijkstra's algorithm which visits too many vertices
   - Ex: Dijkstra's algorithm visits 3191 out of the 4233 ver tices in network to identify a 76 edge path from X to V

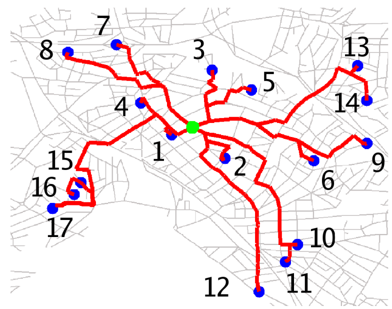
Key to Nearest Neighbor Finding in Spatial Networks

1. Use distance along a graph rather than “as the crow flies”
2. Precompute and store shortest paths between all vertices in network
   - Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$ using path coherence of destination vertices
   - Can reduce to $O(N)$ by also using path coherence of source vertices

3. Decouple domain S of query objects (q) and objects from which neighbors are drawn from domain V of vertices of netwok
   - Implies no need to recompute shortest paths each time q or S change



4. Avoids Dijkstra’s algorithm which visits too many vertices
   - Ex: Dijkstra’s algorithm visits 3191 out of the 4233 ver tices in network to identify a 76 edge path from X to V
5. Instead, only visit vertices on shortest paths to nearest neighbors



Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

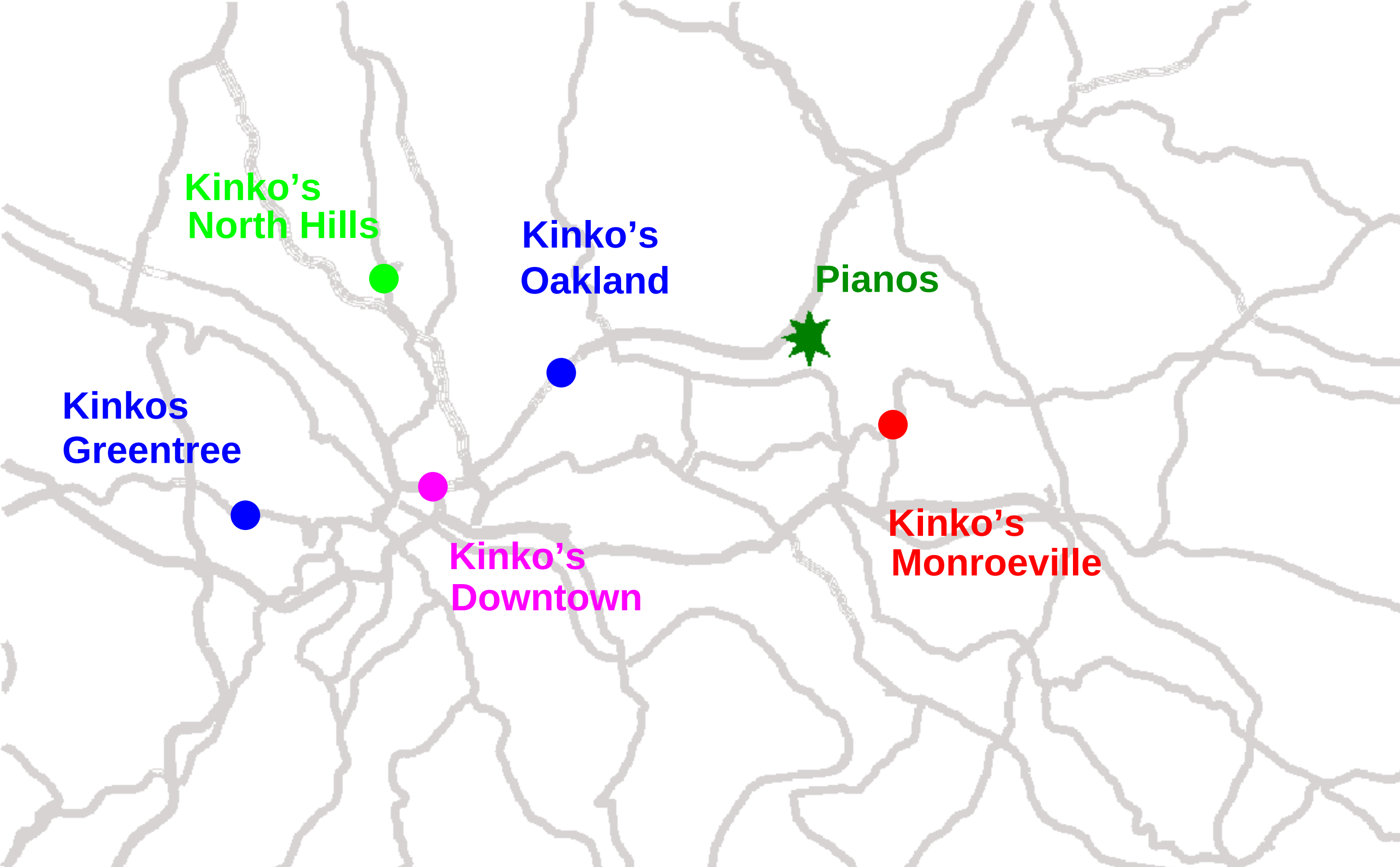
Finding Nearest Neighbors in Spatial Networks

- Spatial Network: graph with spatial components at vertices and/or edges
- Involves shortest path computation
  1. Growing popularity of online mapping services (e.g., Google Maps, Microsoft MapPoint) has led to interest in real time query processing
  2. Finding nearest objects from a set S (e.g., gas stations, restaurants, markets, etc.)
  3. Should be able to make dynamic changes in query so that once found shortest path from A to B that passes through C, can change to pass through D
- Most transportation networks can be modeled as spatial networks. e.g.,
  - Road networks
    - Each intersection is a vertex of the graph, the position of the intersection is associated with the vertex
    - Each edge of the graph corresponds to a road segment. The weight of an edge corresponds to the cost of travel (i.e., distance or time) along the corresponding road segment
  - Airline routes
  - Waterways

Application – Find the closest Kinko's

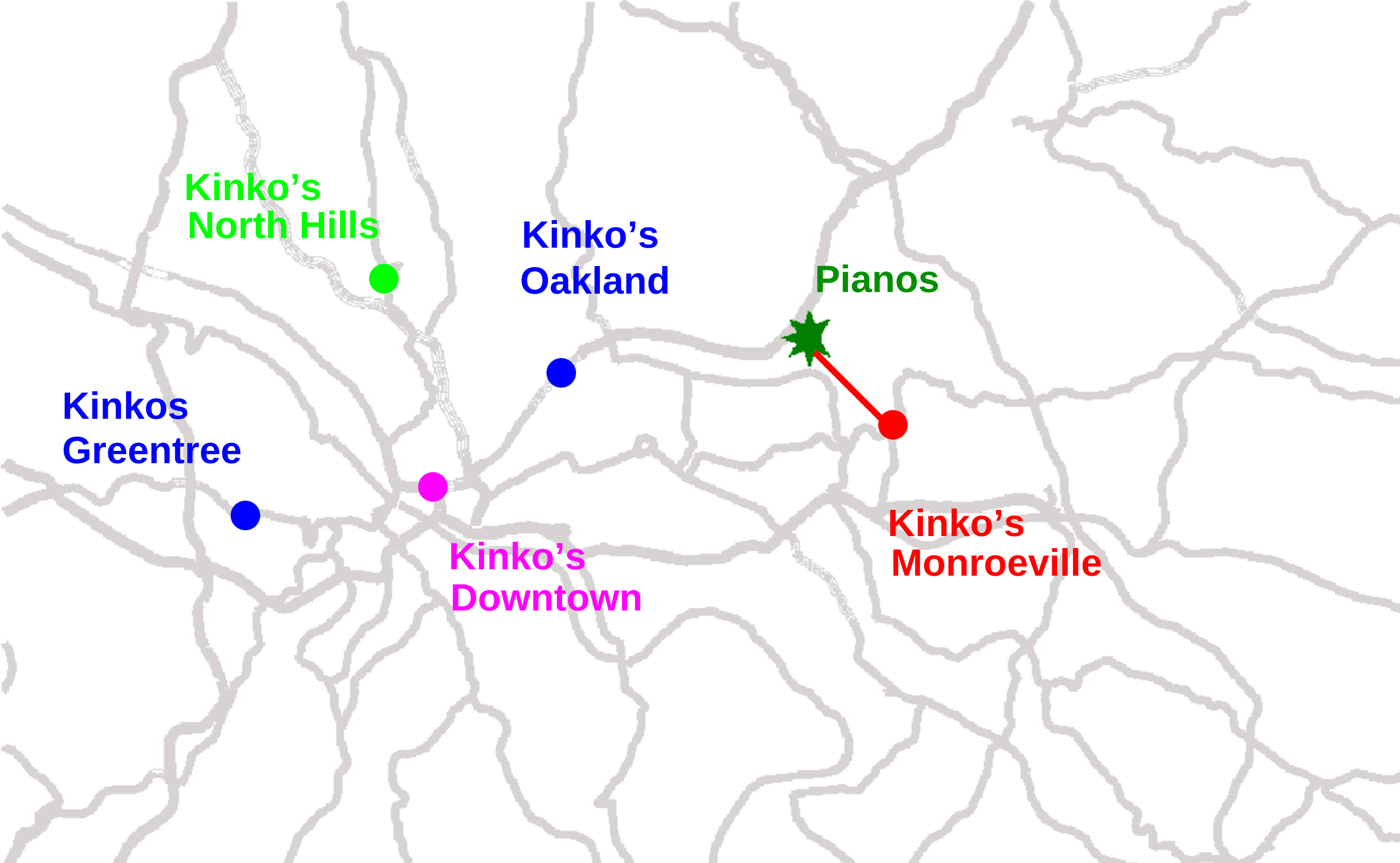
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures

Application – Find the closest Kinko’s



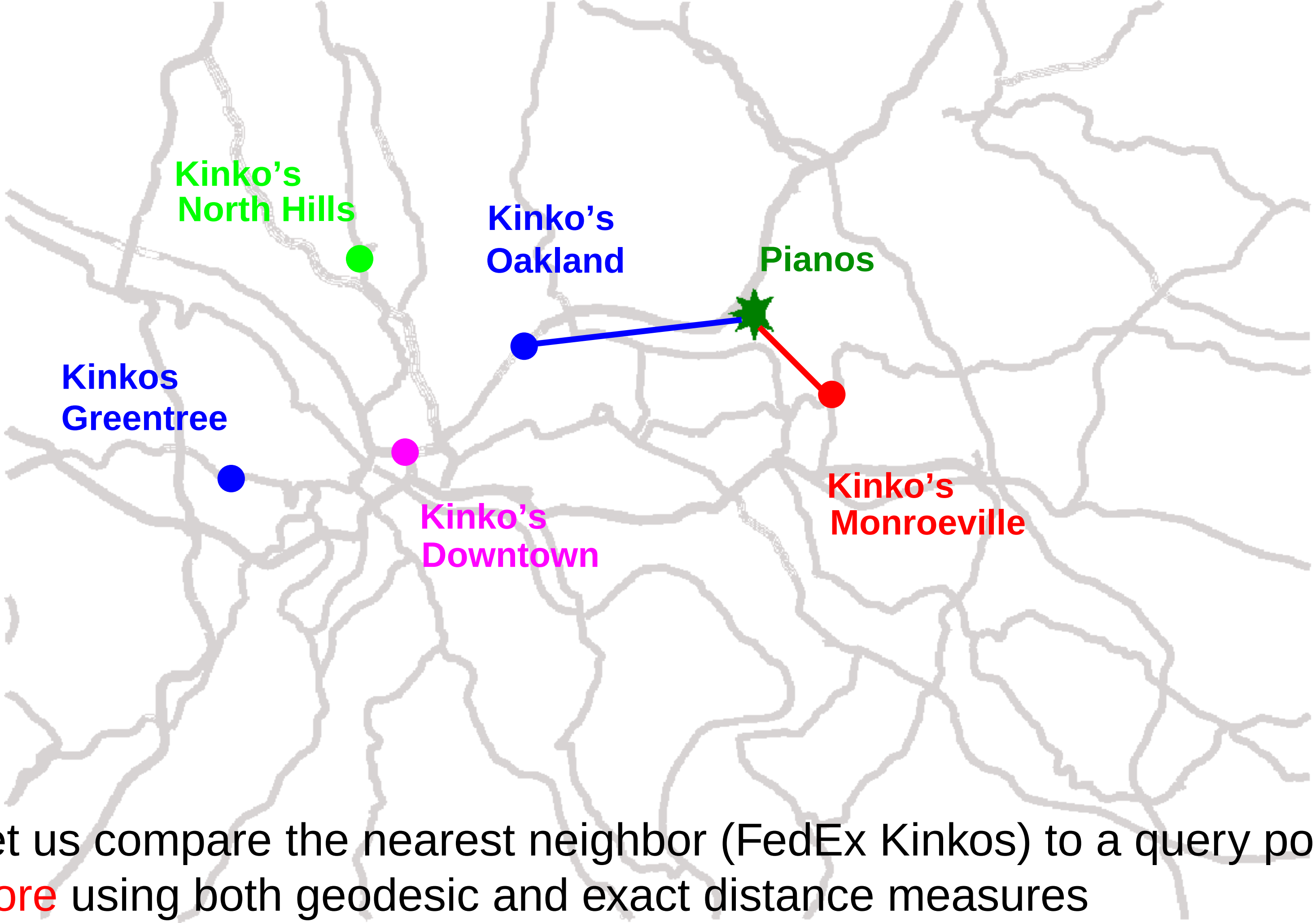
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures

Application – Find the closest Kinko's



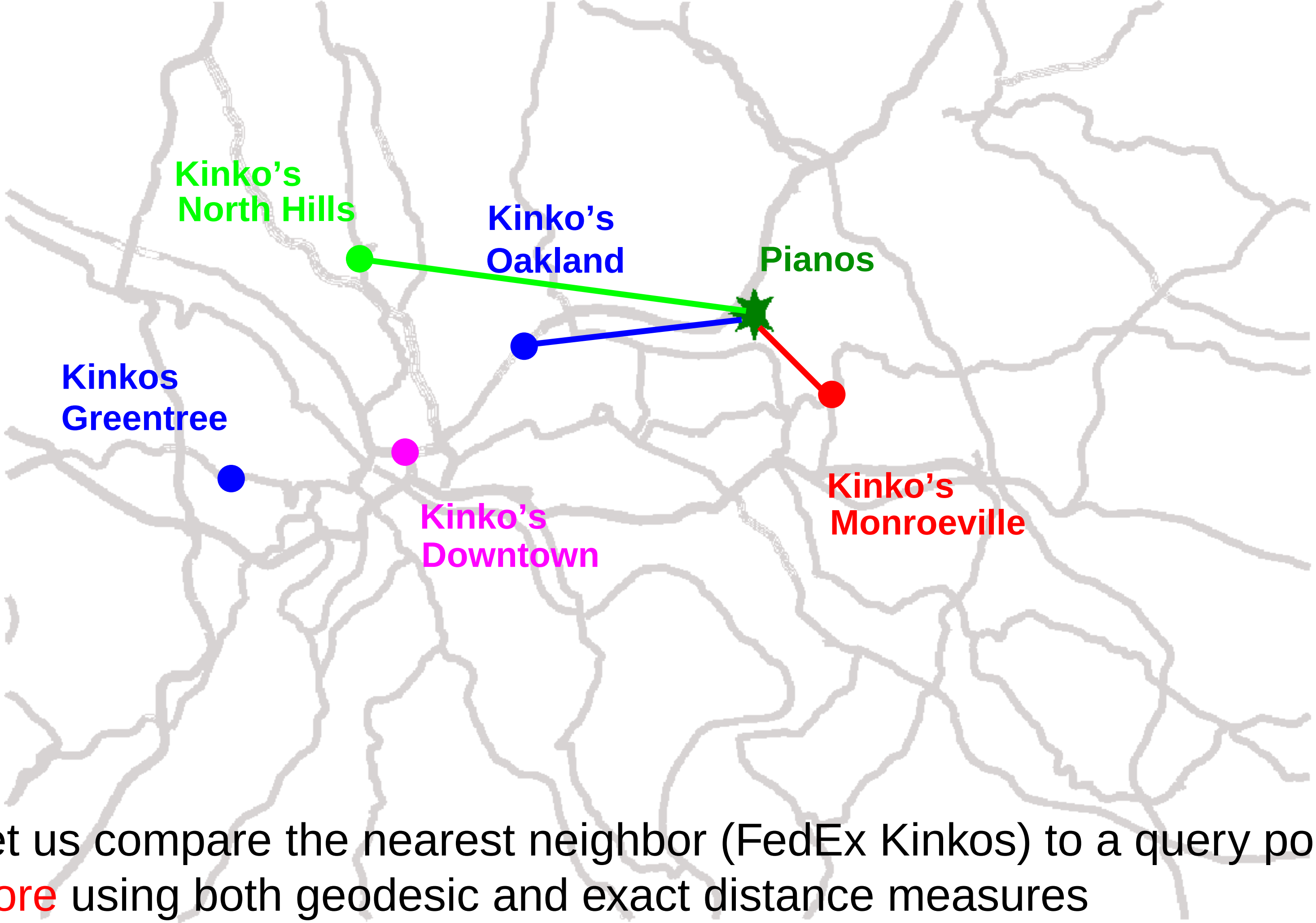
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M

Application – Find the closest Kinko's



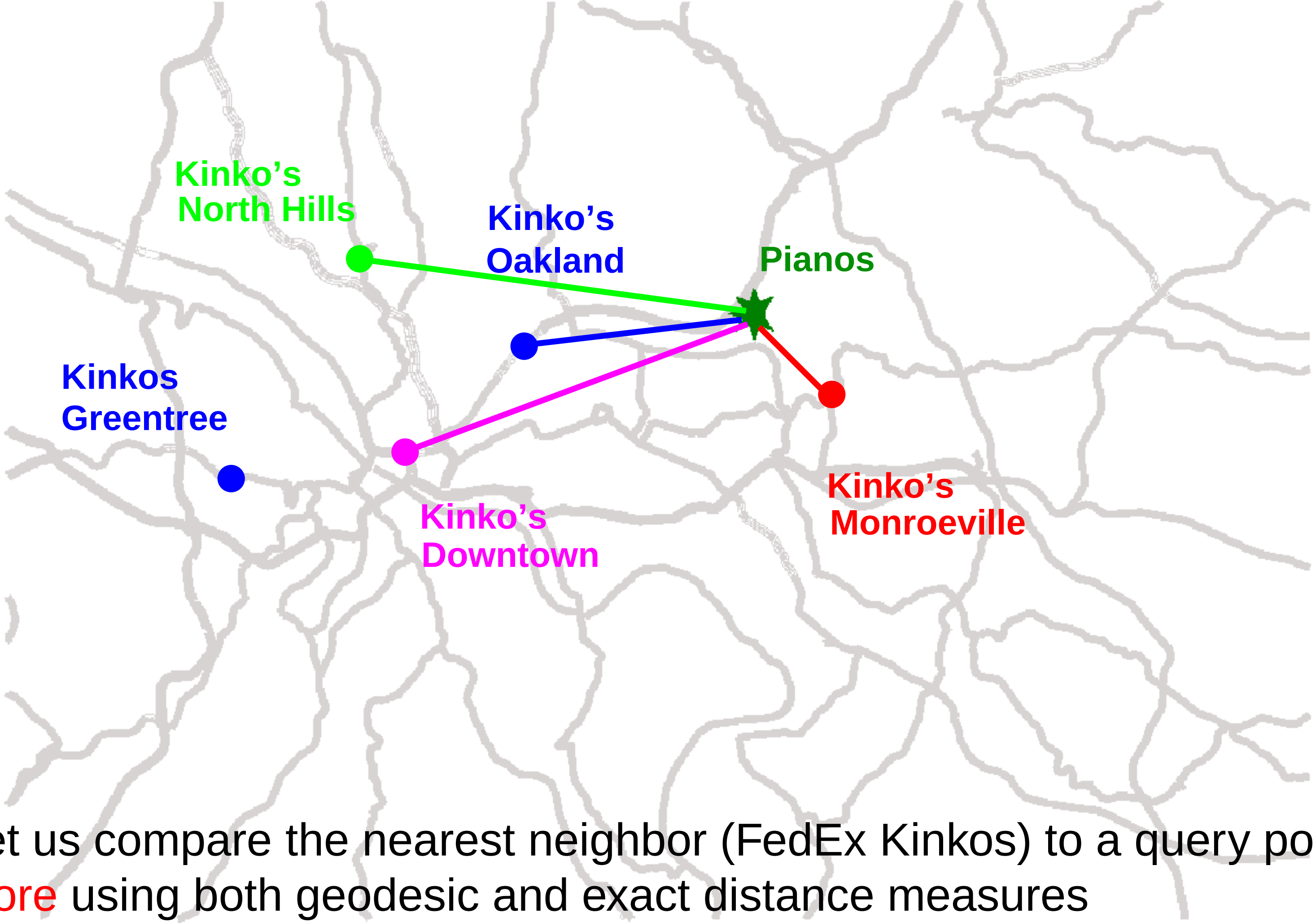
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O

Application – Find the closest Kinko's



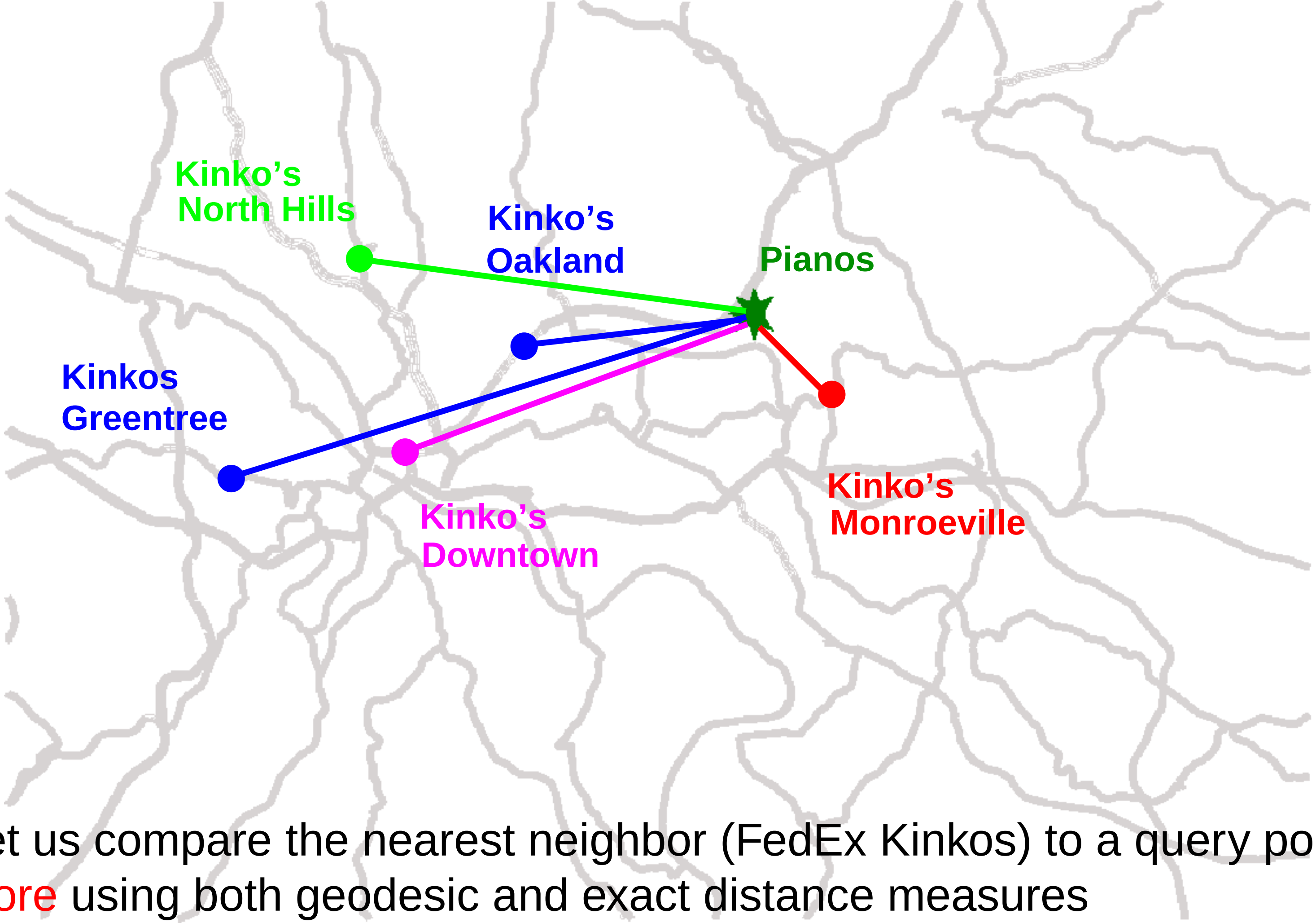
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N

Application – Find the closest Kinko's



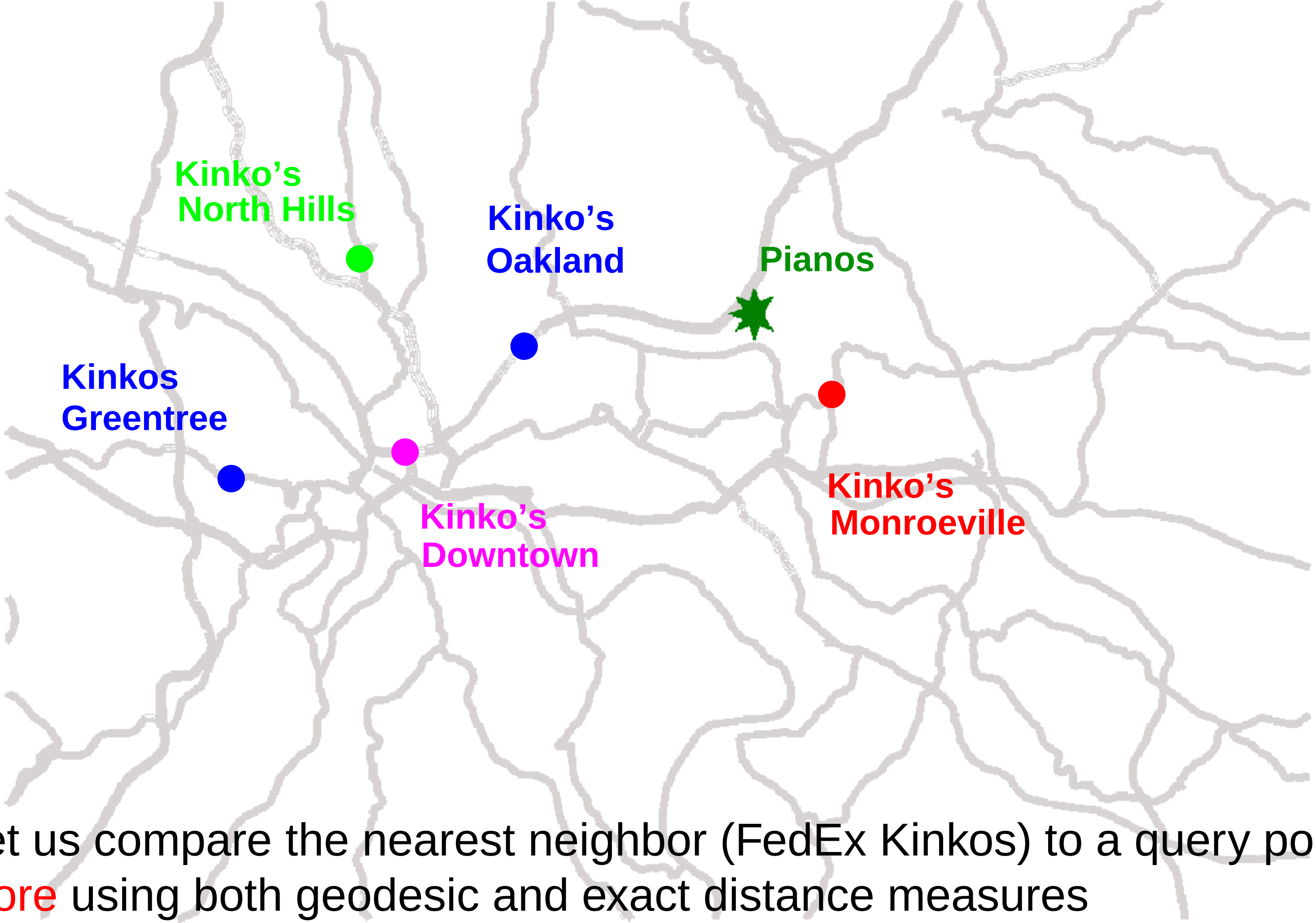
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D

Application – Find the closest Kinko's



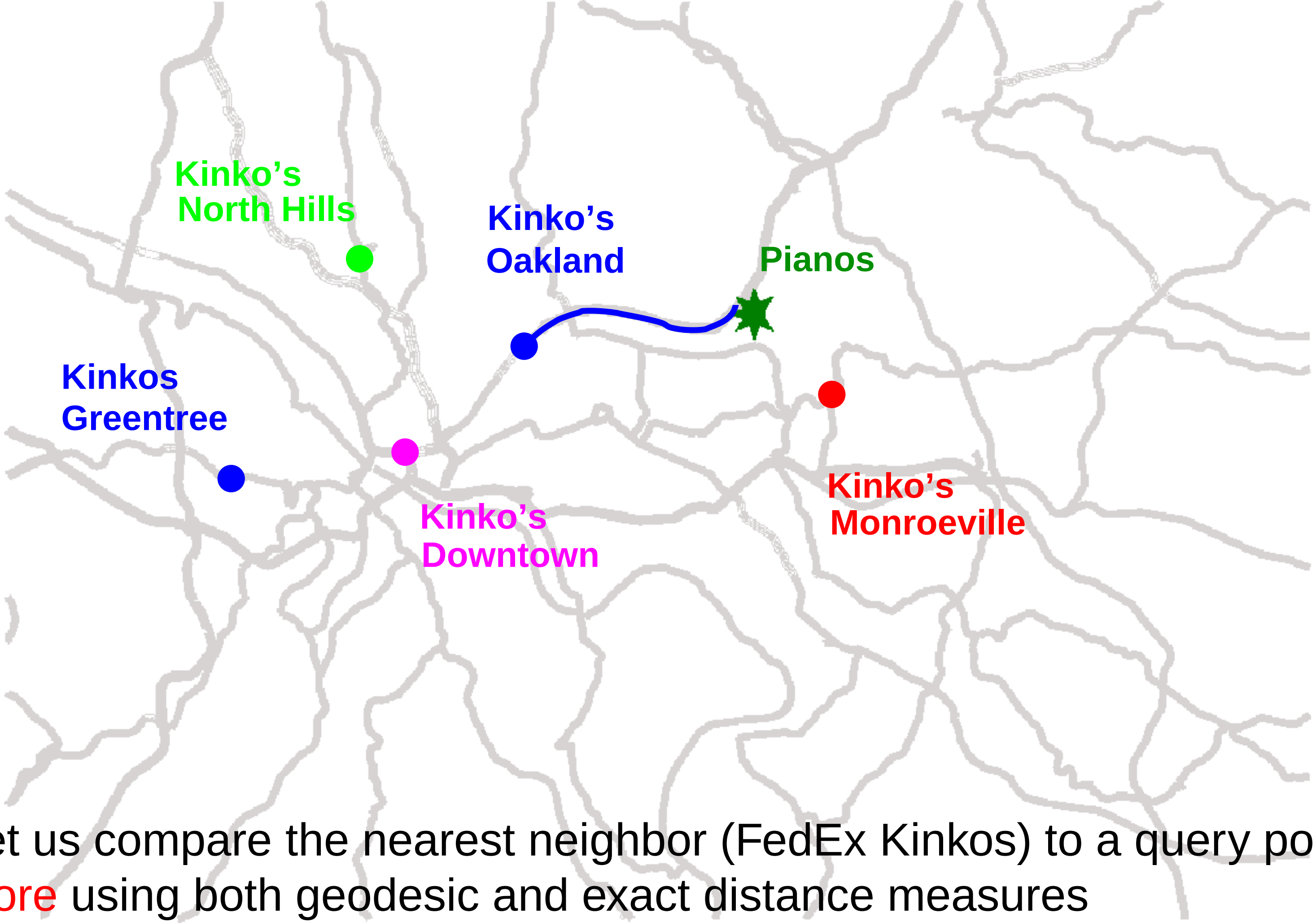
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G

Application – Find the closest Kinko's



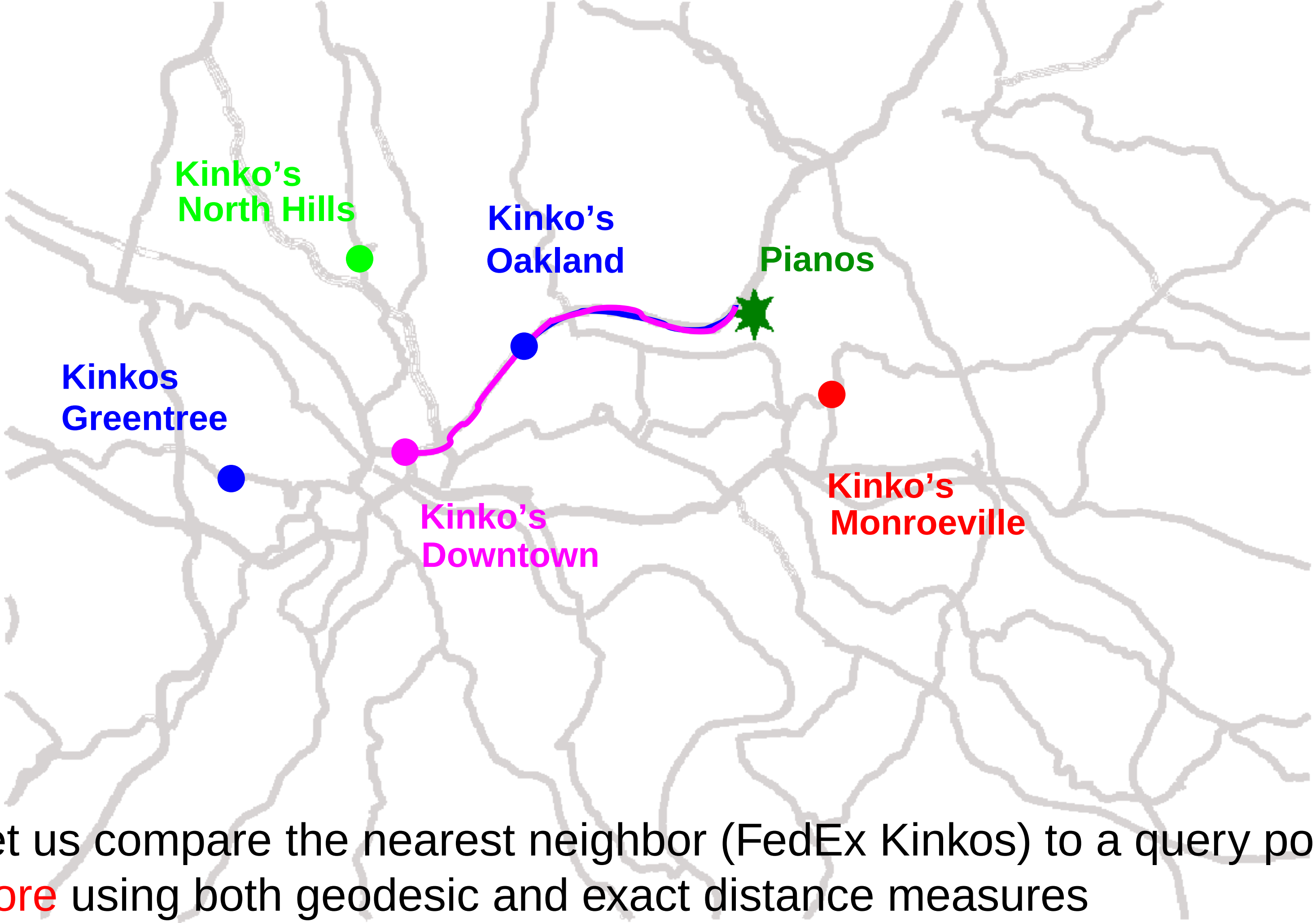
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering

Application – Find the closest Kinko's



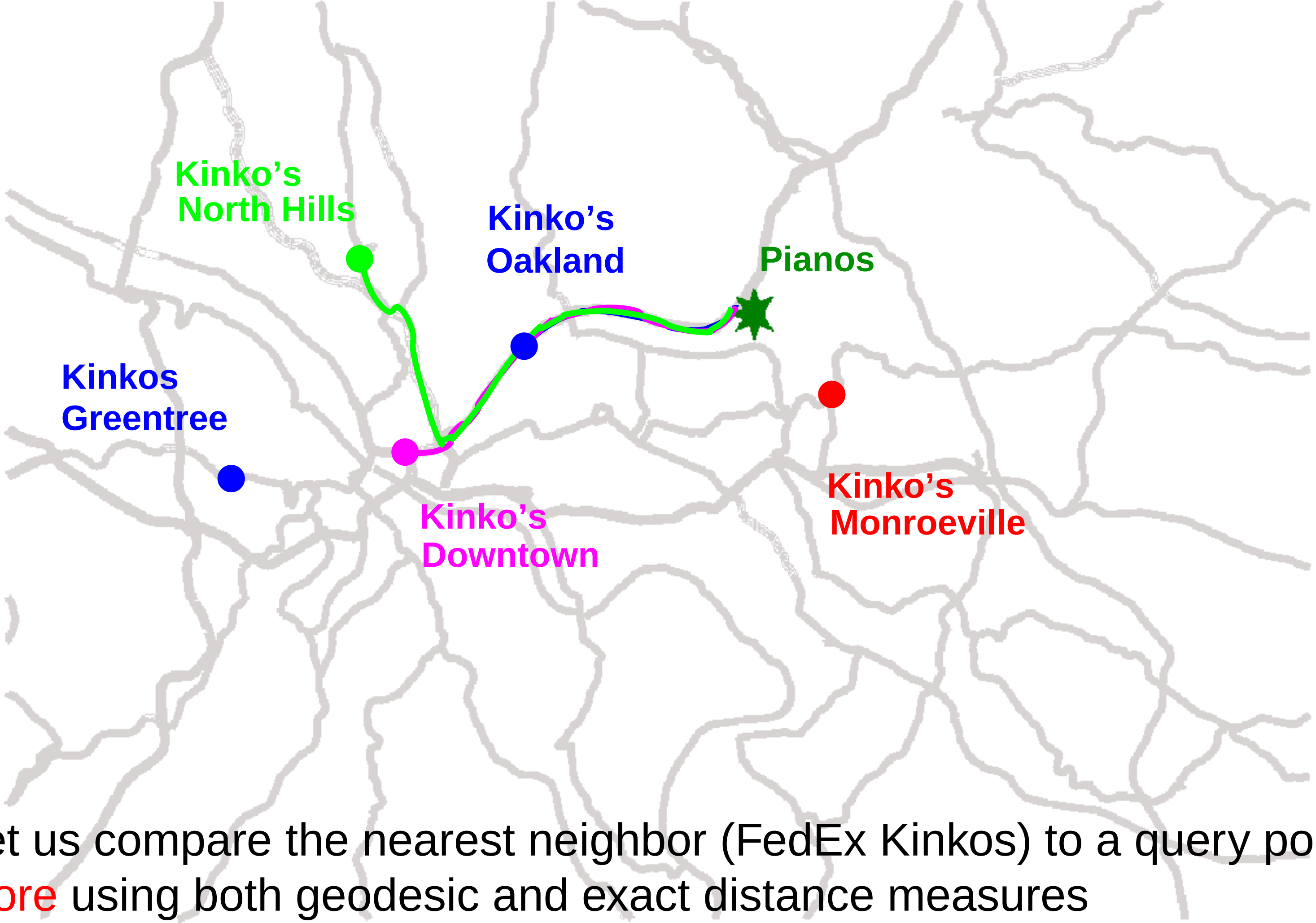
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O

Application – Find the closest Kinko's



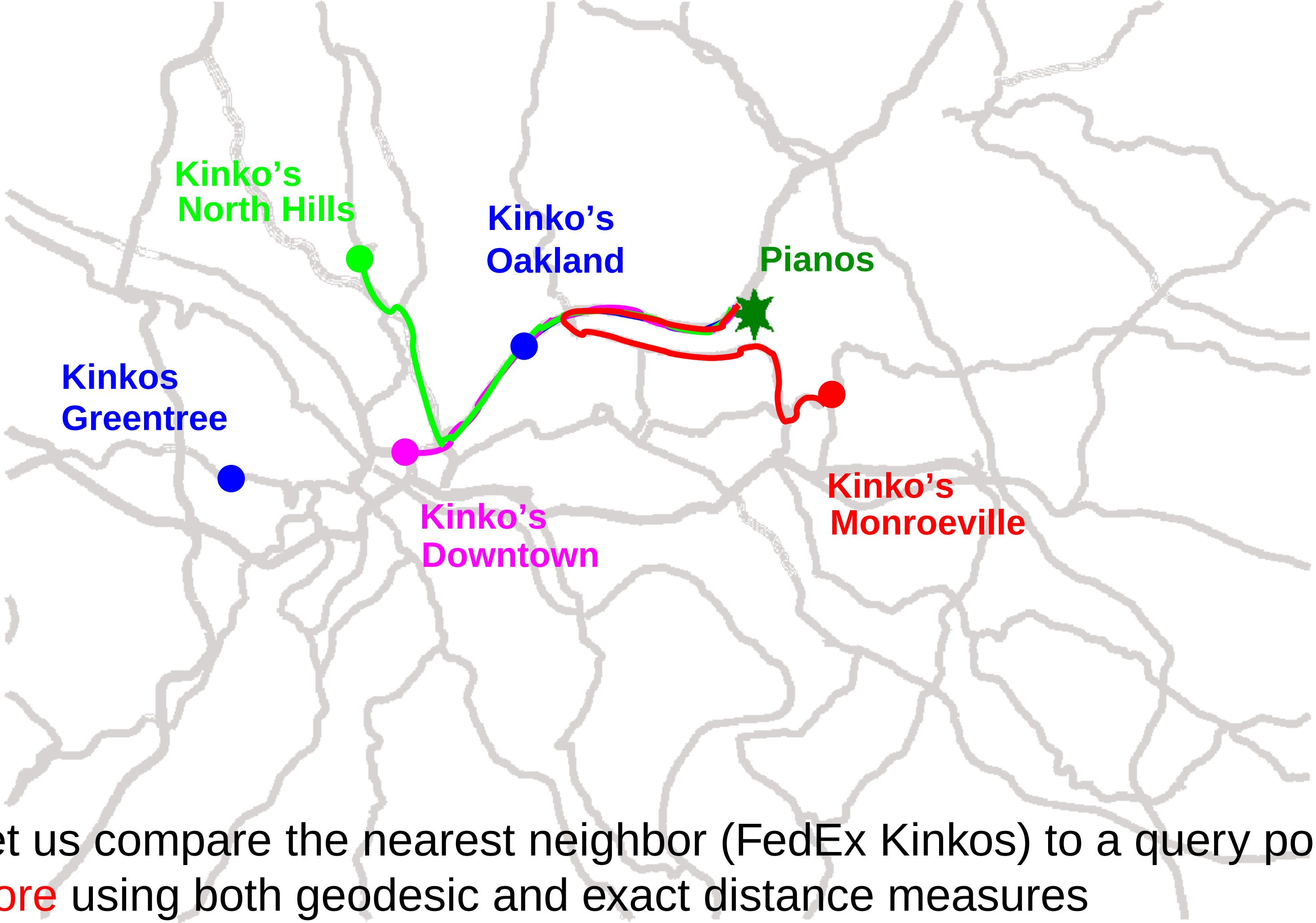
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D

Application – Find the closest Kinko's



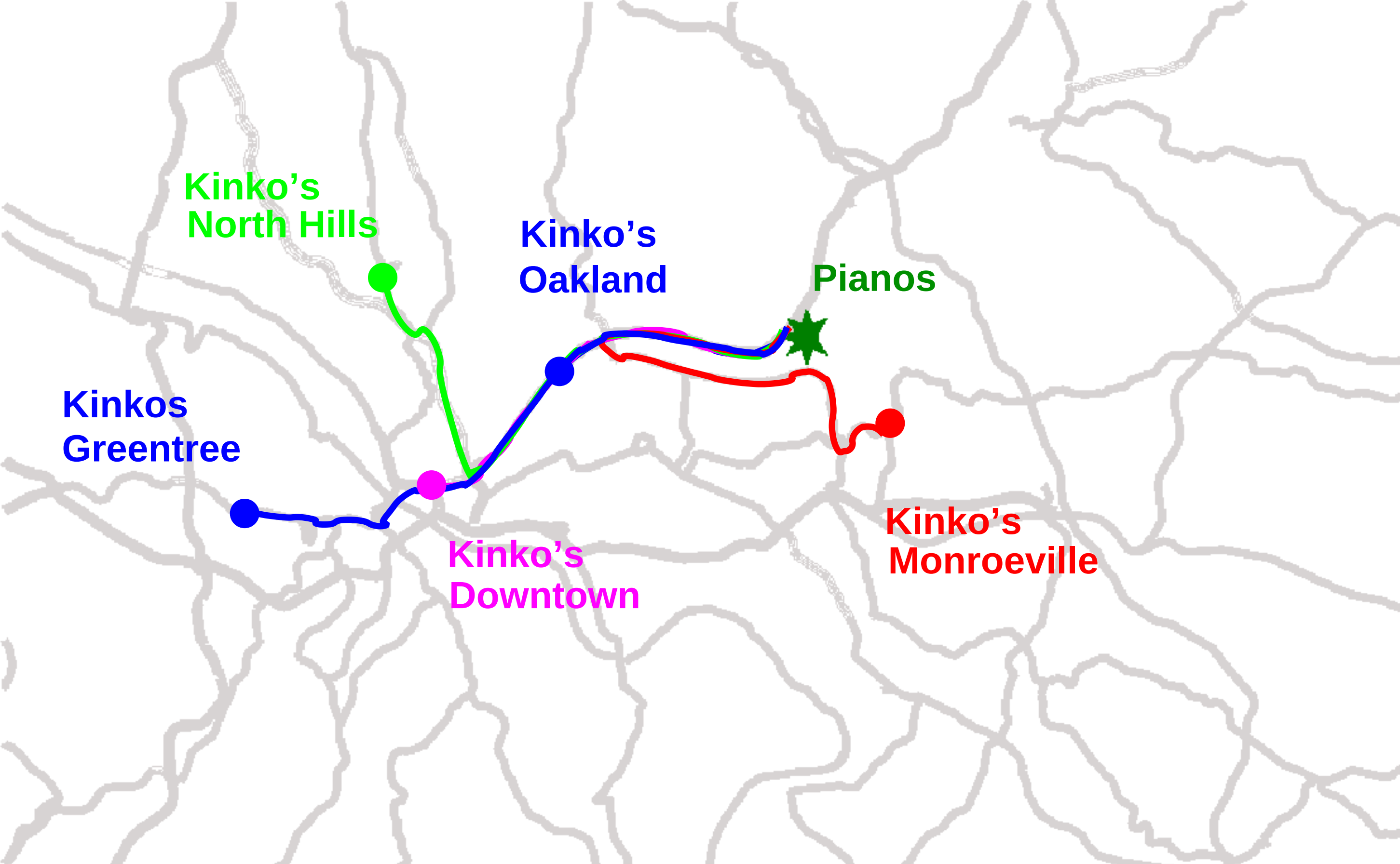
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N

Application – Find the closest Kinko's



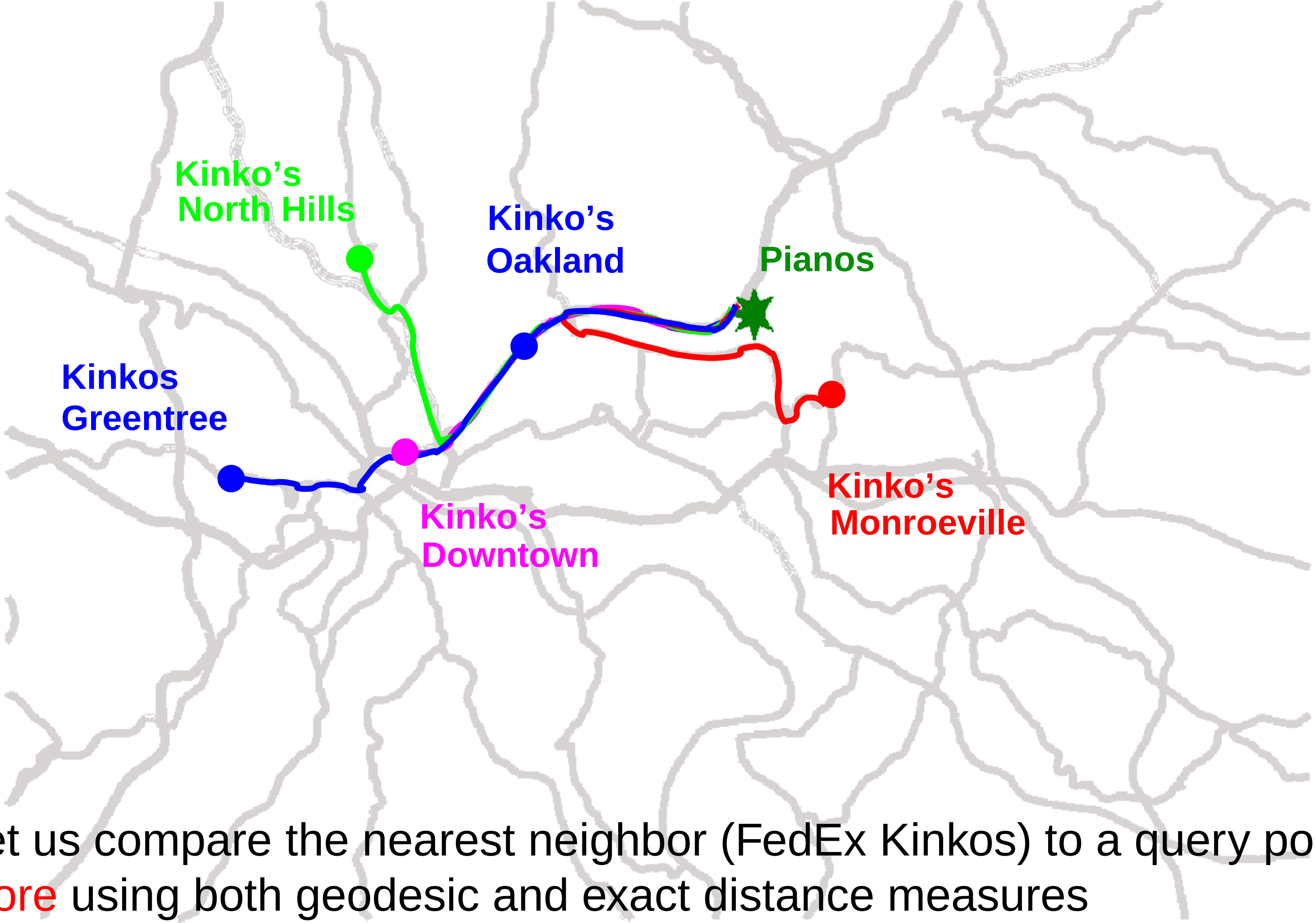
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M

Application – Find the closest Kinko’s



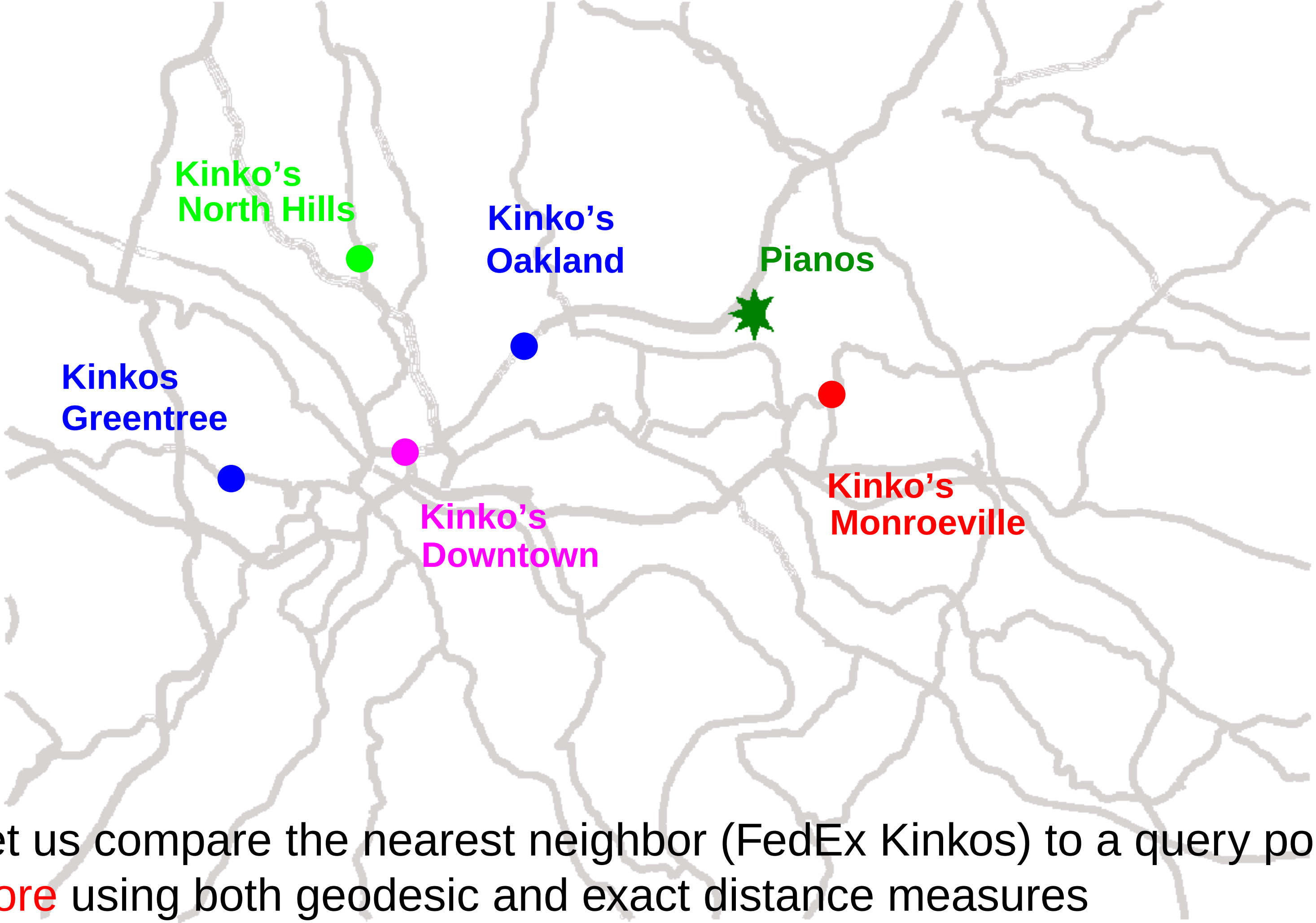
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G

Application – Find the closest Kinko's



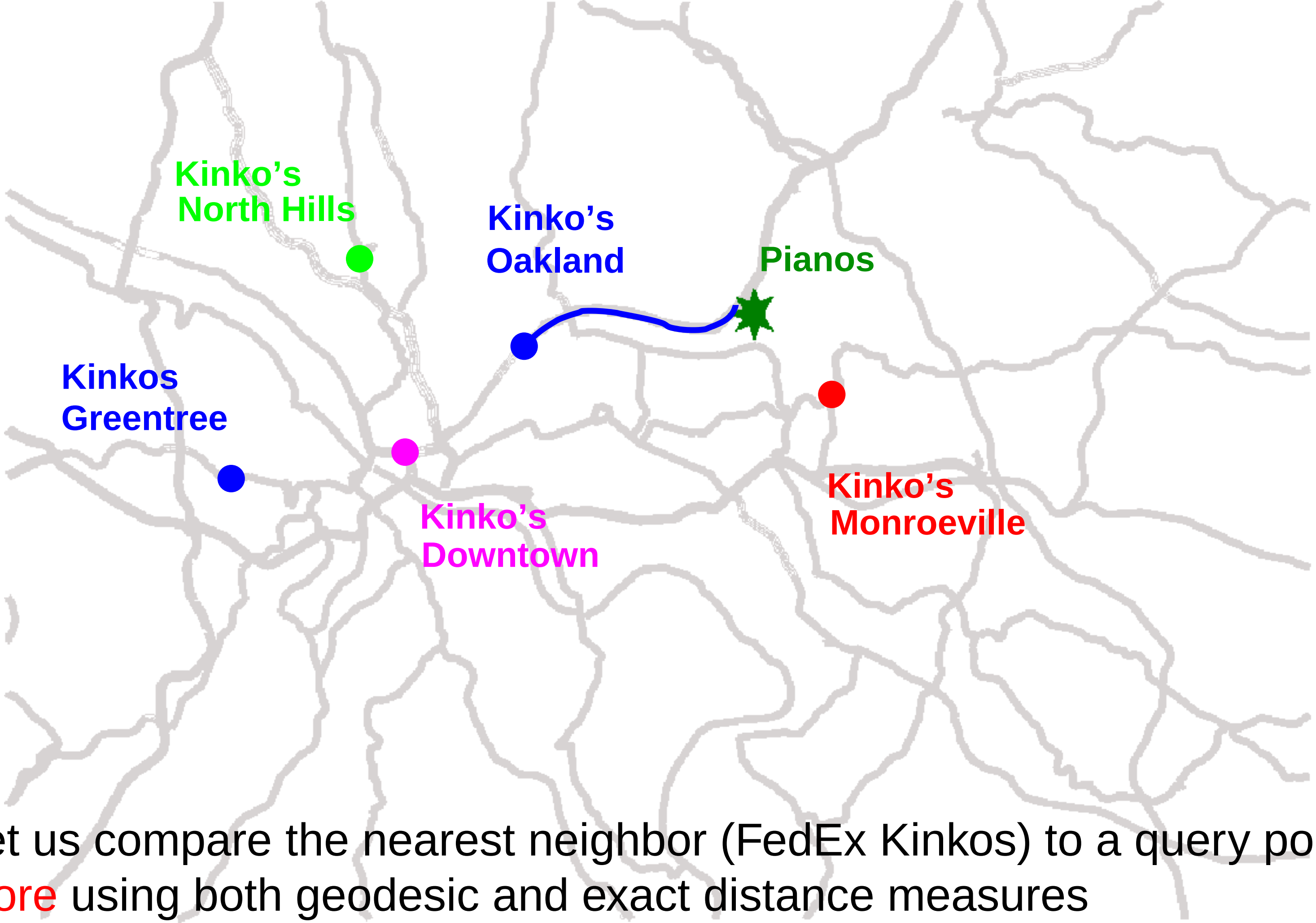
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)

Application – Find the closest Kinko's



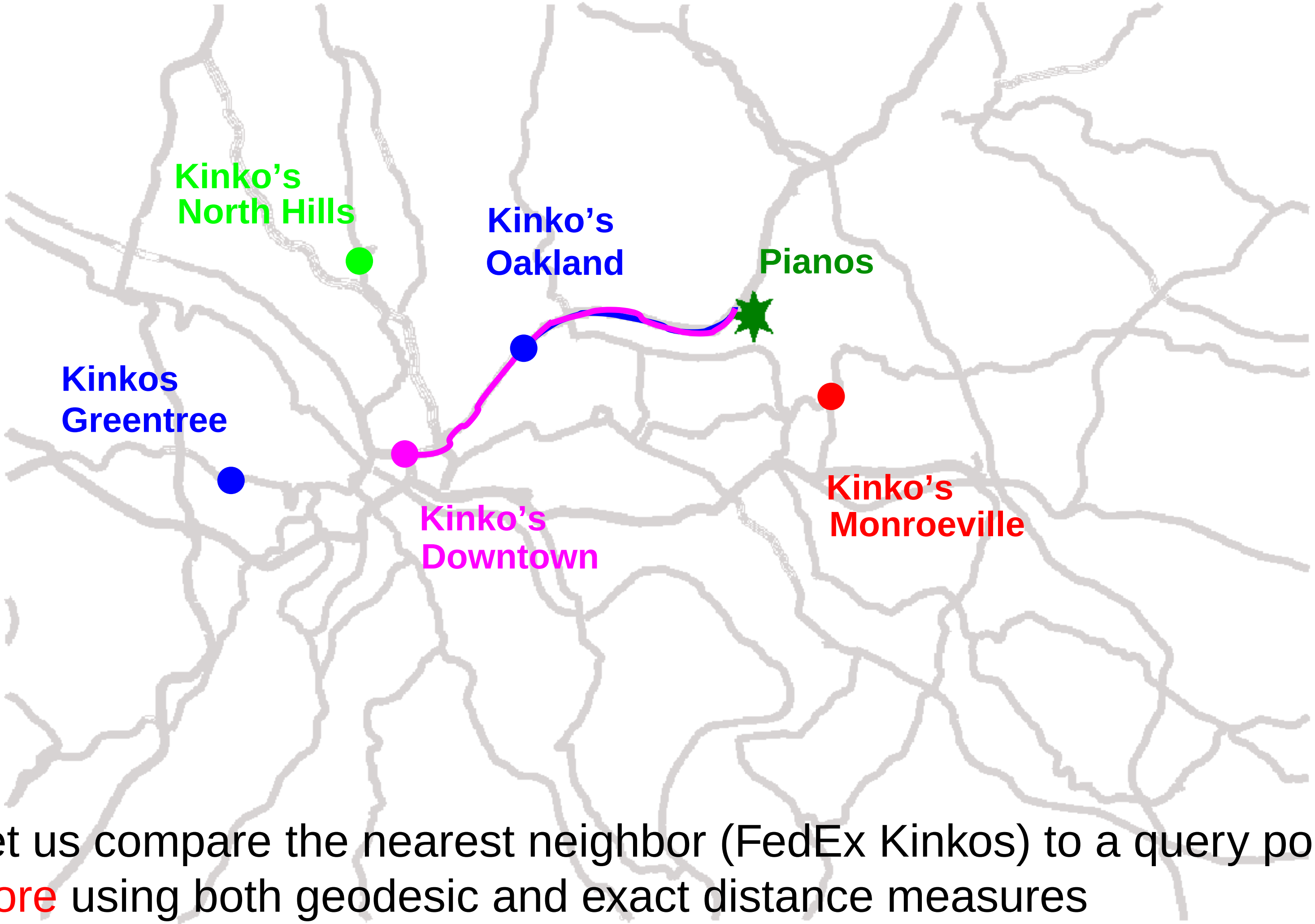
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering

Application – Find the closest Kinko's



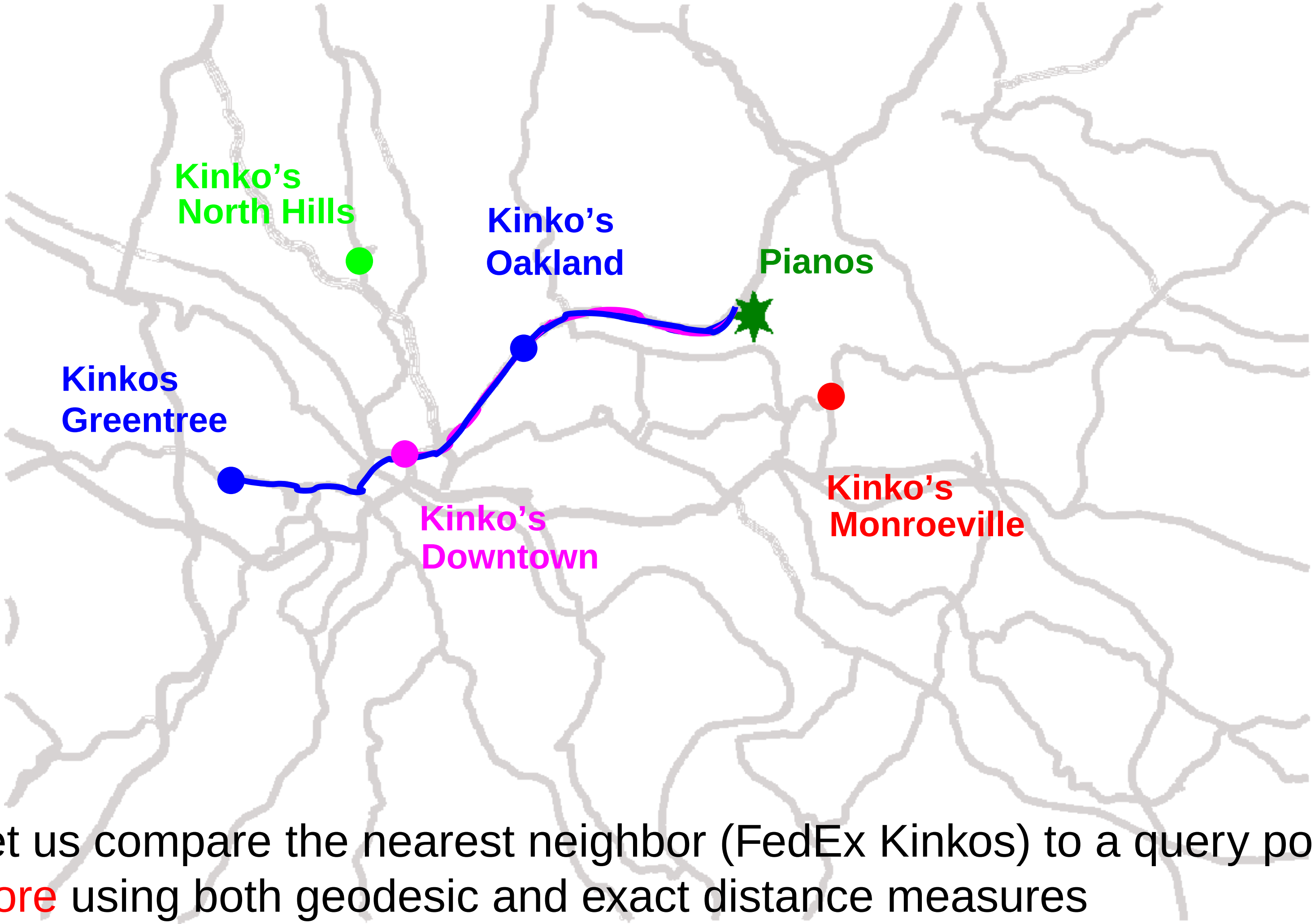
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O

Application – Find the closest Kinko's



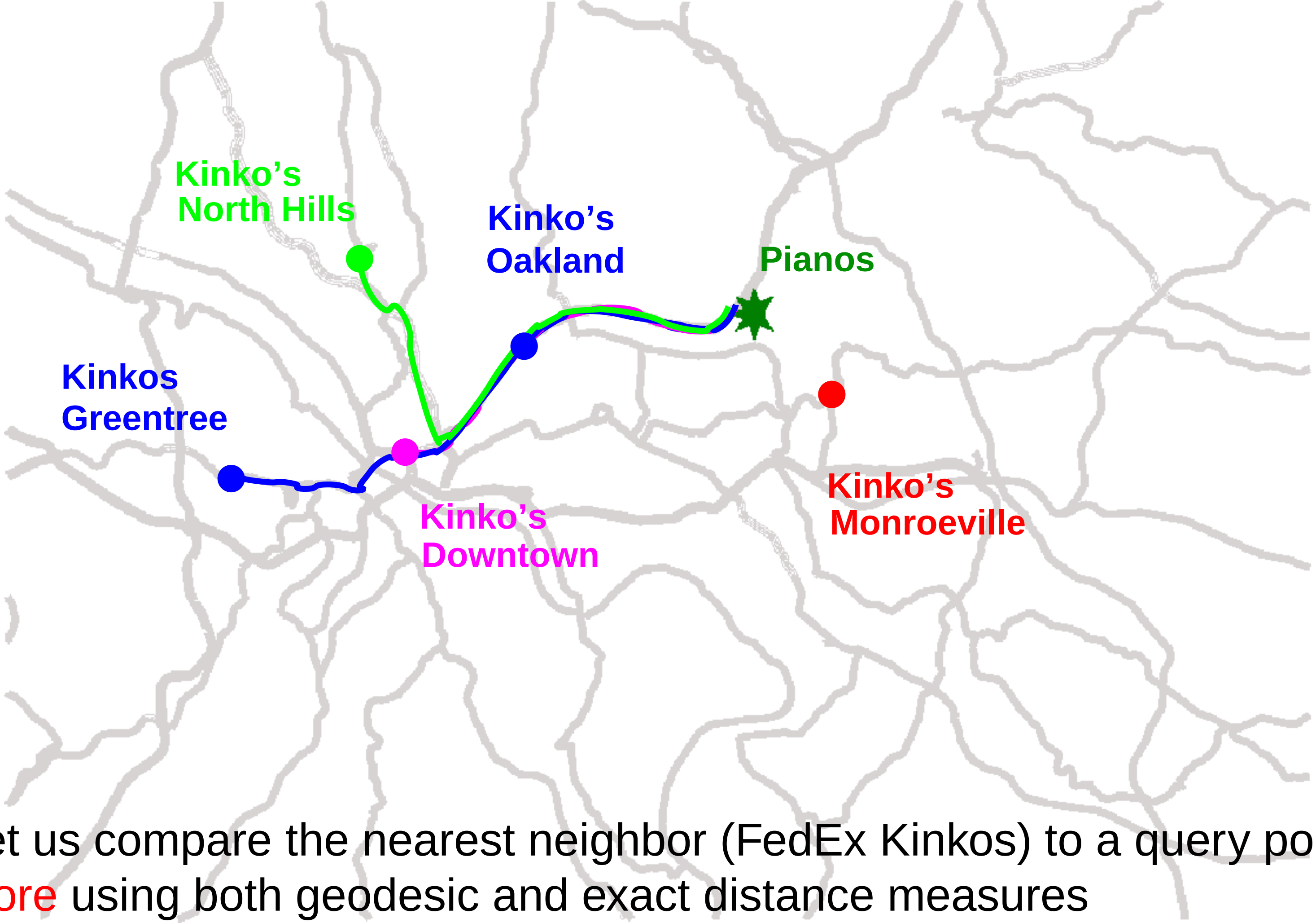
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D

Application – Find the closest Kinko's



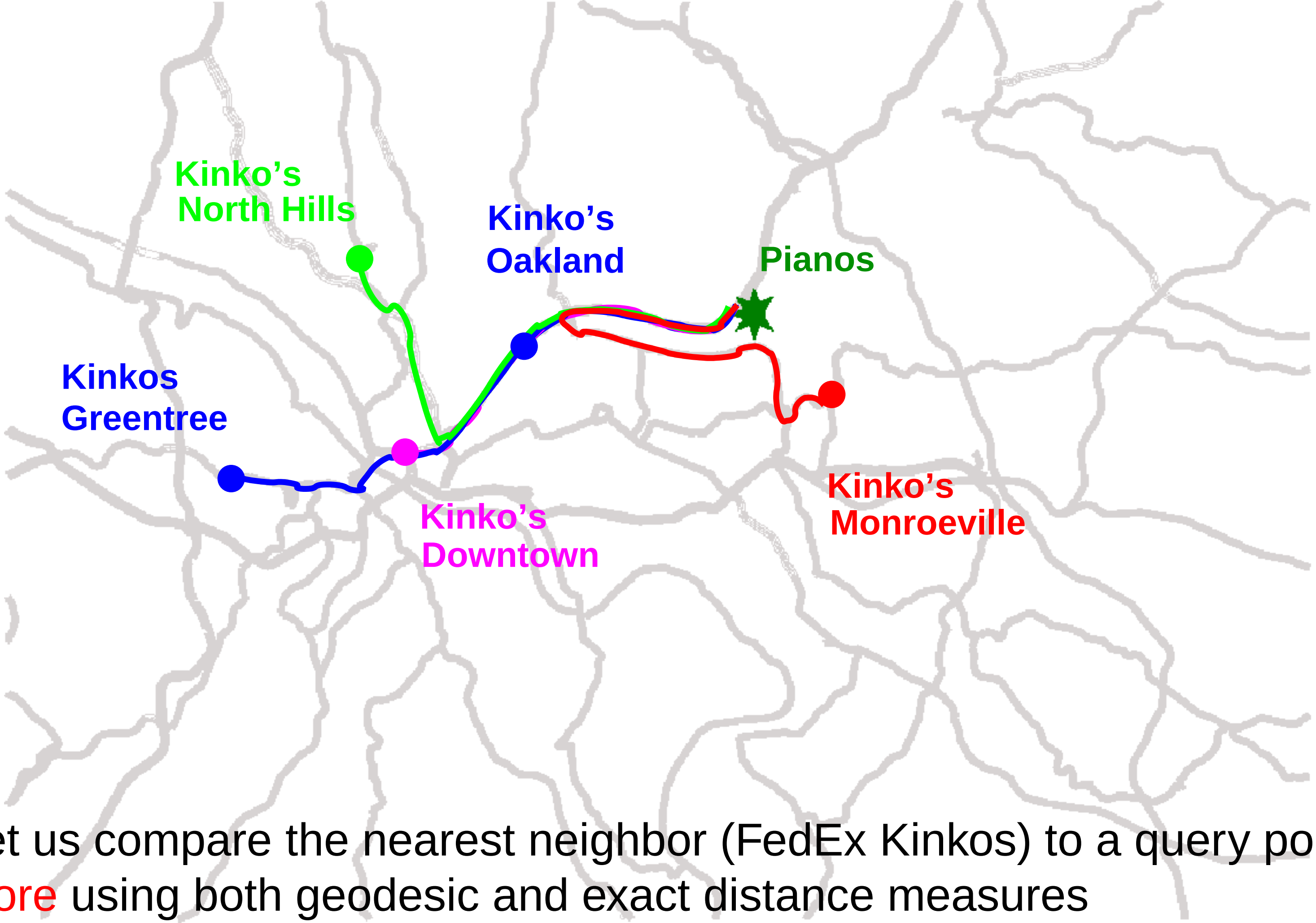
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G

Application – Find the closest Kinko's



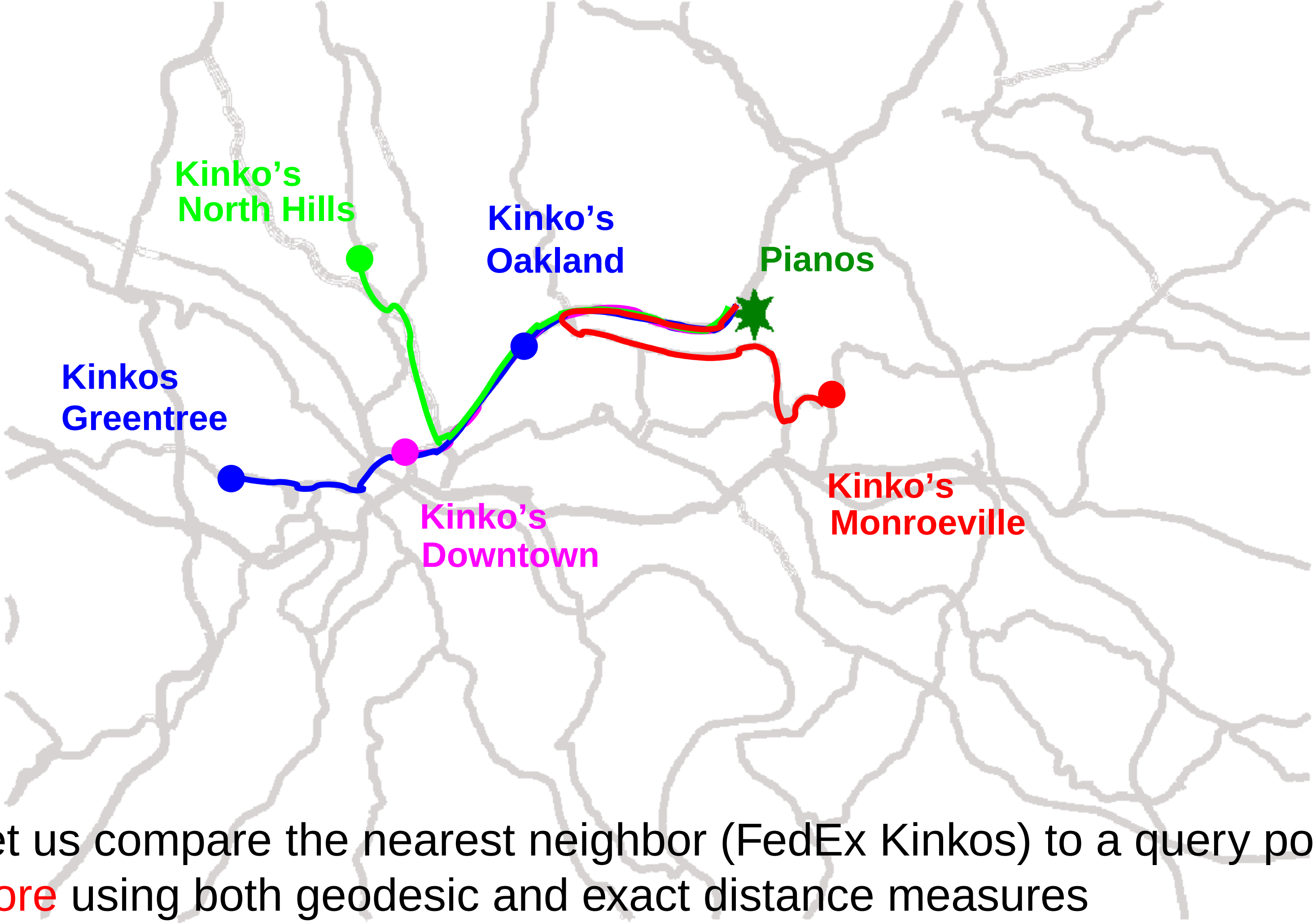
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N

Application – Find the closest Kinko's



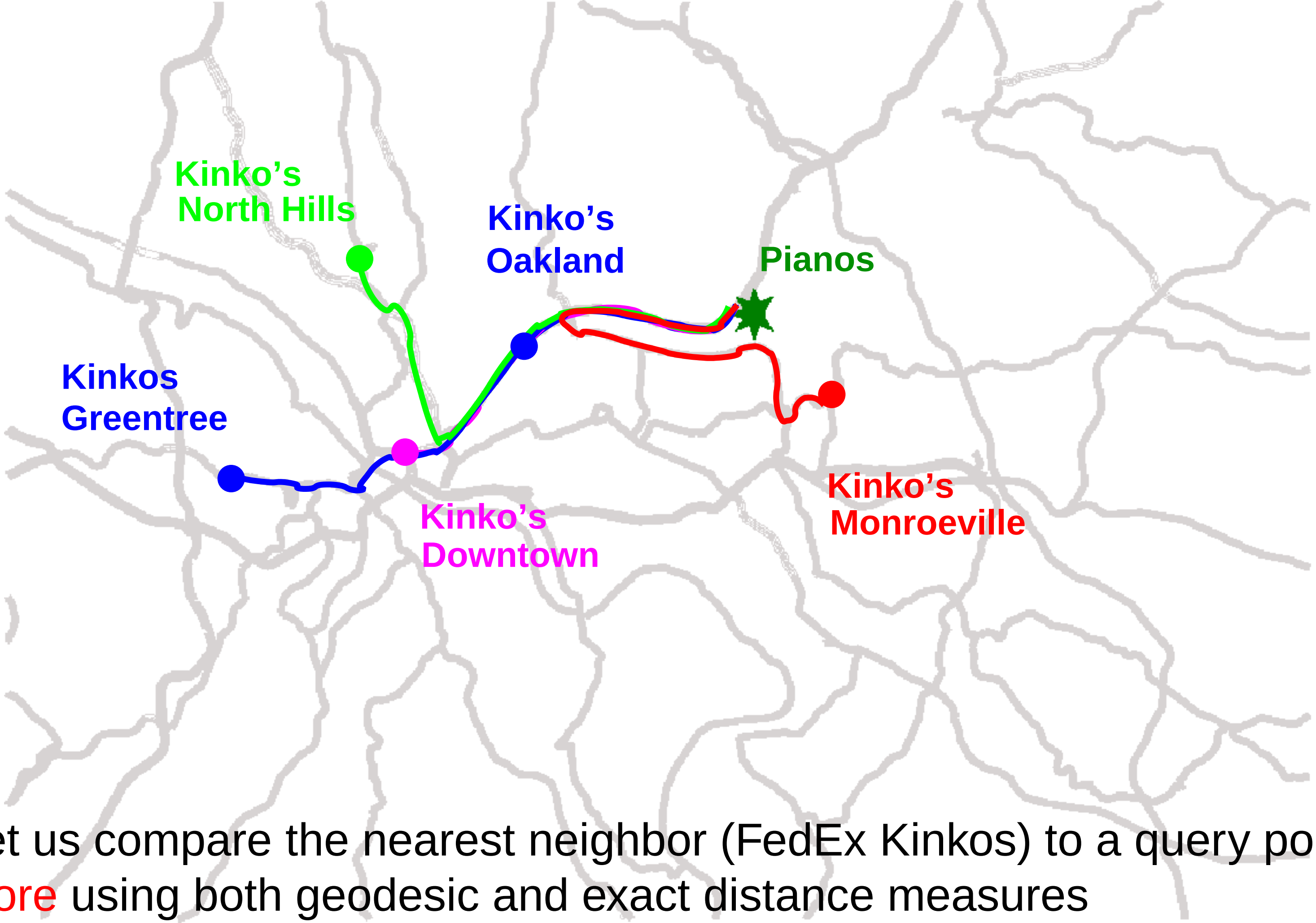
- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M

Application – Find the closest Kinko's



- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M (Error: +32 minutes)

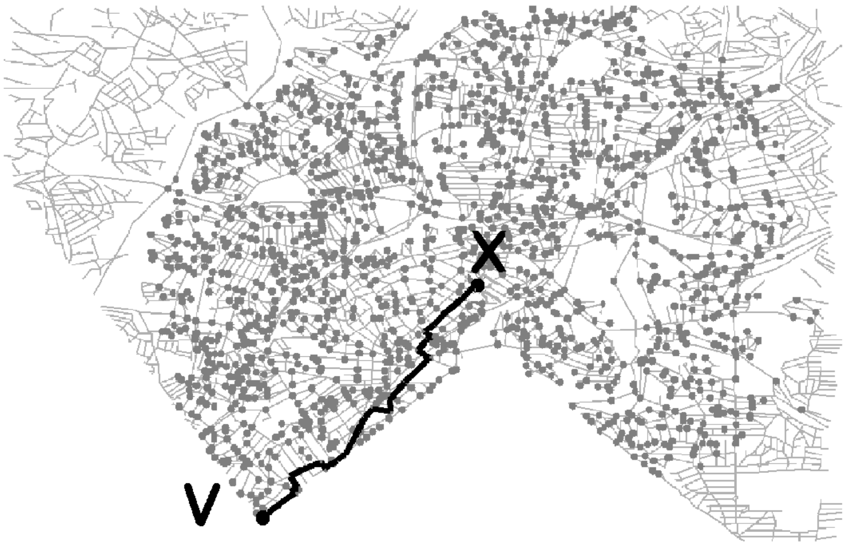
Application – Find the closest Kinko's



- Let us compare the nearest neighbor (FedEx Kinkos) to a query point Piano store using both geodesic and exact distance measures
  - geodesic ordering M O N D G
  - network distance ordering O D N M G (Error: +26 miles)
  - trafficability ordering O D G N M (Error: +32 minutes)
- Challenge: Real time + exact queries

Shortest Path Computation

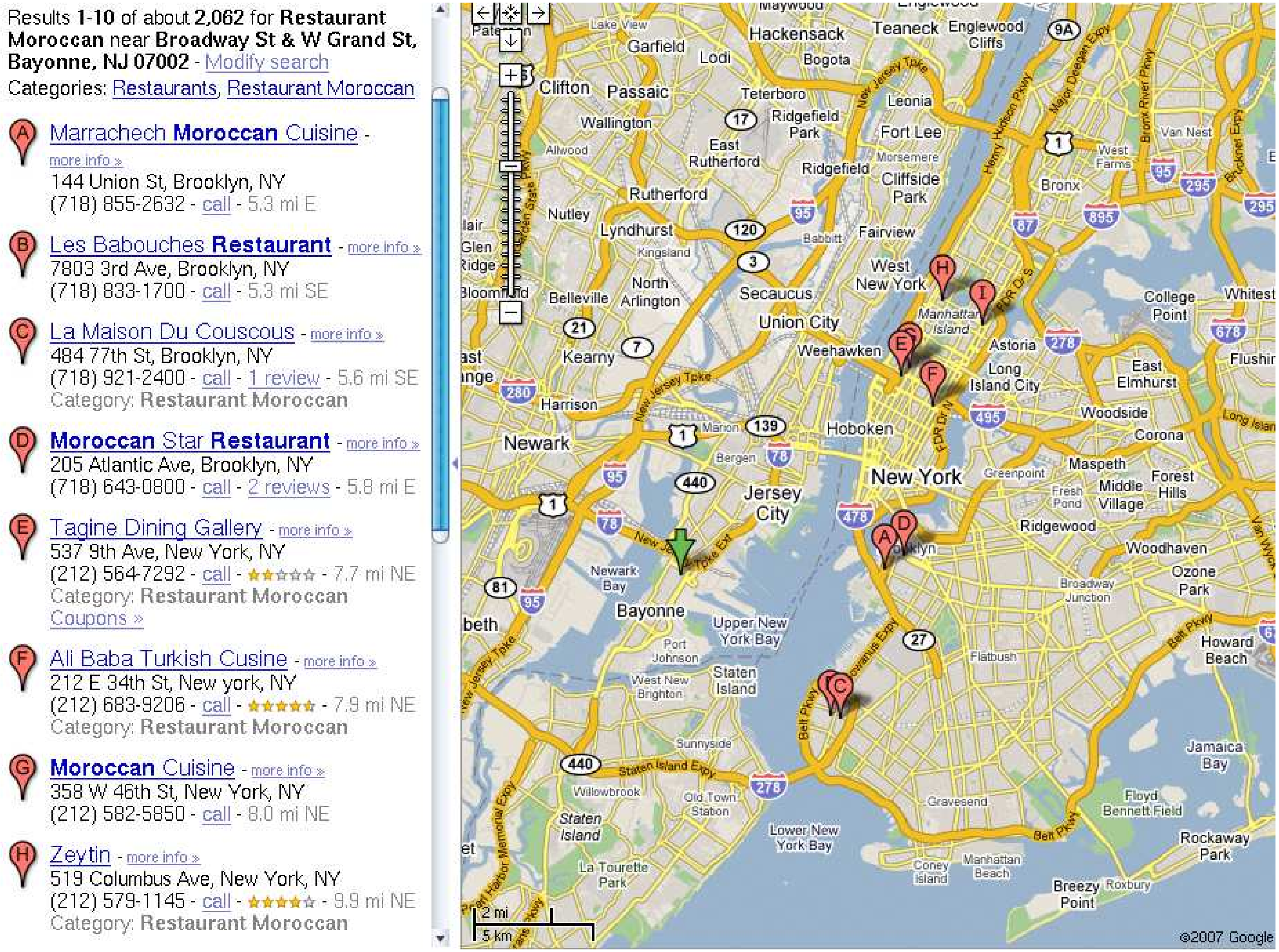
- Shortest path computation is a primitive operation
- Usually use Dijkstra's shortest path algorithm
  - Not feasible in real time for large spatial networks
  - Algorithm visits too many vertices during the search process
  - Ex: Dijkstra's algorithm visits 3191 out of a total of 4233 vertices in the spatial network to identify a path comprising 75 vertices between X and V



- Popular solution: Use "crow flying" (geodesic) distance

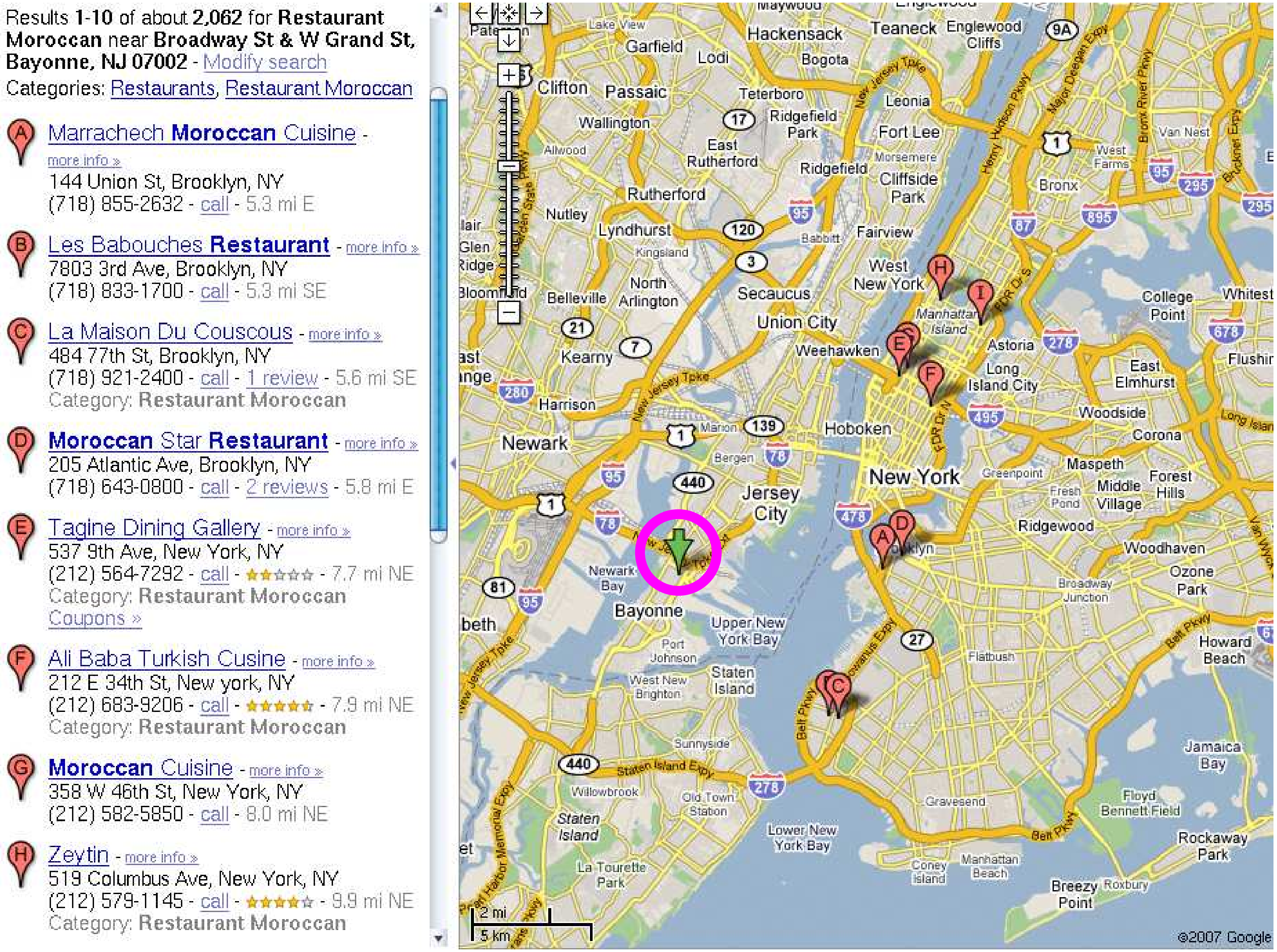
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)

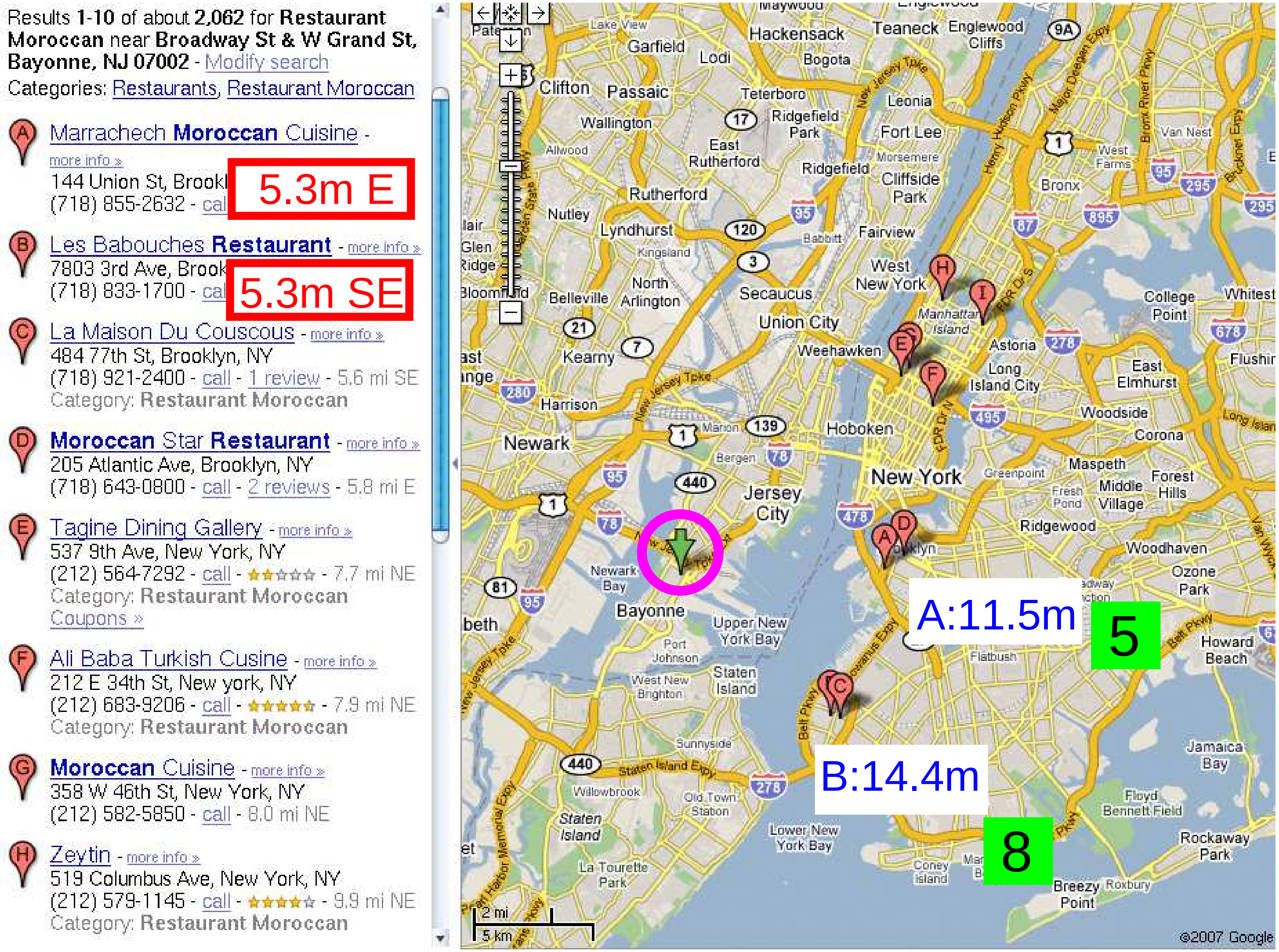


Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)

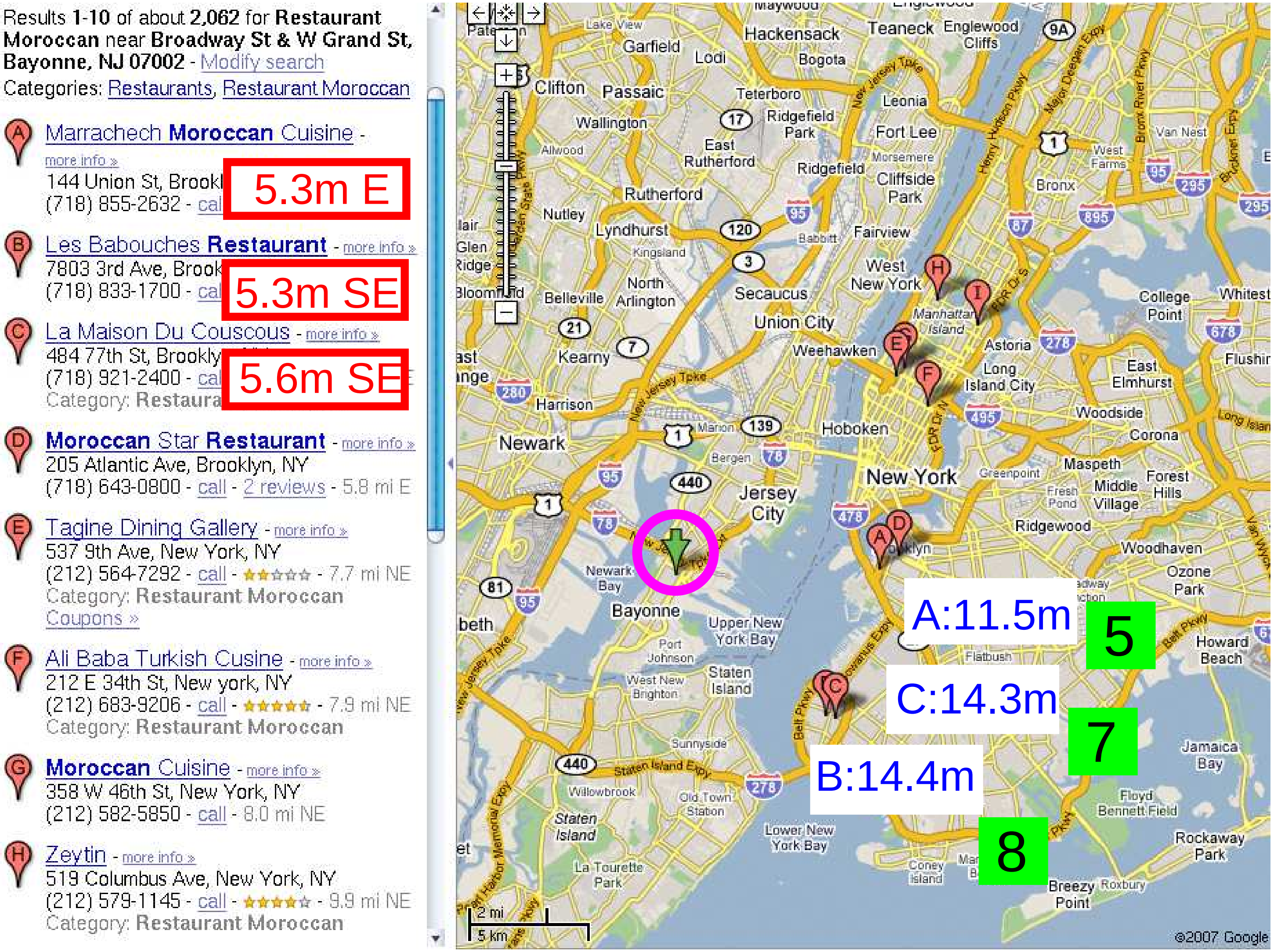
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



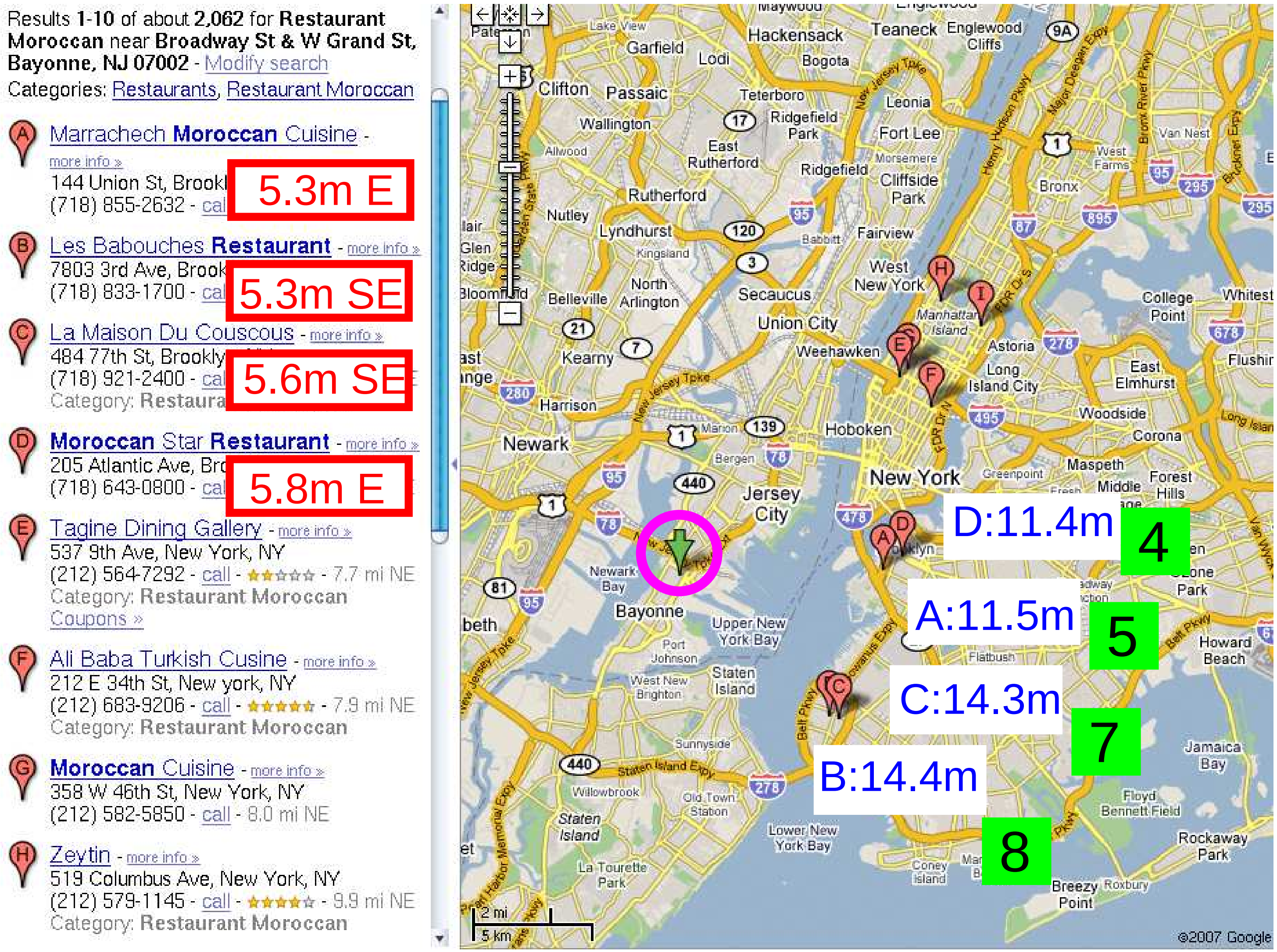
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



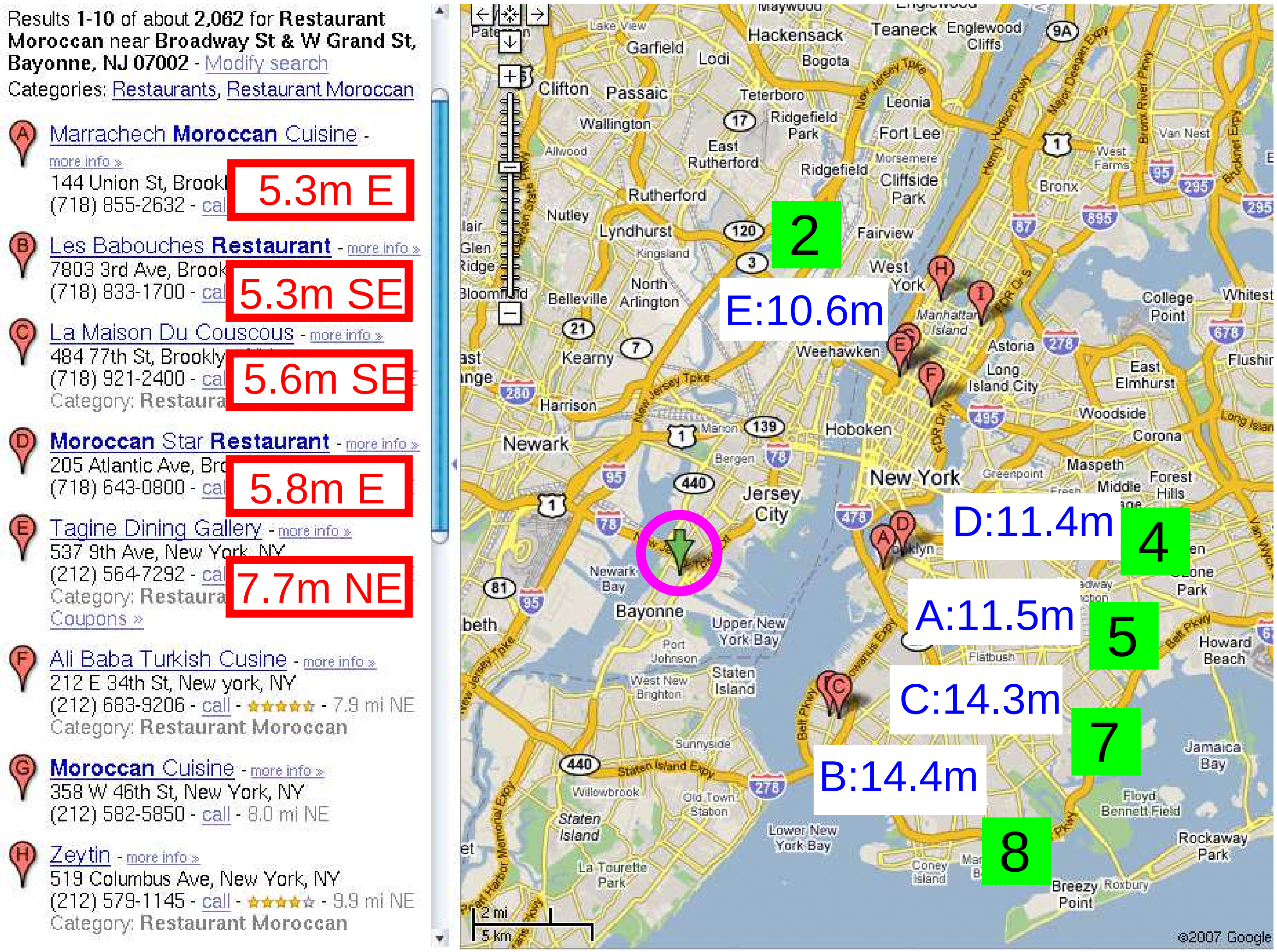
Proximity Search on "Google Local"

- Let us examine the errors between ordering by the spatial distance ("as the crow flies" used by Google) and by the network distance (used by us)



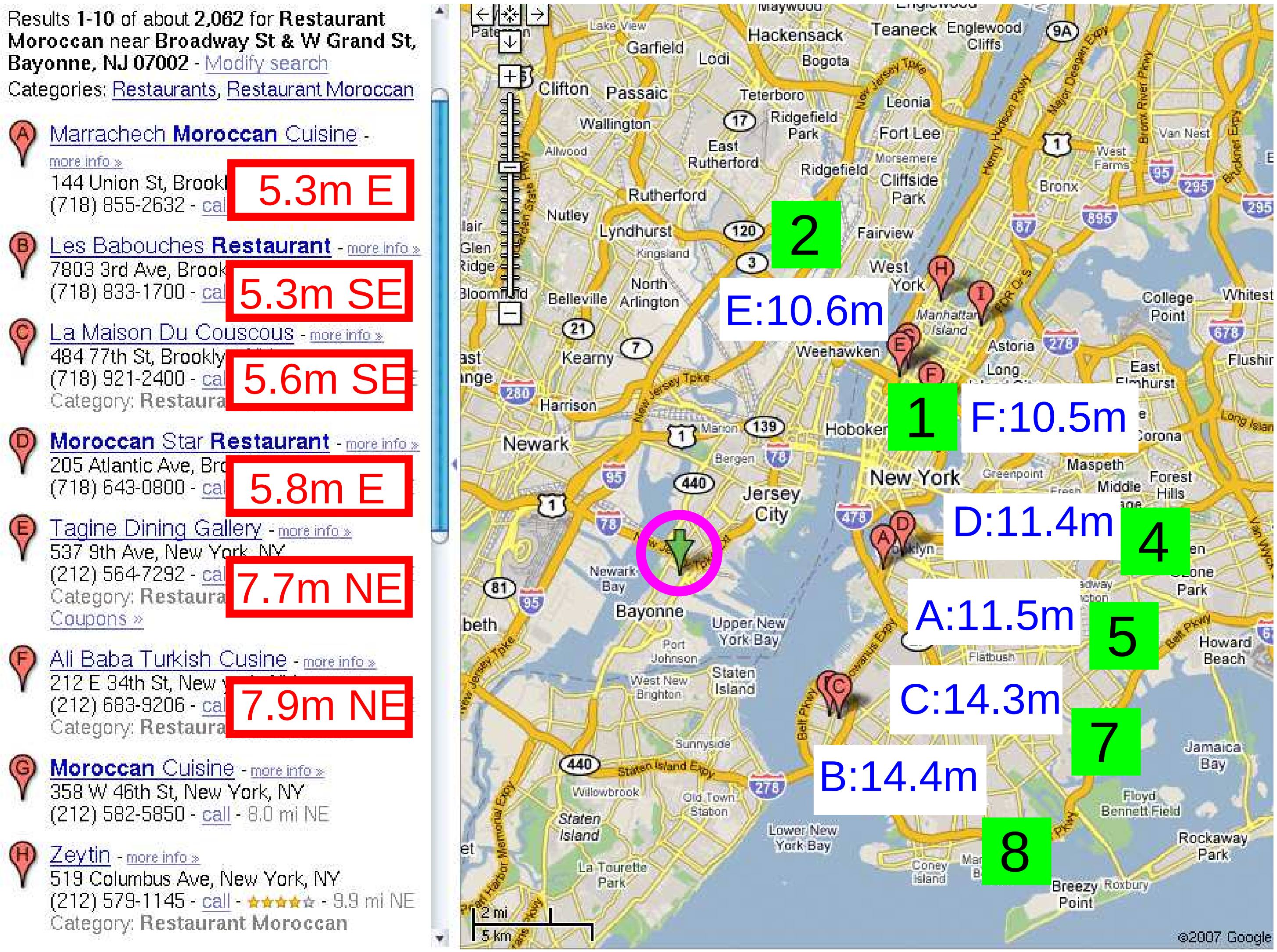
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



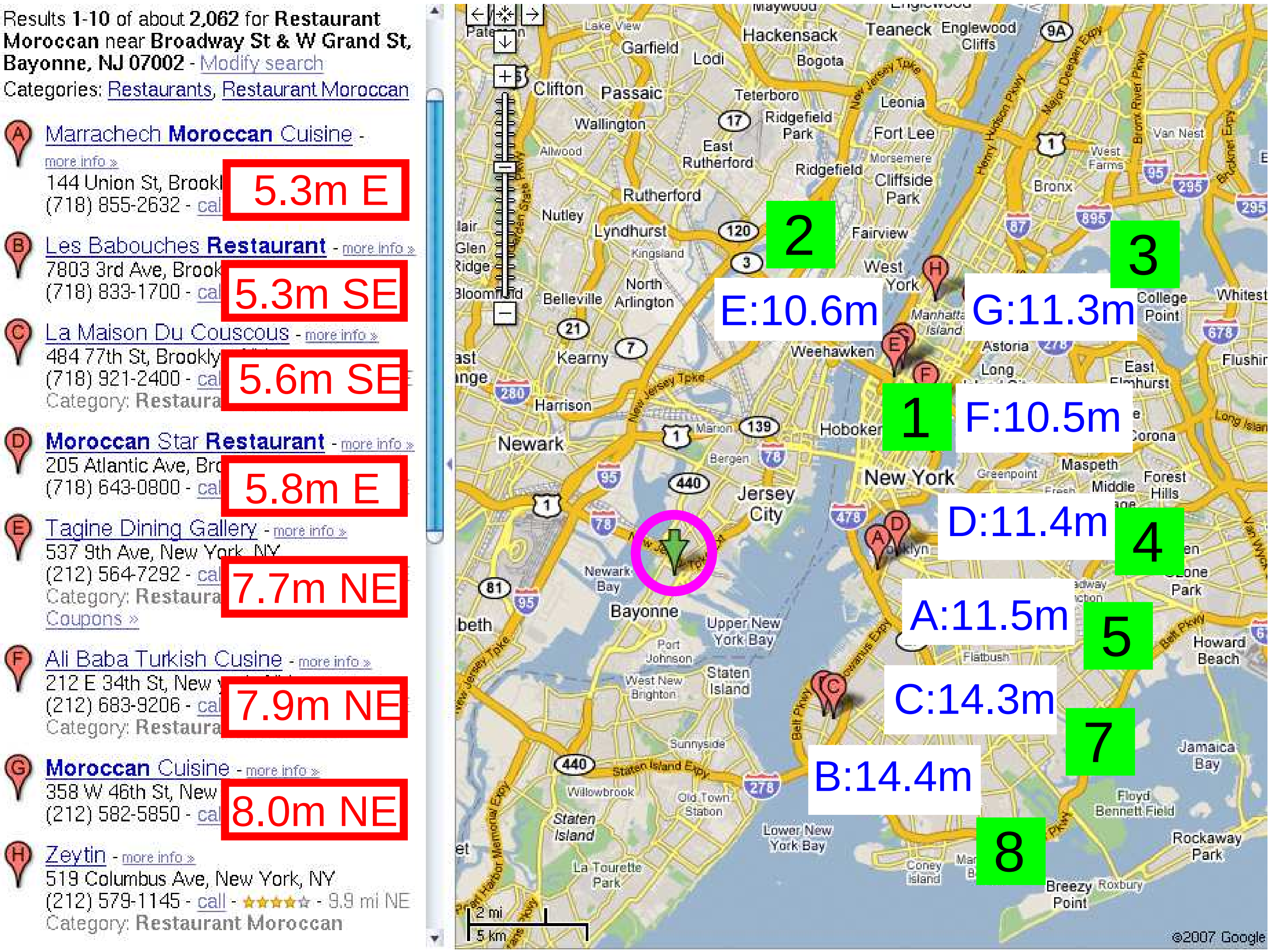
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



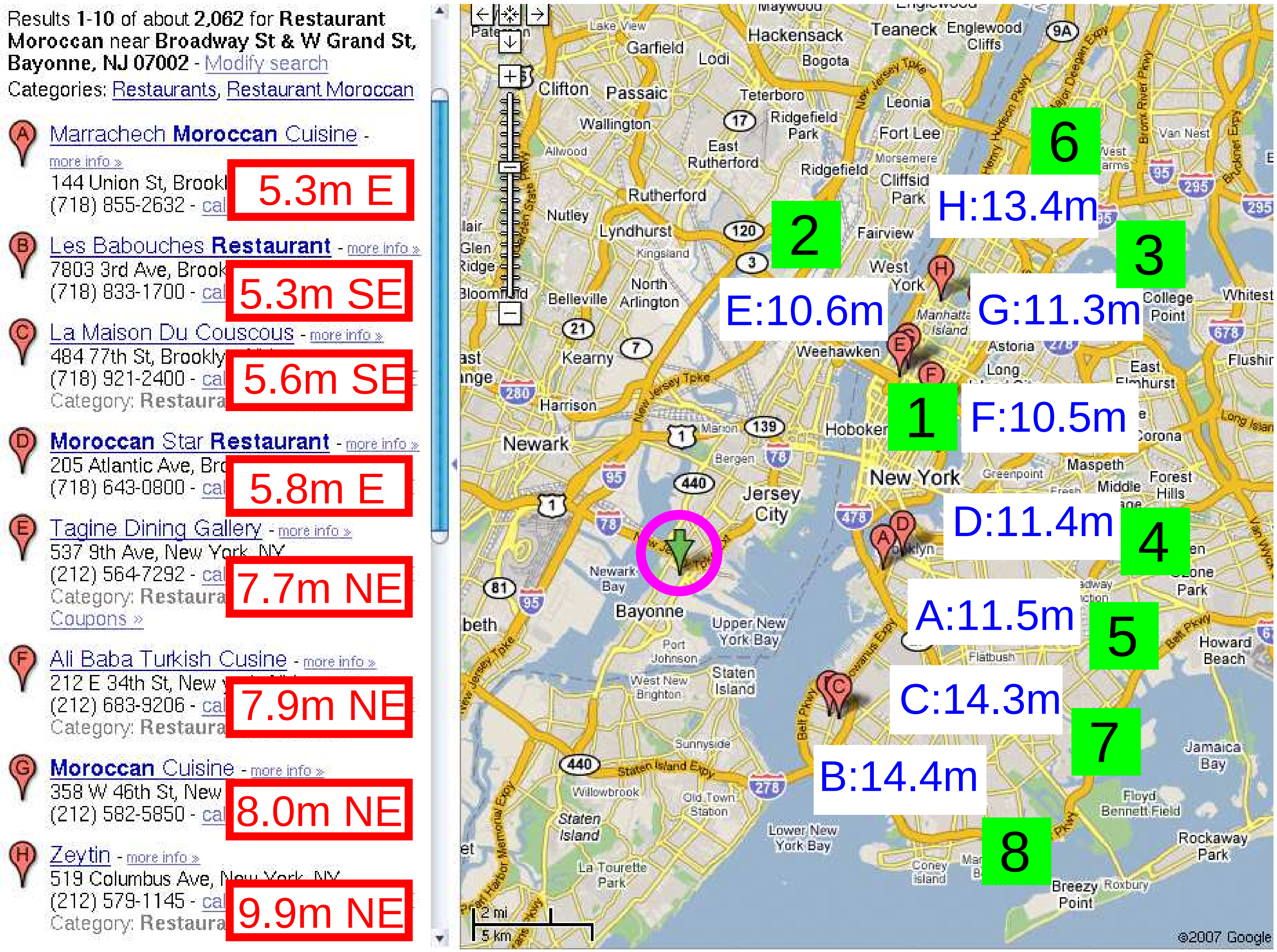
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



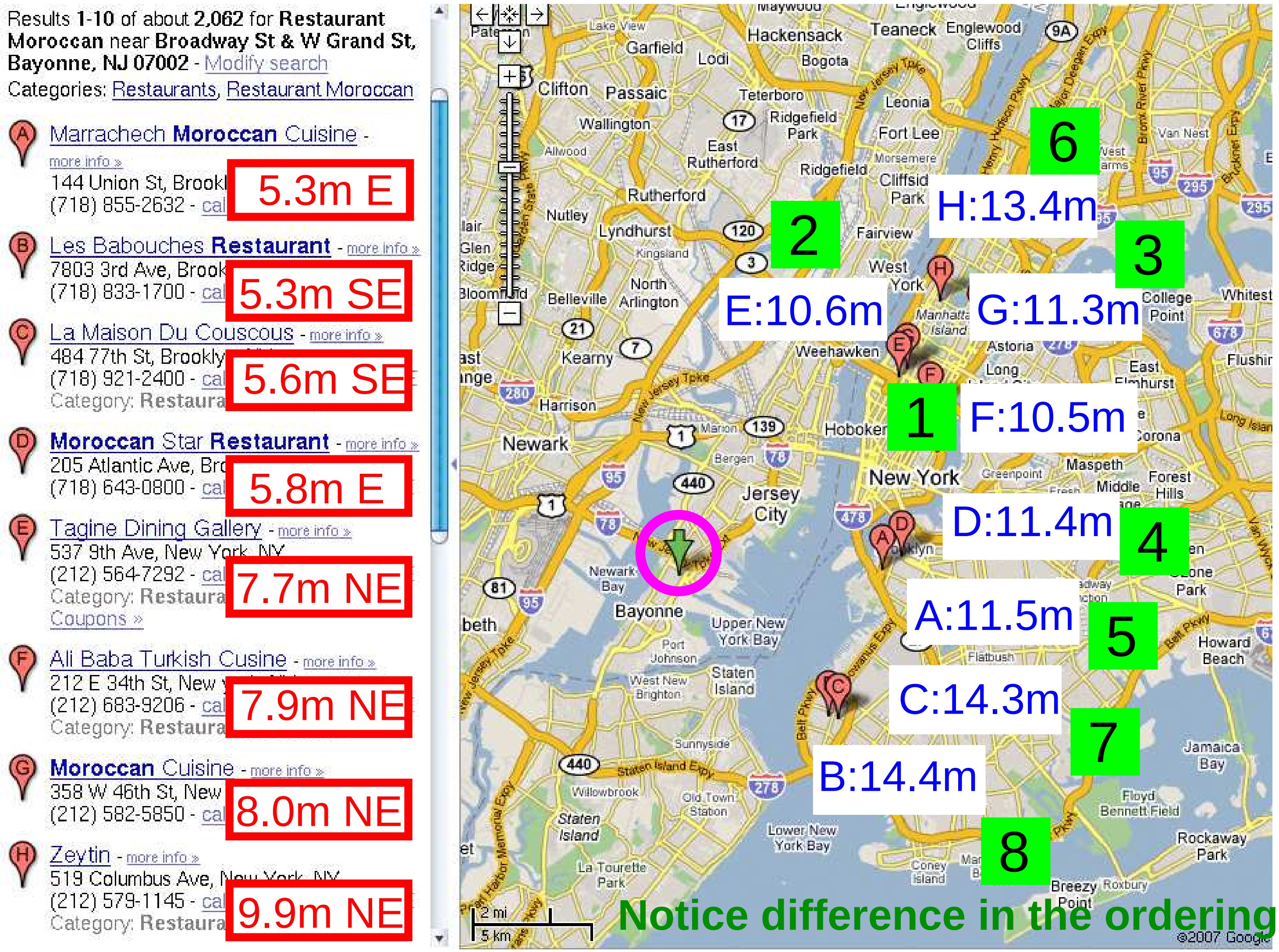
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



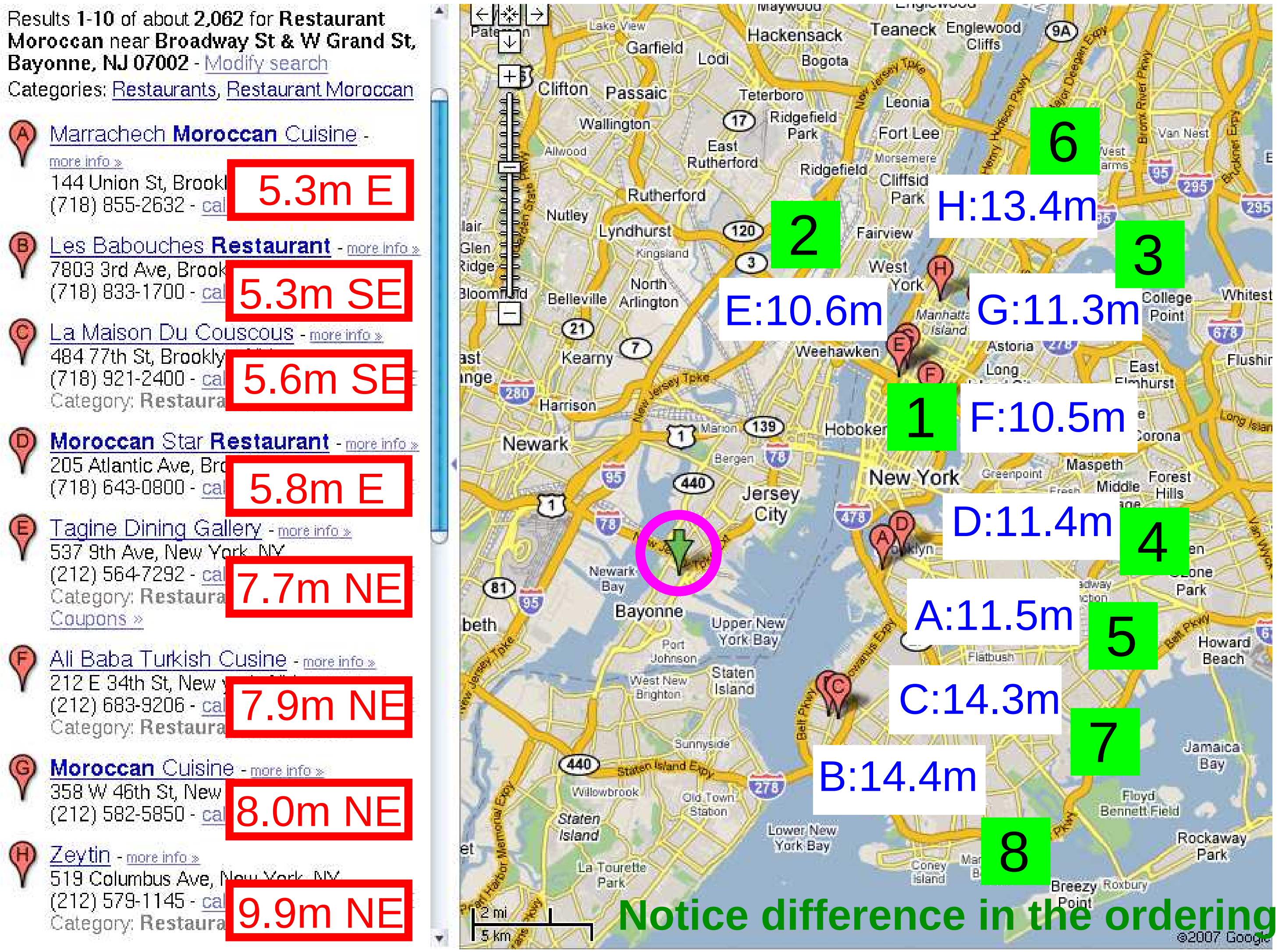
Proximity Search on “Google Local”

- Let us examine the errors between ordering by the spatial distance (“as the crow flies” used by Google) and by the network distance (used by us)



Proximity Search on "Google Local"

- Let us examine the errors between ordering by the spatial distance ("as the crow flies" used by Google) and by the network distance (used by us)



- Goal: Instant answers as well as accurate answers

Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

Precomputation of Shortest Paths

- By precomputing and storing all of the shortest paths, nearest neighbor queries could be answered instantly
  - How to effectively compute the shortest path?
  - How to effectively store the shortest path?
  - Challenge: very large network (approximately 24 million vertices)
- Result: Enables decoupling nearest neighbor and shortest path computation processes
  - Decouples domain S of query objects and objects from which the neighbors are drawn from domain V of the vertices of the spatial network
  - Implies no need to recompute shortest paths anew each time there are changes in q or S

Strategy: Precomputation

- Idea: Precompute and store all pairs shortest paths
  1. How to compute shortest paths?
  2. How to store shortest paths?
     - Challenge: Very large network (24,000,000 vertices)
- Trade-off: Space requirements vs. retrieval time

k=shortest path length   n=# vertices   m=# edges   constants $\delta > 1$ and $\epsilon > 0$

Approach	Space	Query Time	
		Path	Distance
Explicit Path Storage	$O(n^3)$	$O(1)$	$O(1)$
Next-Hop Storage	$O(n^2)$	$O(k)$	$O(1)$
Dijkstra's Algorithm	$O(m+n)$	$O(m+n\log n)$	$O(m+n\log n)$
SILC	$O(n\sqrt{n})$	$O(k\log n)$	Approx: $O(\log n)$
Distance Oracle–1	$O((\frac{1}{\epsilon})^2 n)$	—	ϵ-Approx:$O(\log n)$
Distance Oracle–2	$O((\frac{1}{\epsilon})^2 n\log n)$	—	ϵ-Approx:$O(1)$
Path Oracle	$O((2+\frac{1}{(\delta-1)})^2 n)$	$O(k\log n)$	—
Path-Distance Oracle	$O(\max((2+\frac{1}{(\delta-1)})^2, \frac{1}{\epsilon}^2)n)$	$O(k\log n)$	ϵ-Approx:$O(\log n)$

SILC Path Encoding

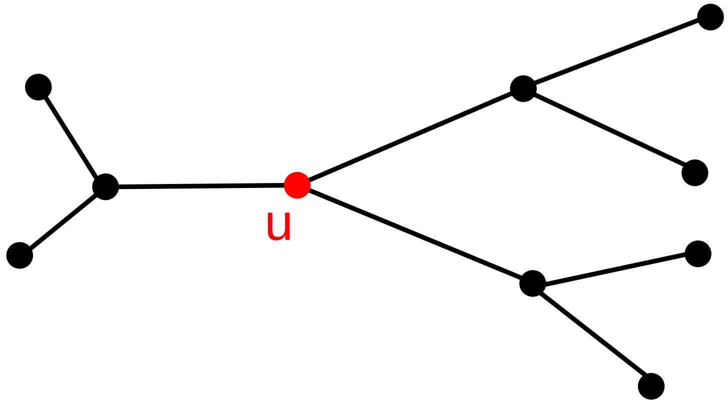
- The SILC path encoding takes advantage of the path coherence

SILC Path Encoding

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm

SILC Path Encoding

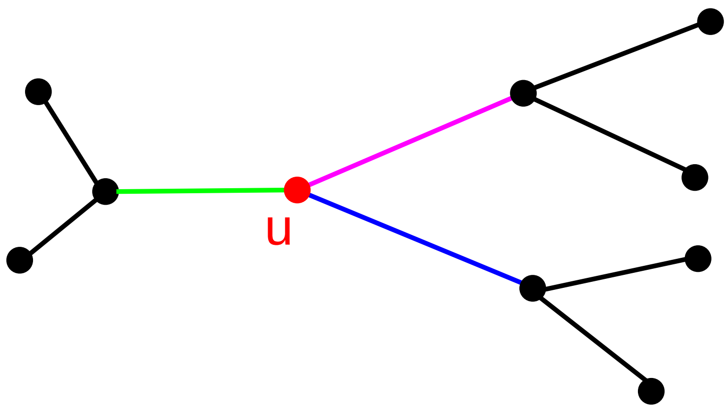
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



  - Source vertex u in a spatial network

SILC Path Encoding

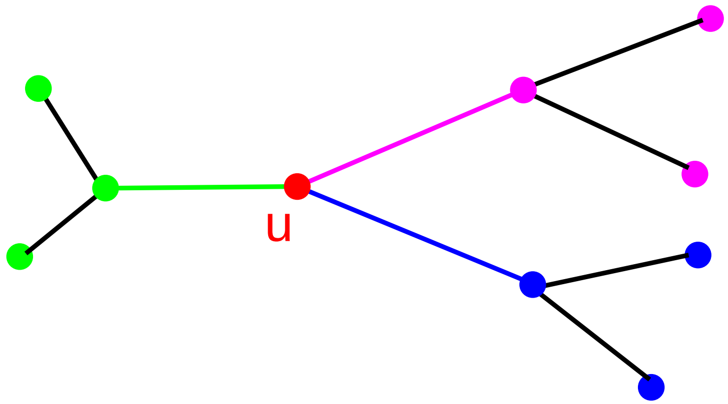
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



  - Source vertex u in a spatial network
  - Assign colors to the outgoing edges of u

SILC Path Encoding

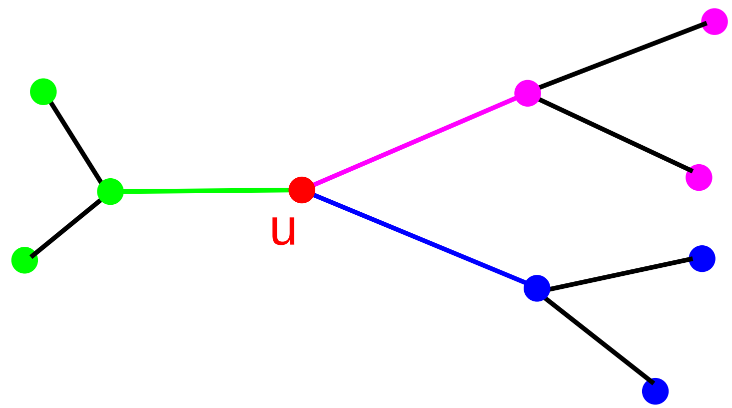
- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

SILC Path Encoding

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm

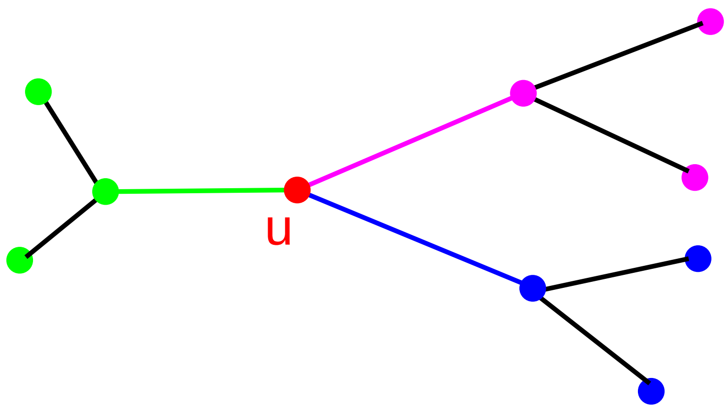


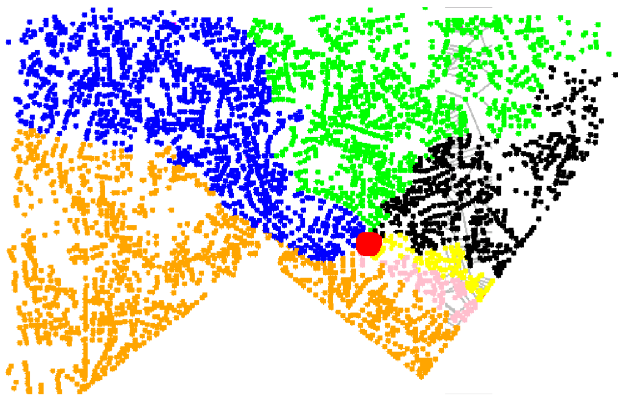
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD

SILC Path Encoding

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



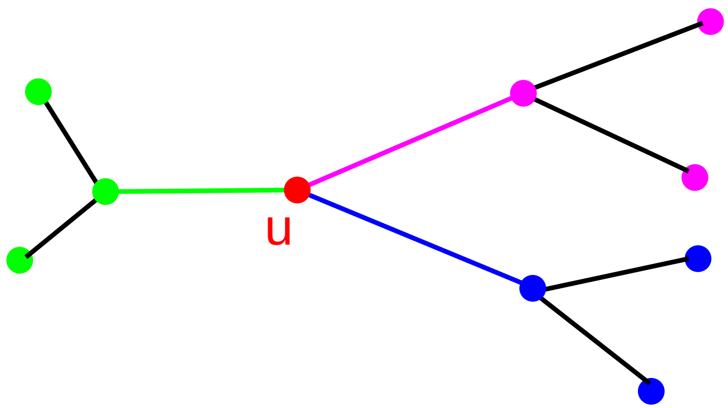


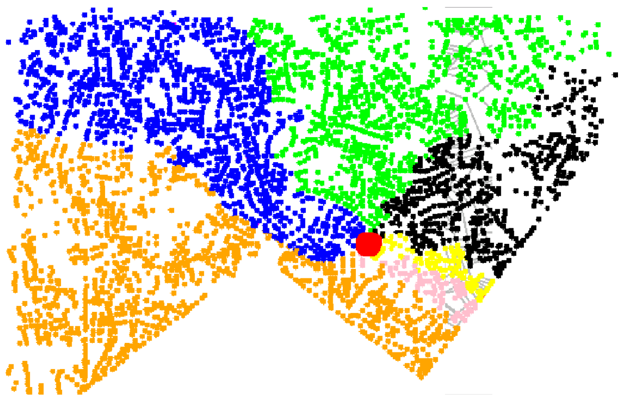
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

SILC Path Encoding

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm





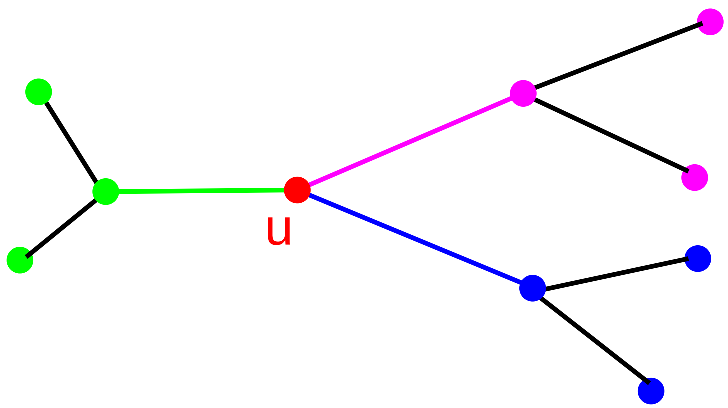
- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

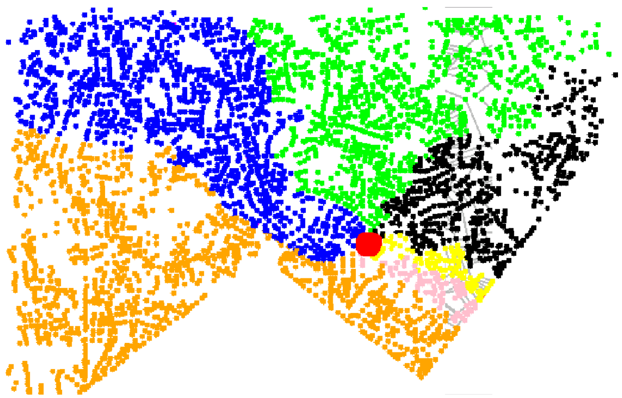
- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

- Resulting representation is termed the *shortest-path map* of u

SILC Path Encoding

- The SILC path encoding takes advantage of the path coherence
- How? Use a *coloring* algorithm



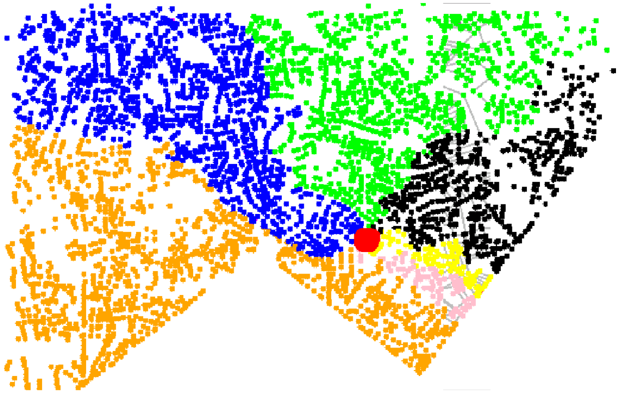


- Source vertex u in a spatial network
- Assign colors to the outgoing edges of u
- Color vertex based on the first edge on the shortest path from u

- Source vertex u in the spatial network of Silver Spring, MD
- Color remaining vertices based on which of the six adjacent vertices of u is the first link in the shortest path from u

- Resulting representation is termed the *shortest-path map* of u
- Assuming planar spatial network graphs means that the coloring results in spatially contiguous colored regions due to path coherence

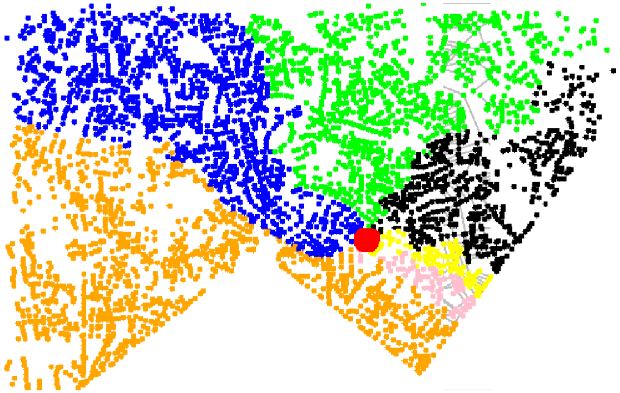
How to Store Colored Regions?



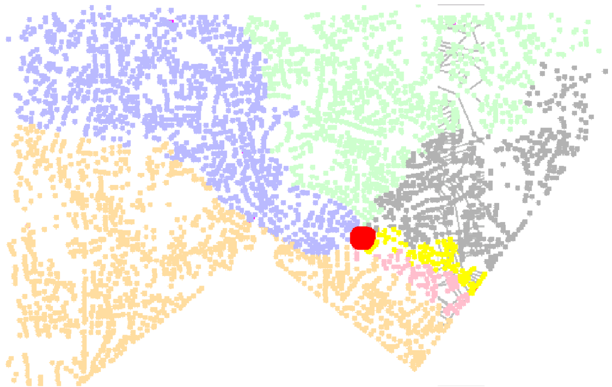
Shortest-path Map

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



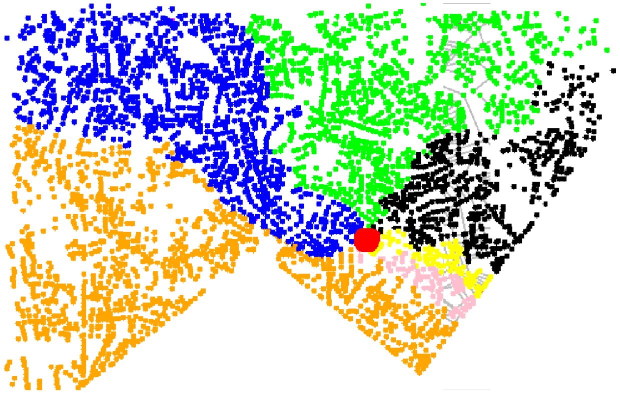
Shortest-path Map



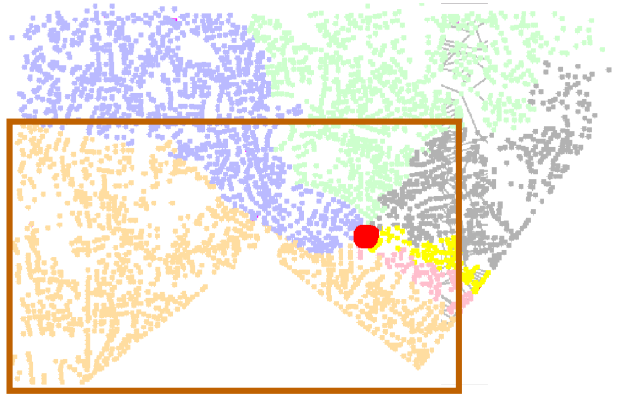
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



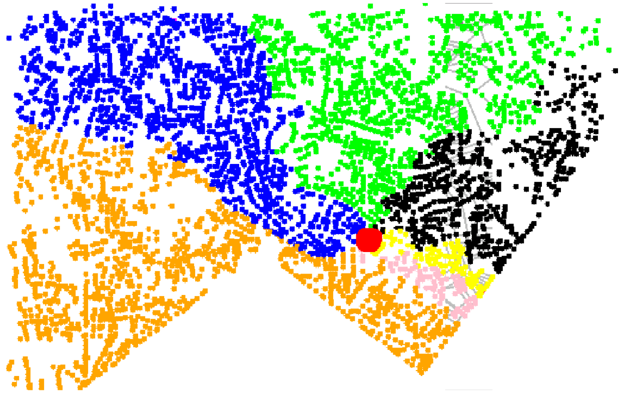
Shortest-path Map



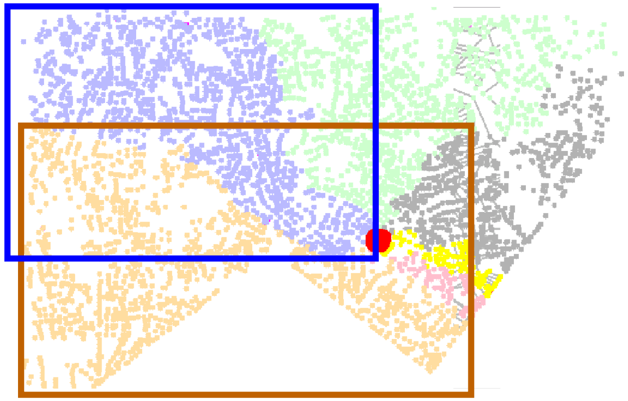
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



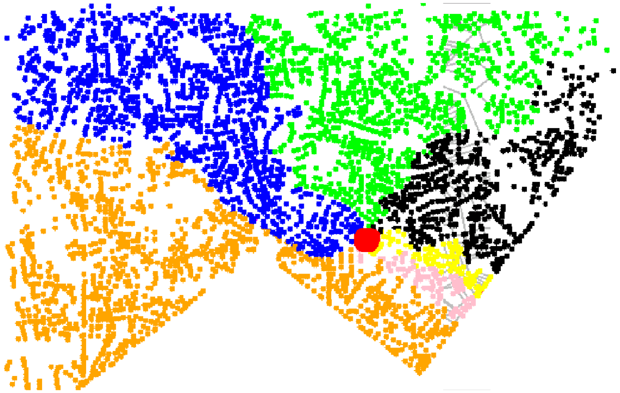
Shortest-path Map



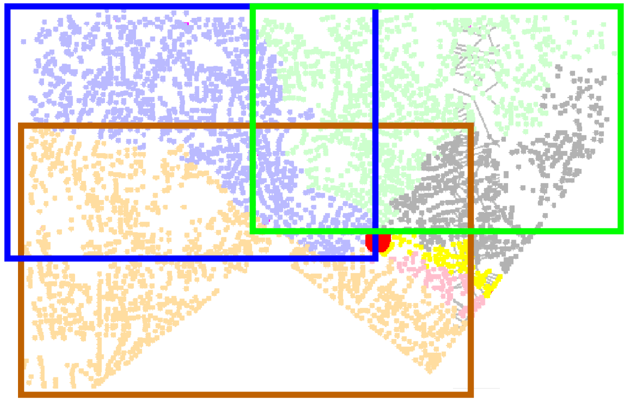
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



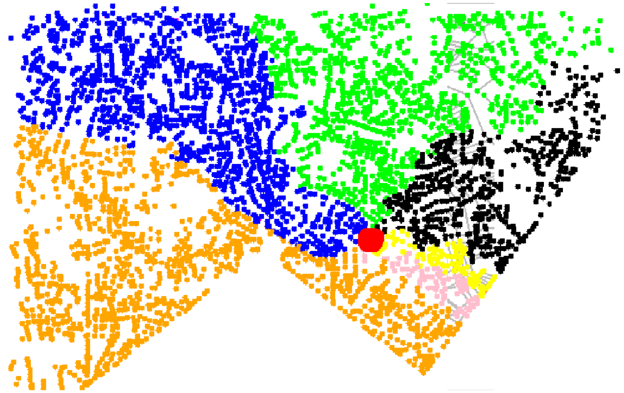
Shortest-path Map



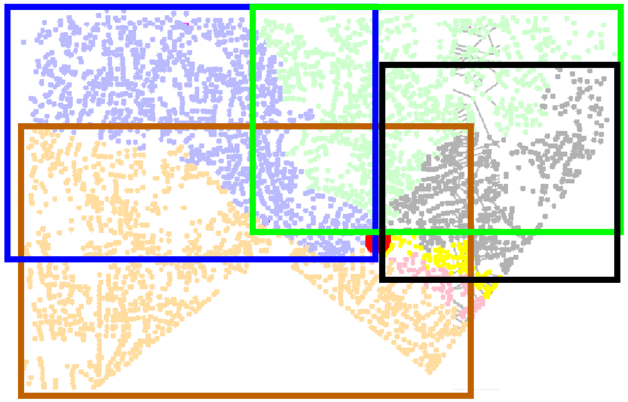
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



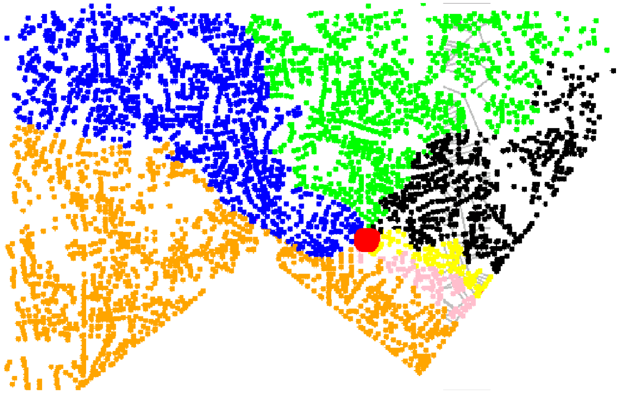
Shortest-path Map



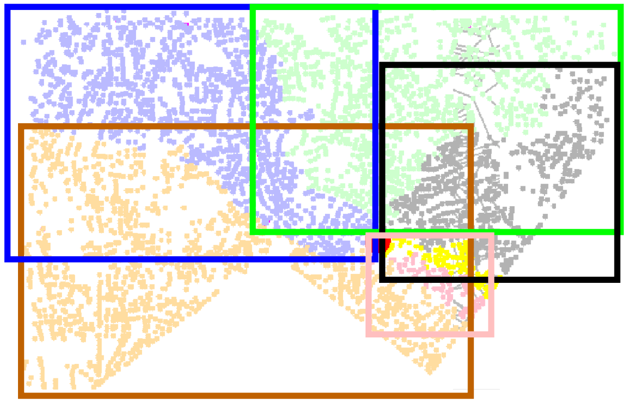
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



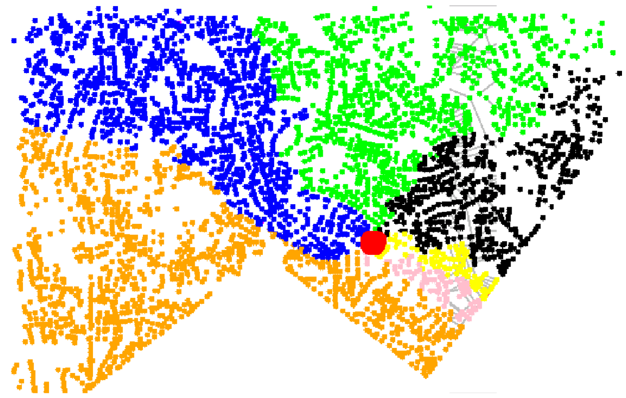
Shortest-path Map



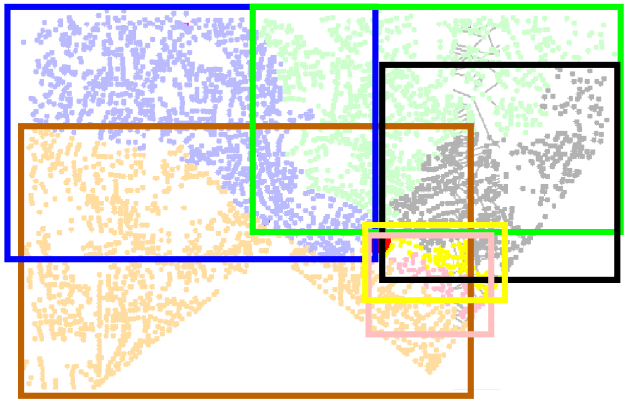
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]



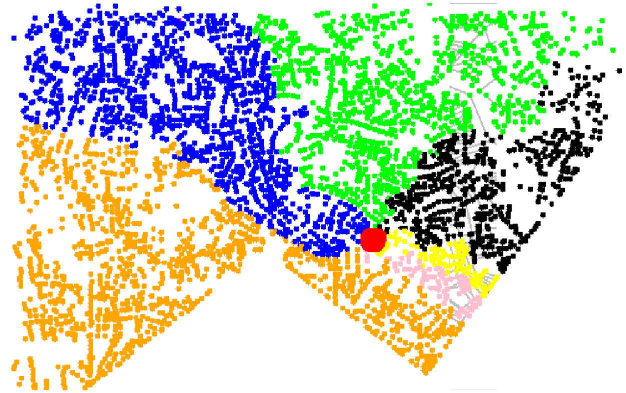
Shortest-path Map



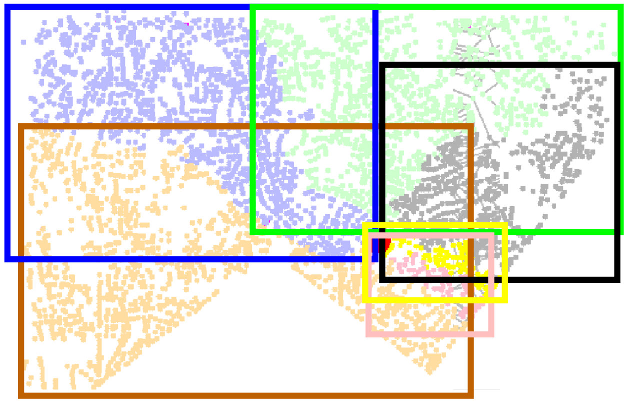
R-tree

How to Store Colored Regions?

- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm



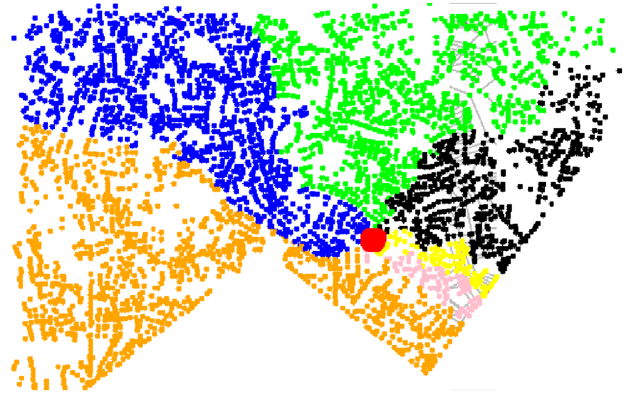
Shortest-path Map



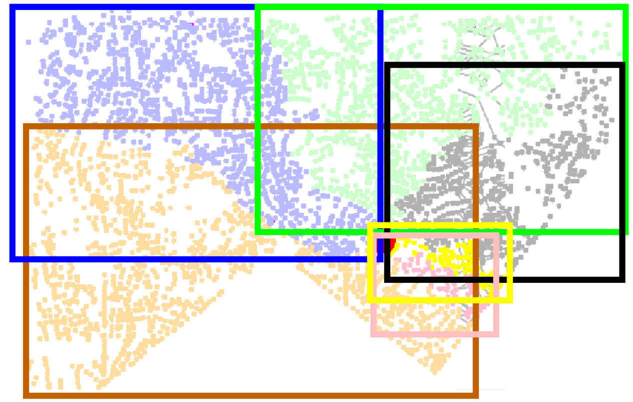
R-tree

How to Store Colored Regions?

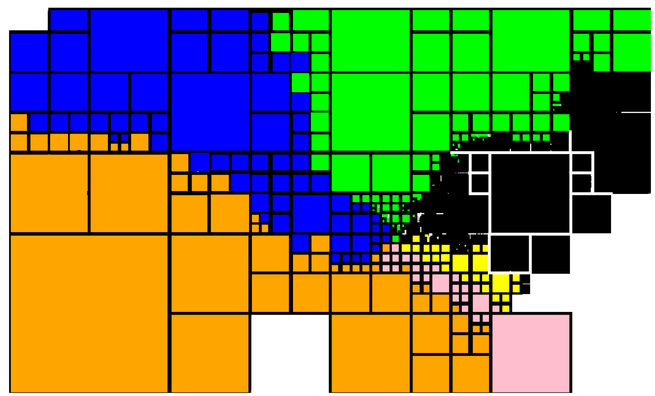
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree



Shortest-path Map



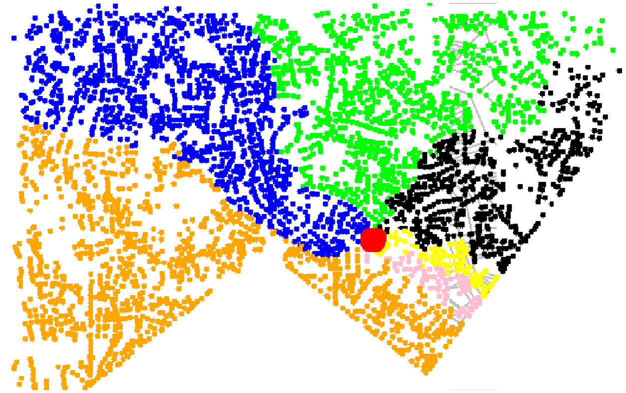
R-tree



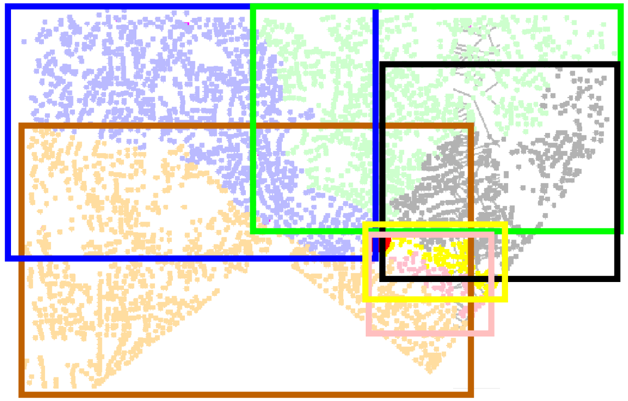
Shortest-Path Quadtree

How to Store Colored Regions?

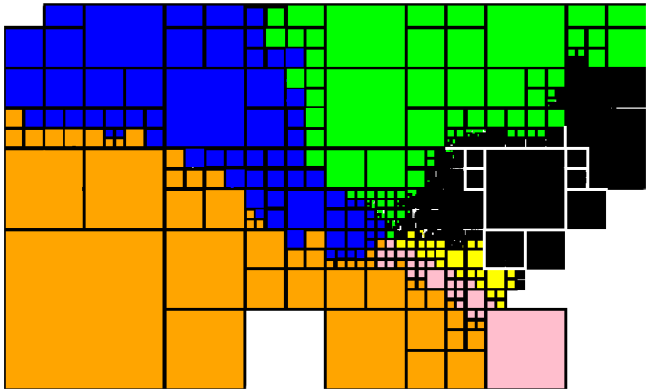
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color



Shortest-path Map



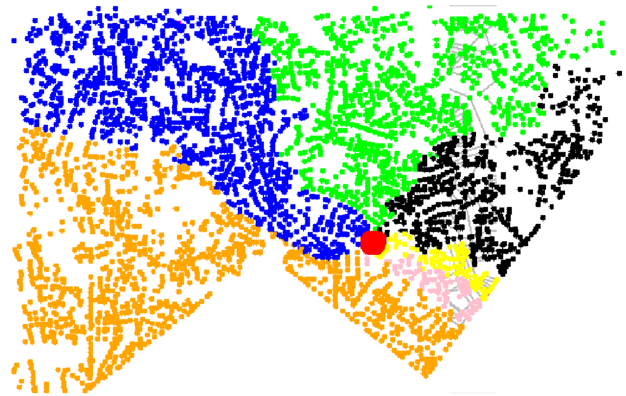
R-tree



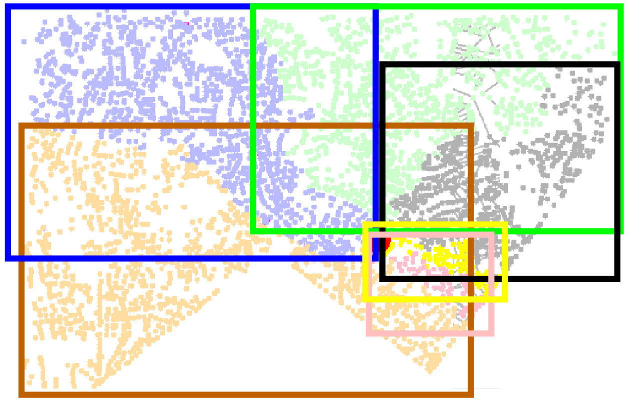
Shortest-Path Quadtree

How to Store Colored Regions?

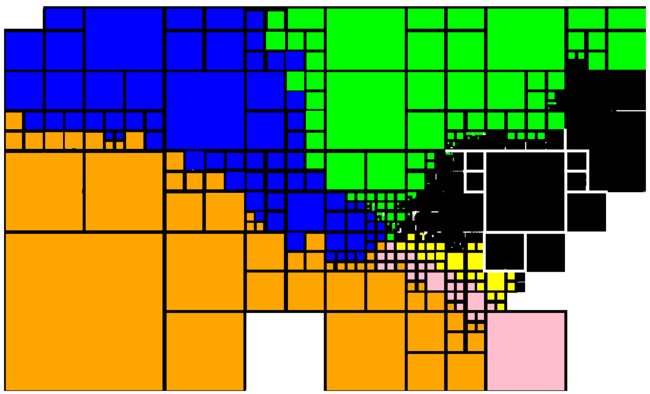
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks



Shortest-path Map



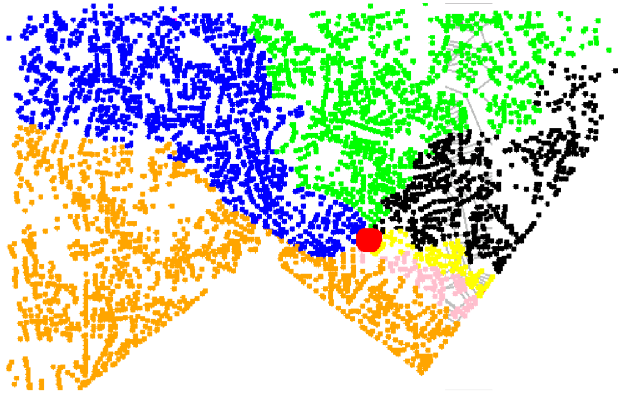
R-tree



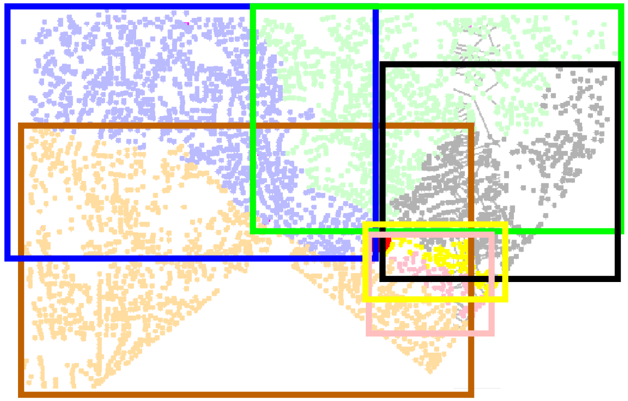
Shortest-Path Quadtree

How to Store Colored Regions?

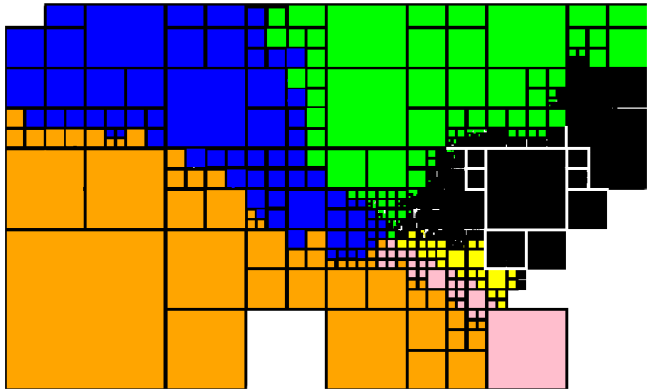
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks
  - Note: no need to store identity of vertices in the blocks



Shortest-path Map



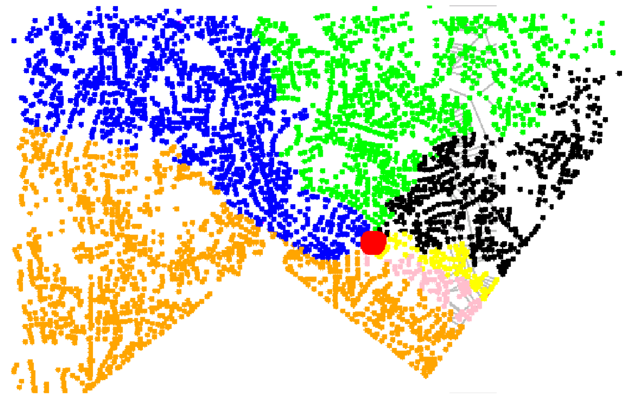
R-tree



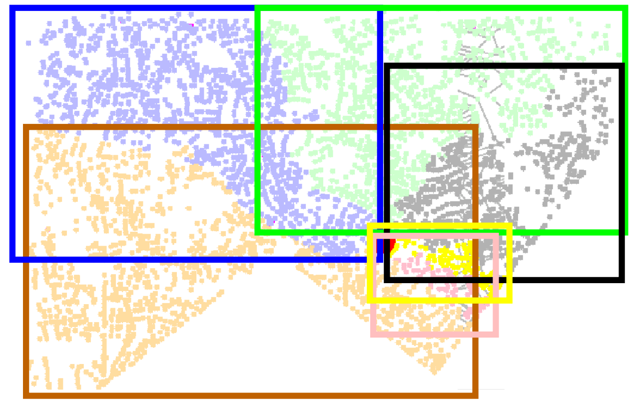
Shortest-Path Quadtree

How to Store Colored Regions?

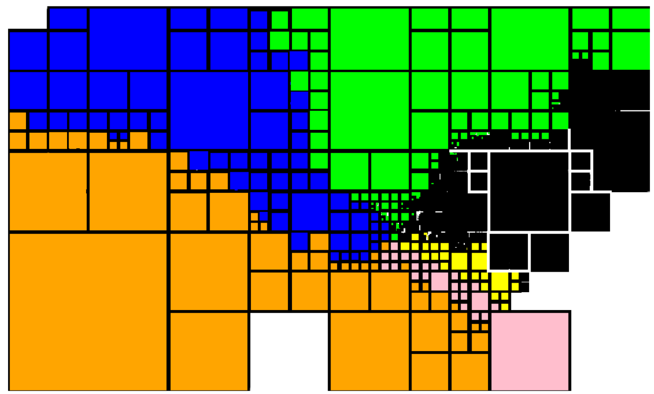
- Minimum bounding boxes (e.g., R-tree) [Wagn03]
  - overlapping boxes imply identity of next vertex cannot be uniquely determined causing the shortest path algorithm to possibly degenerate to Dijkstra's algorithm
- Disjoint decomposition: shortest-path quadtree
  - Decompose until all vertices in block have the same color
- Shortest-path quadtree stored as a collection of Morton blocks
  - Note: no need to store identity of vertices in the blocks
- Proposed encoding leverages the dimensionality reduction property of MX and region quadtrees
  - Required storage cost to represent a region R in a region and MX quadtree is $O(p)$, where p is the perimeter of R



Shortest-path Map



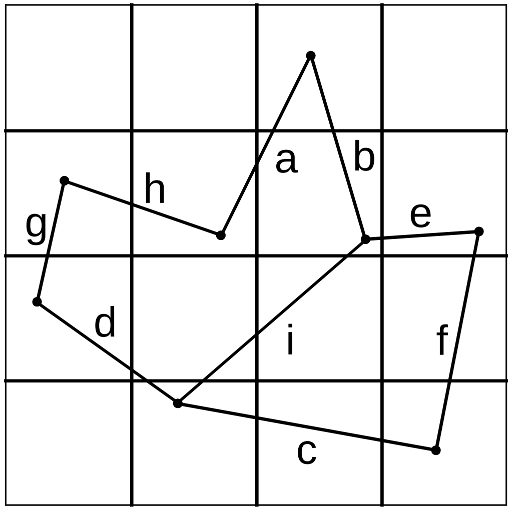
R-tree



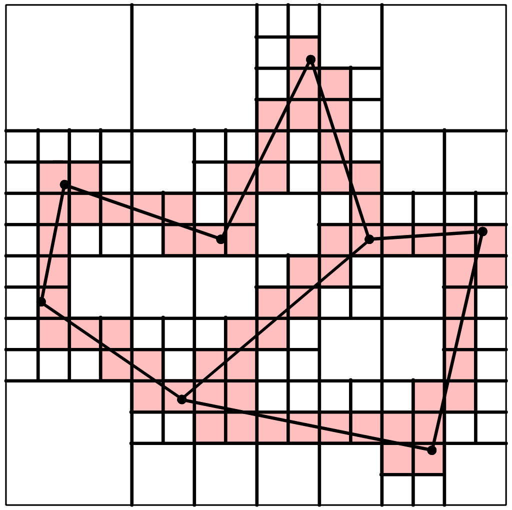
Shortest-Path Quadtree

Quadtree Complexity Theorem

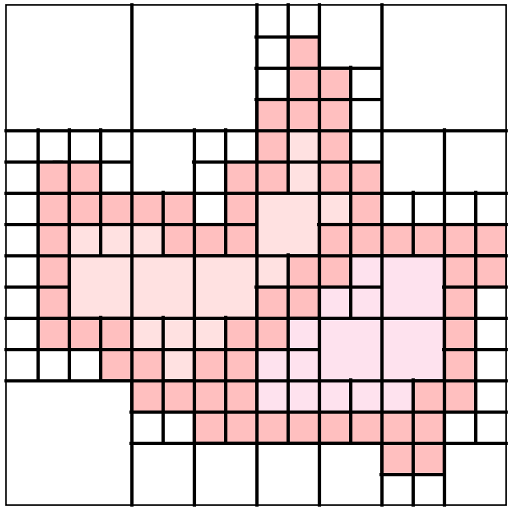
- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)



Simple Polygons



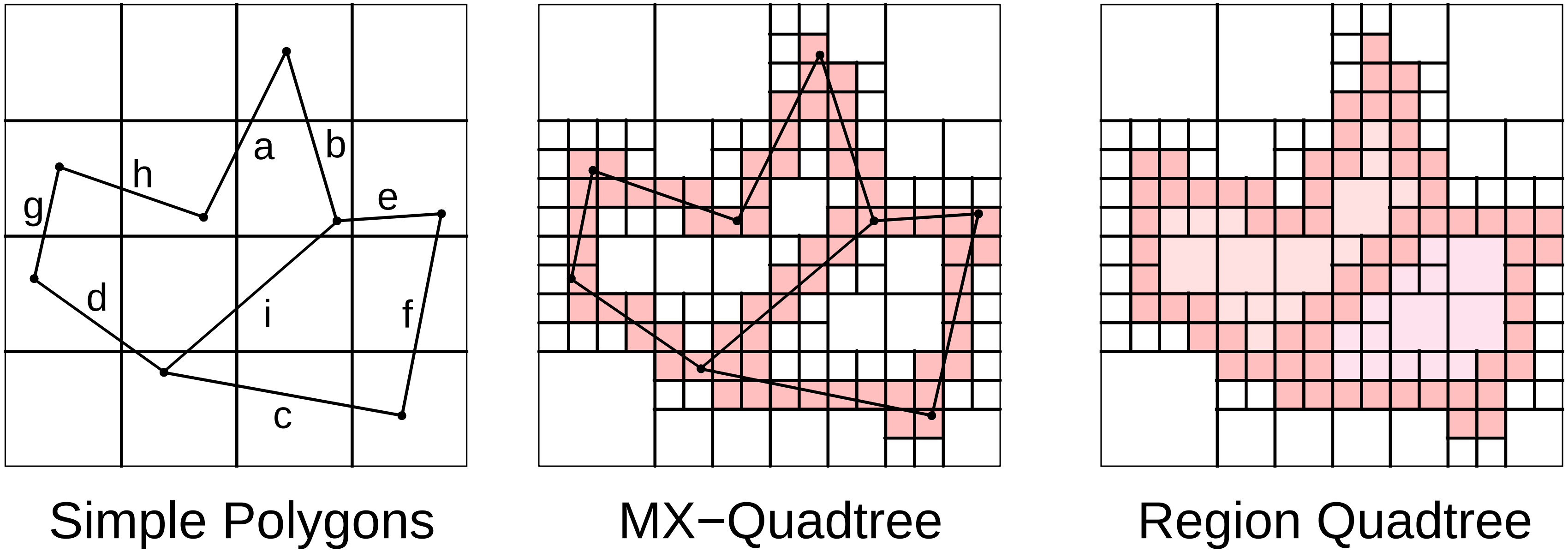
MX−Quadtree



Region Quadtree

Quadtree Complexity Theorem

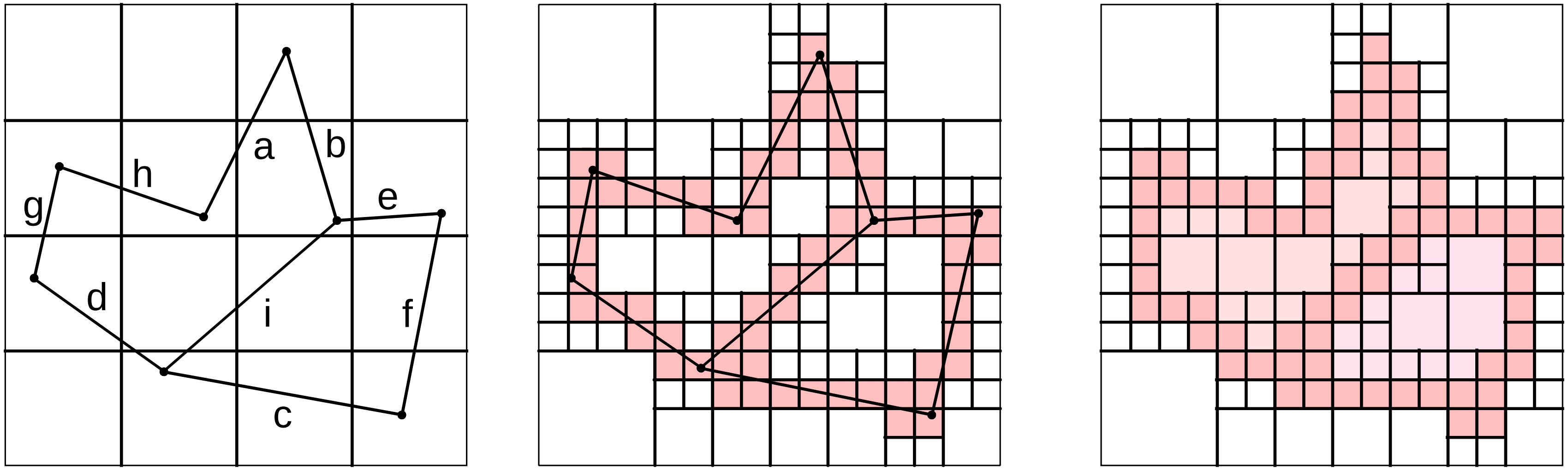
- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)



- Easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

Quadtree Complexity Theorem

- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)

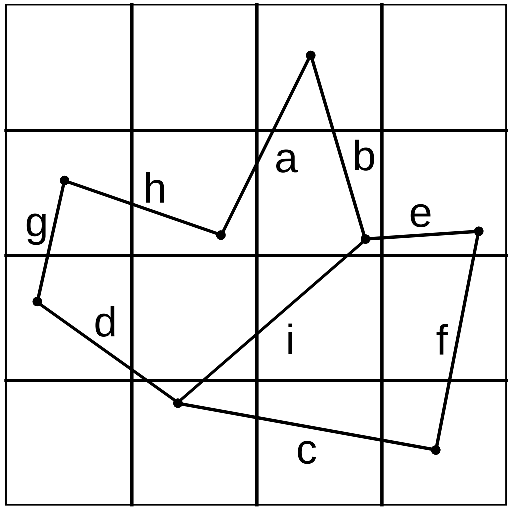


Simple Polygons　　MX−Quadtree　　Region Quadtree

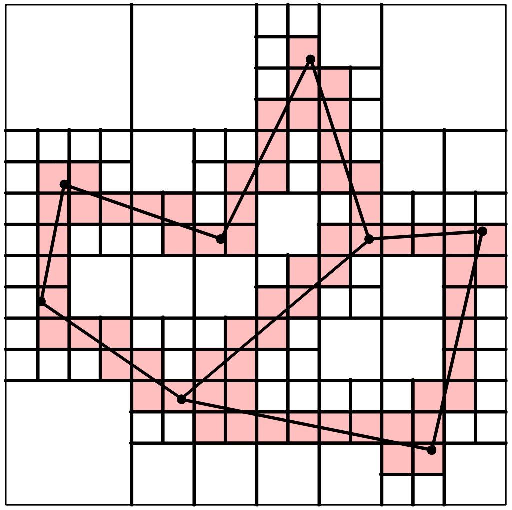
- Easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases
- Shortest-path quadtree requires less space than MX and region quadtrees as no decomposition takes place at boundaries that pass through empty nodes even though number of polygons exceeds the vertex outdegree

Quadtree Complexity Theorem

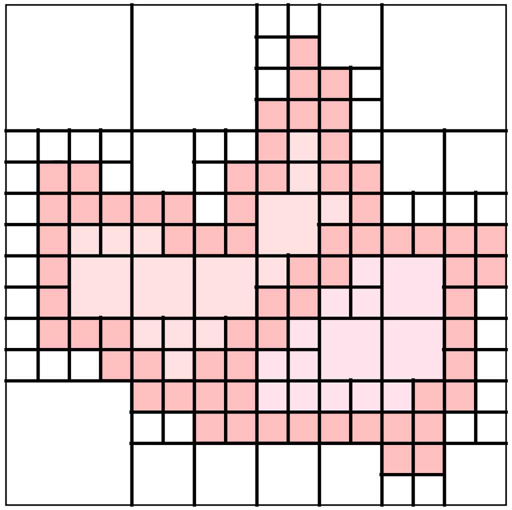
- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)



Simple Polygons



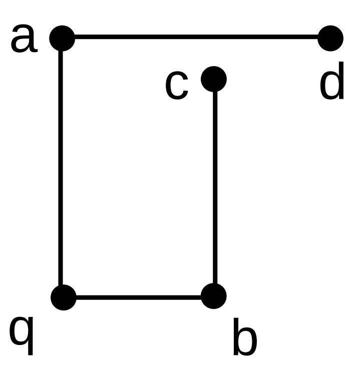
MX−Quadtree



Region Quadtree

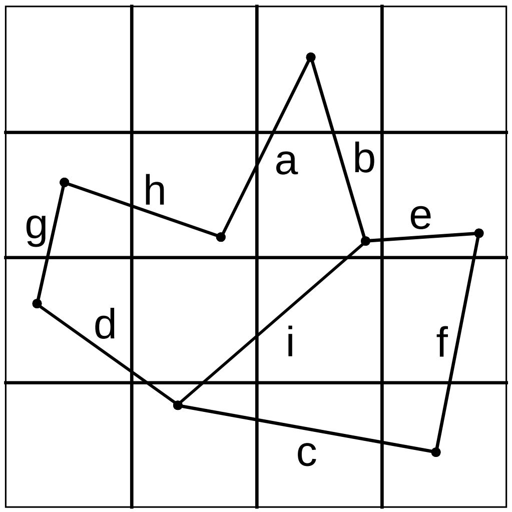
- Easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

- Shortest-path quadtree requires less space than MX and region quadtrees as no decomposition takes place at boundaries that pass through empty nodes even though number of polygons exceeds the vertex outdegree

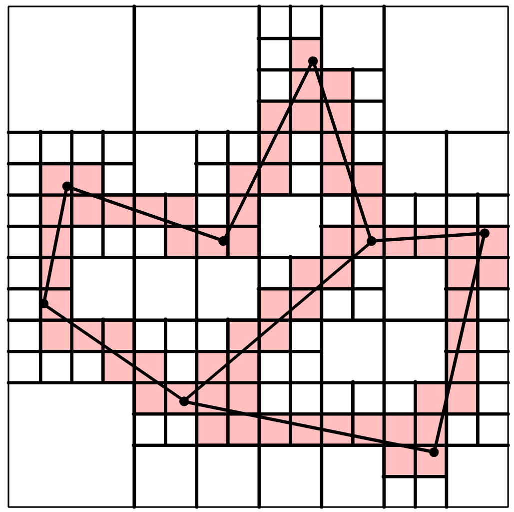


Quadtree Complexity Theorem

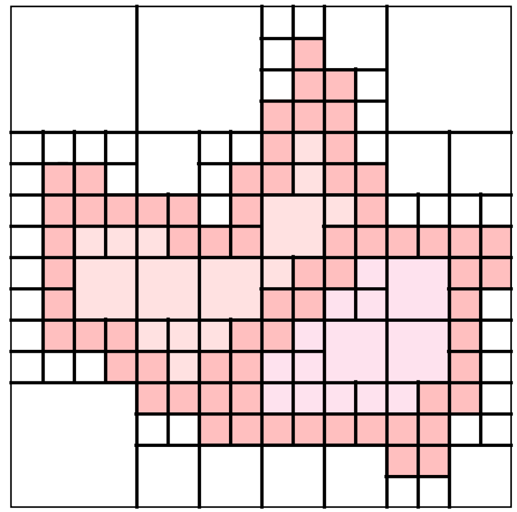
- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)



Simple Polygons



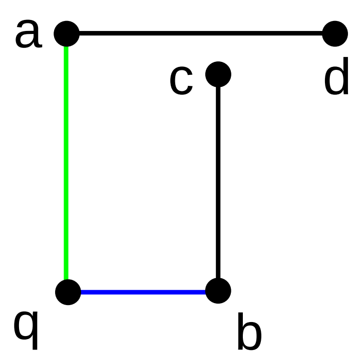
MX−Quadtree



Region Quadtree

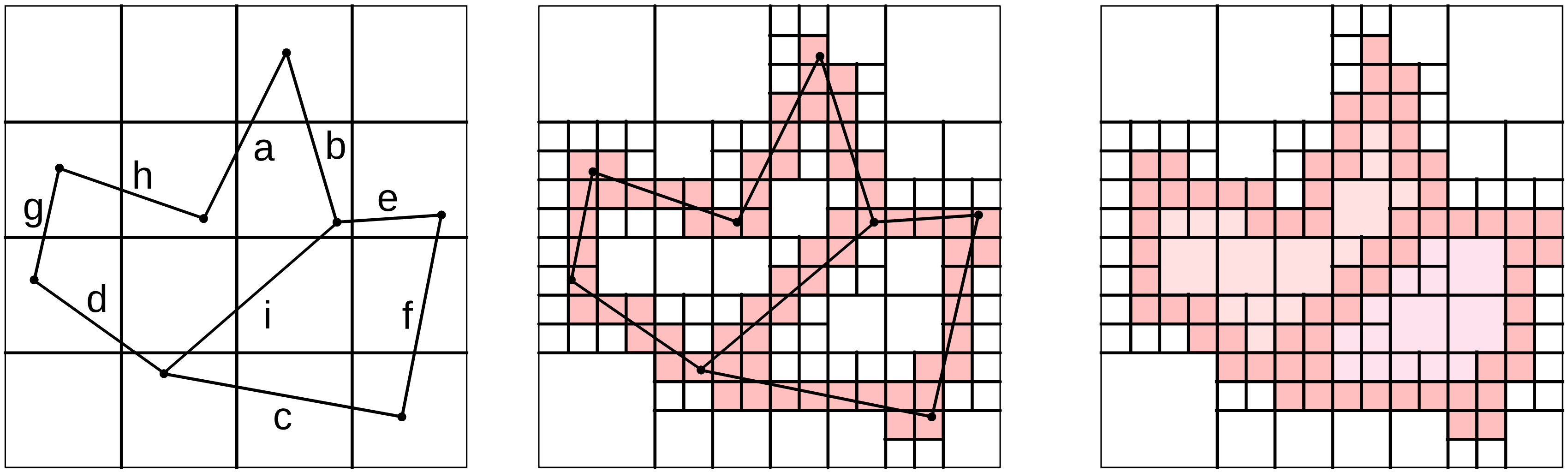
- Easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

- Shortest-path quadtree requires less space than MX and region quadtrees as no decomposition takes place at boundaries that pass through empty nodes even though number of polygons exceeds the vertex outdegree



Quadtree Complexity Theorem

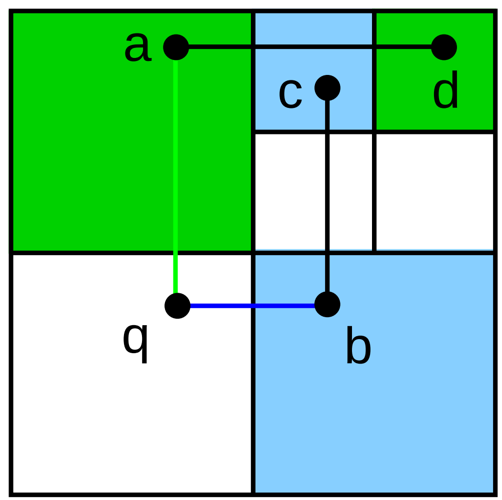
- Quadtree corresponding to a polygon of perimeter p embedded in a $2^q \times 2^q$ image has $O(p+q)$ nodes (Hunter)



Simple Polygons MX−Quadtree Region Quadtree

- Easy to see dependence on perimeter as decomposition only takes place on the boundary as the resolution increases

- Shortest-path quadtree requires less space than MX and region quadtrees as no decomposition takes place at boundaries that pass through empty nodes even though number of polygons exceeds the vertex outdegree



Quadtree Complexity Theorem on Shortest-Path Map

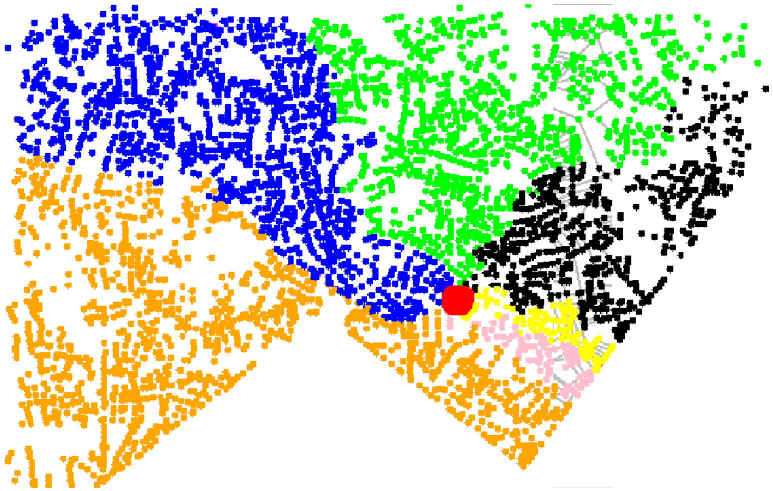
- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions

Quadtree Complexity Theorem on Shortest-Path Map

- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous

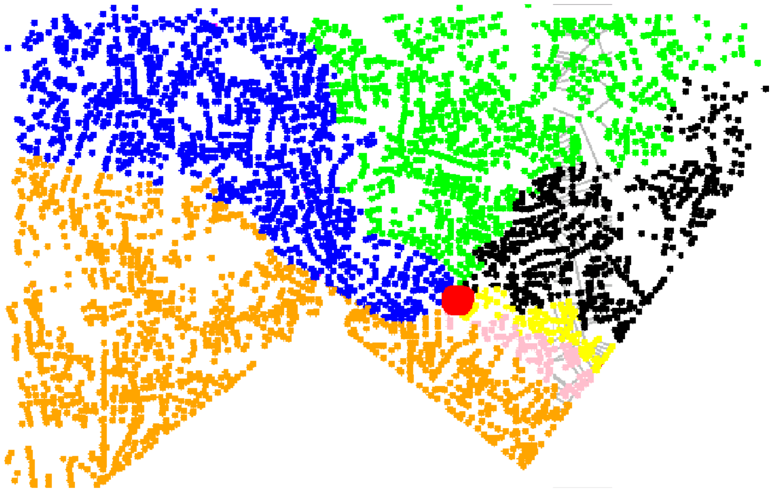
Quadtree Complexity Theorem on Shortest-Path Map

- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous



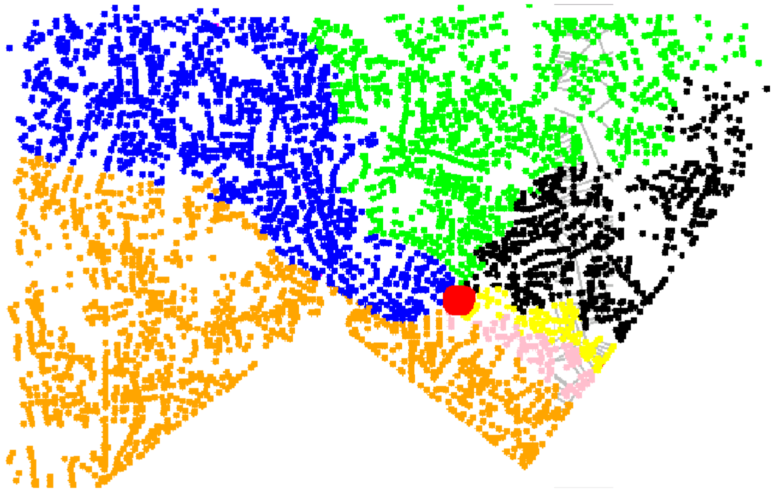
Quadtree Complexity Theorem on Shortest-Path Map

- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous
- Quadtree Complexity Theorem can be applied to the MX-quadtree for the polygons containing the regions in the shortest-path map



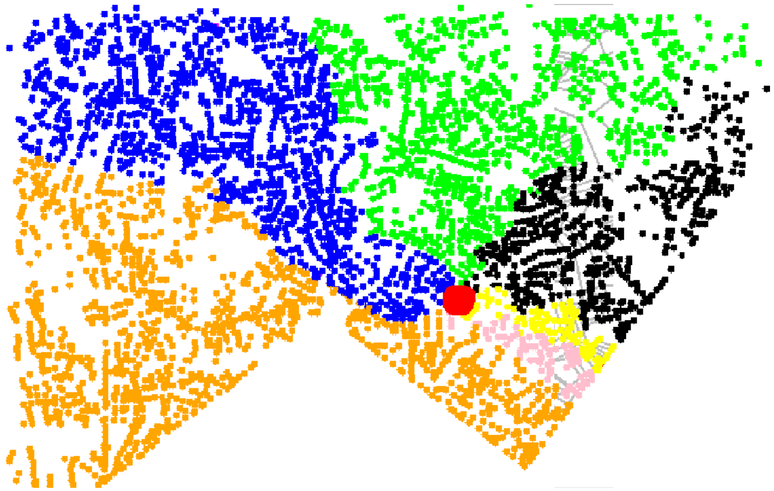
Quadtree Complexity Theorem on Shortest-Path Map

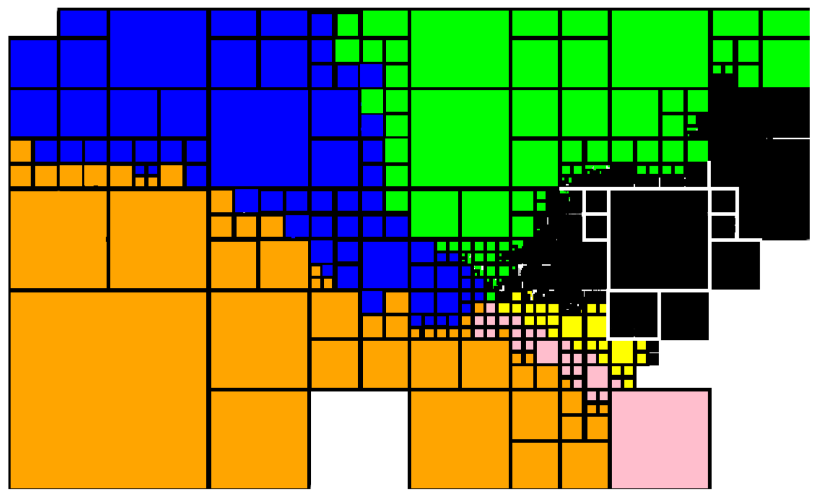
- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous
- Quadtree Complexity Theorem can be applied to the MX-quadtree for the polygons containing the regions in the shortest-path map
- Size of shortest-path quadtree is no more than the MX quadtree as no need to decompose, to the pixel level, the empty blocks through which the boundaries pass



Quadtree Complexity Theorem on Shortest-Path Map

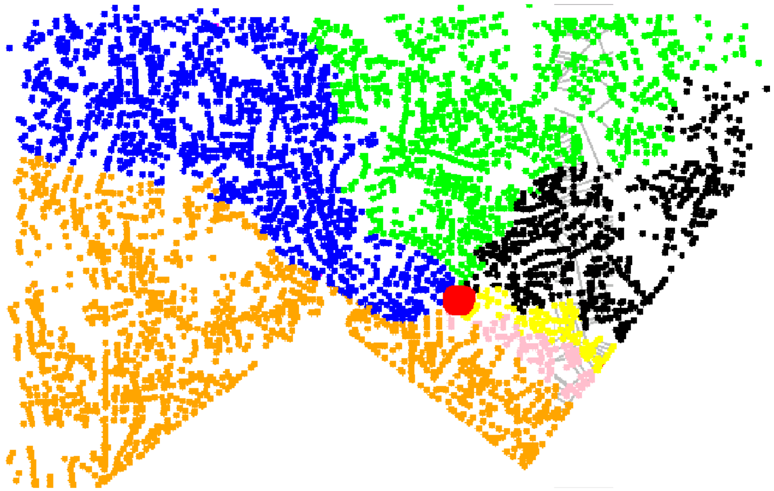
- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous
- Quadtree Complexity Theorem can be applied to the MX-quadtree for the polygons containing the regions in the shortest-path map
- Size of shortest-path quadtree is no more than the MX quadtree as no need to decompose, to the pixel level, the empty blocks through which the boundaries pass

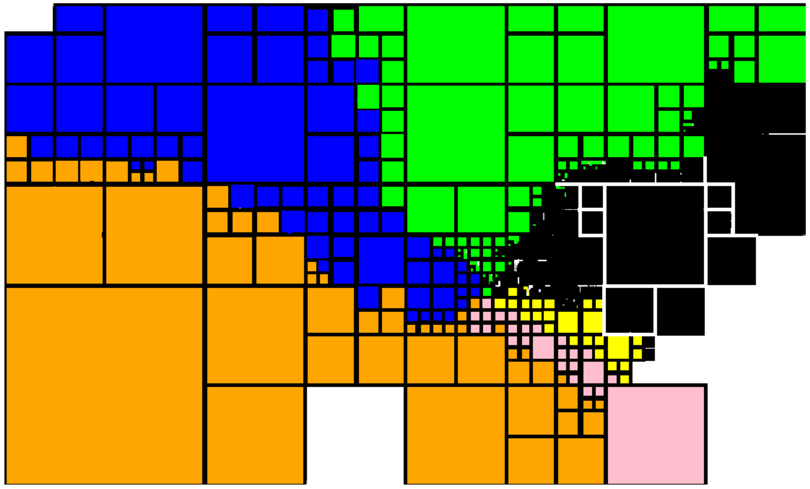




Quadtree Complexity Theorem on Shortest-Path Map

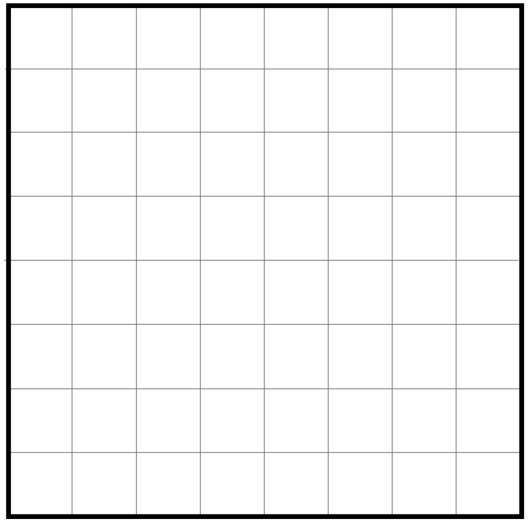
- Quadtree Complexity theorem cannot be directly applied to shortest-path quadtrees owing to the discontinuous regions
- However, for planar graphs the shortest-path map of a vertex is contiguous
- Quadtree Complexity Theorem can be applied to the MX-quadtree for the polygons containing the regions in the shortest-path map
- Size of shortest-path quadtree is no more than the MX quadtree as no need to decompose, to the pixel level, the empty blocks through which the boundaries pass
- Hence, shortest-path quadtrees are at worse O(perimeter) i.e., dimension reducing





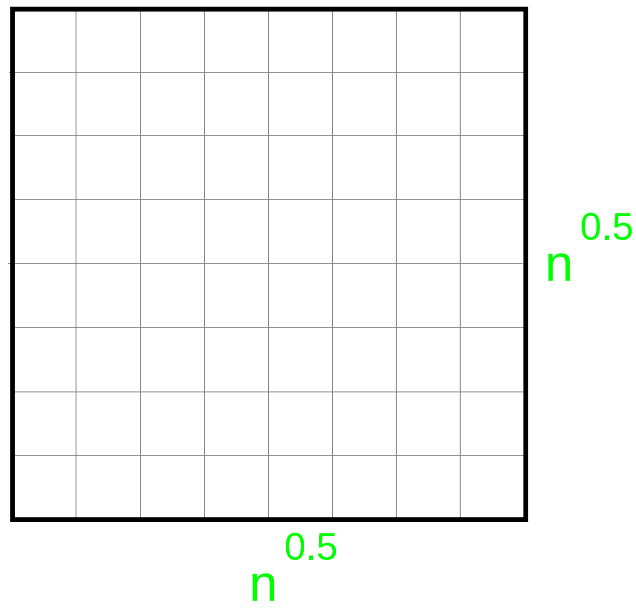
Space Complexity Analysis of Shortest-Path Quadtrees

- Consider a spatial network containing N vertices



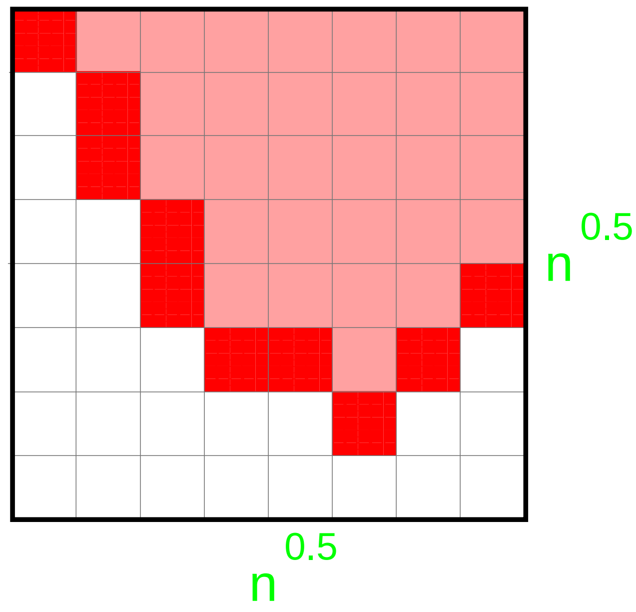
Space Complexity Analysis of Shortest-Path Quadtrees

- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it



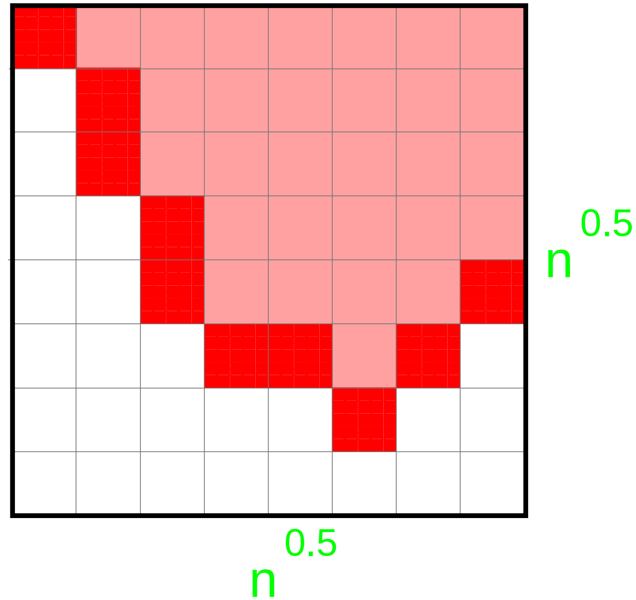
Space Complexity Analysis of Shortest-Path Quadtrees

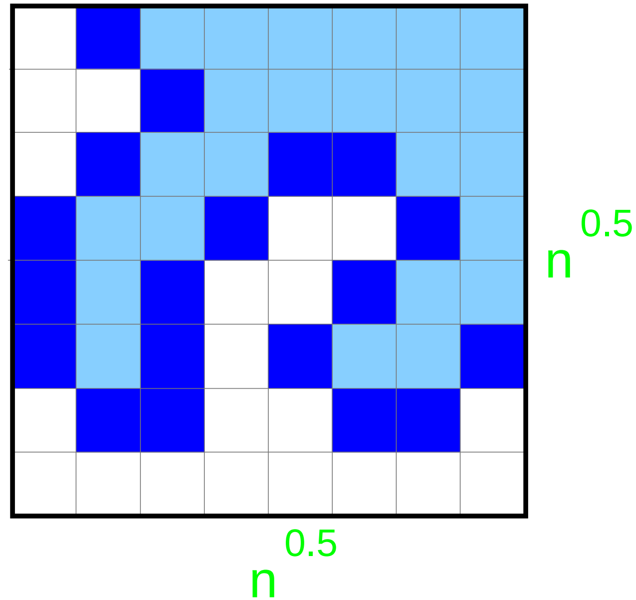
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$



Space Complexity Analysis of Shortest-Path Quadtrees

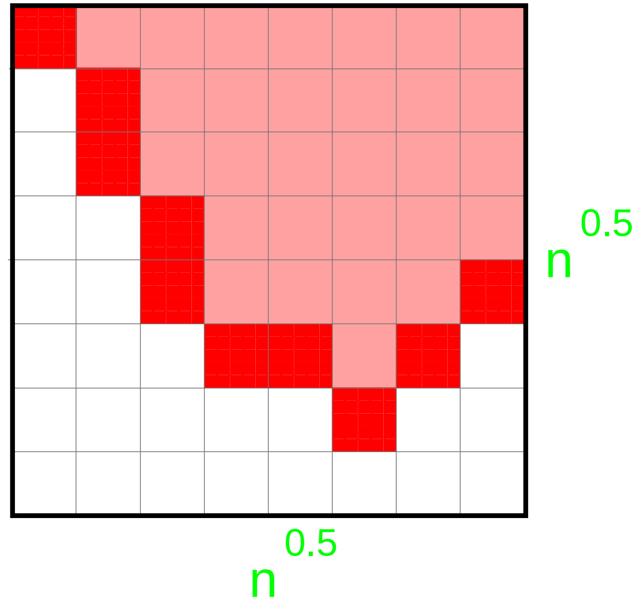
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$

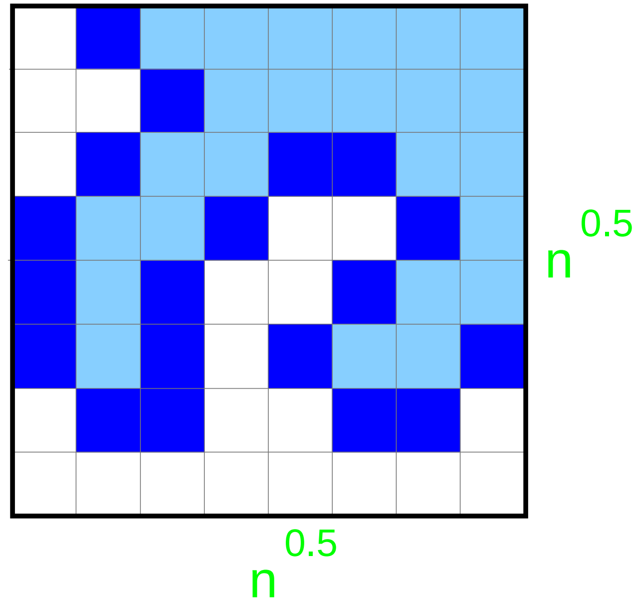




Space Complexity Analysis of Shortest-Path Quadtrees

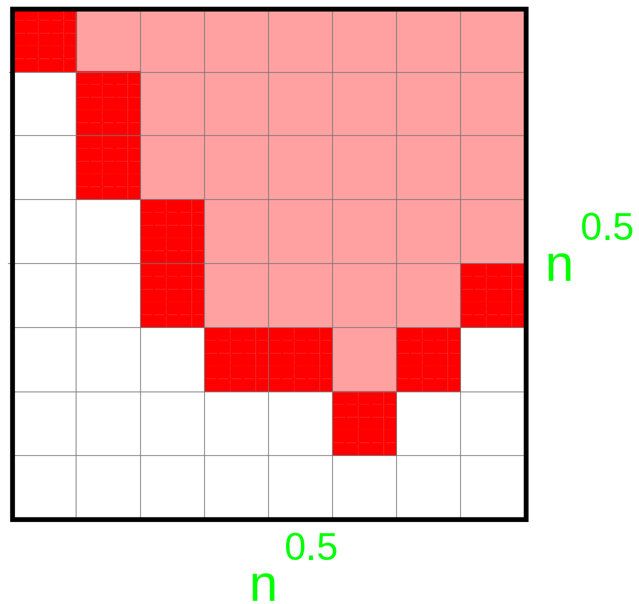
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries

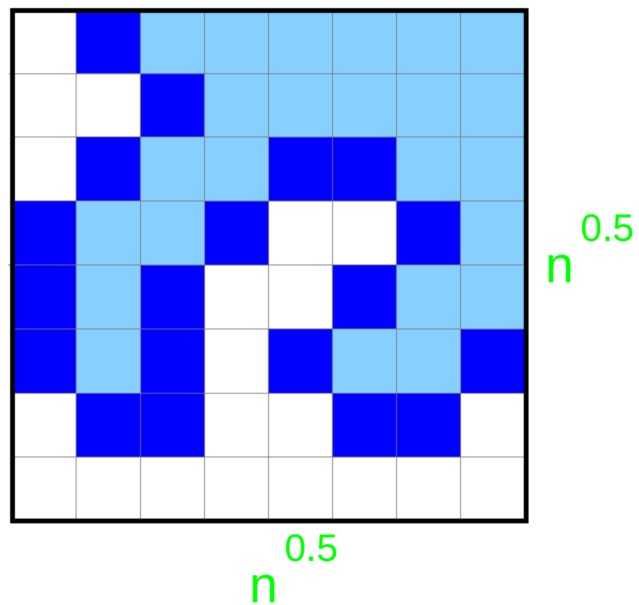




Space Complexity Analysis of Shortest-Path Quadtrees

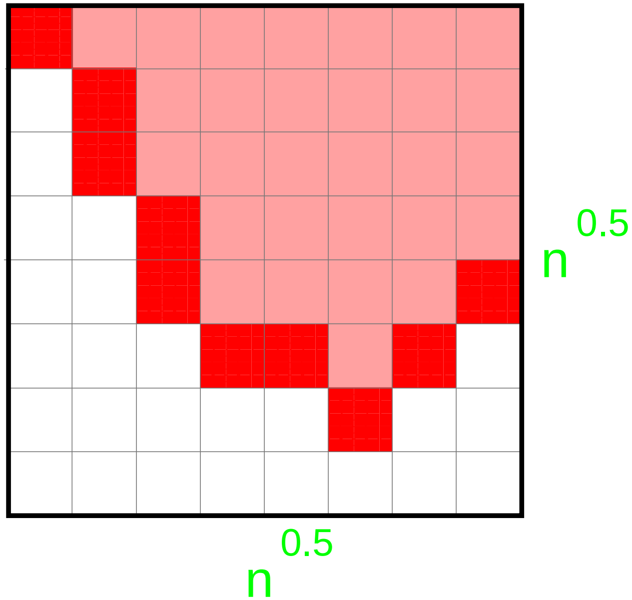
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$

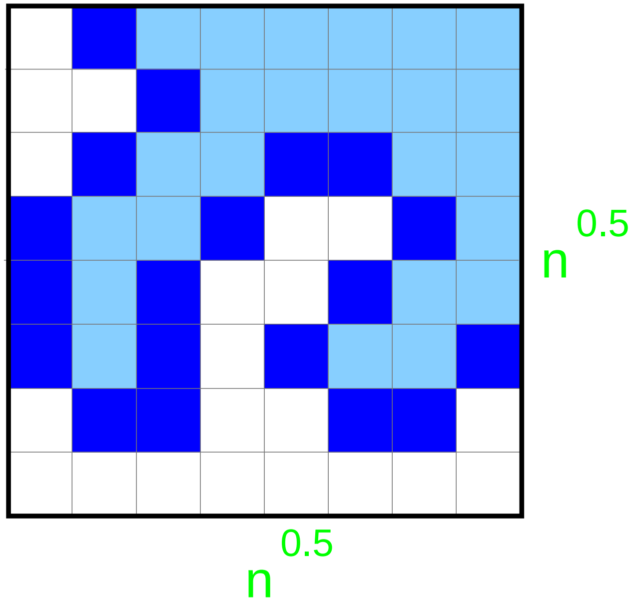




Space Complexity Analysis of Shortest-Path Quadtrees

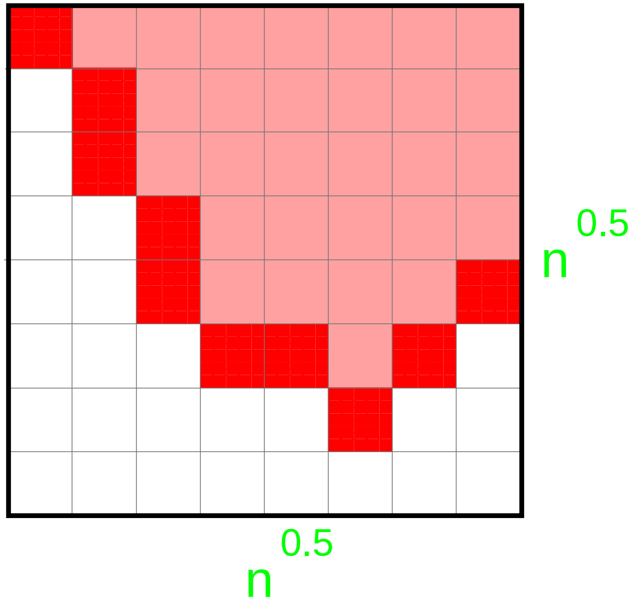
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u

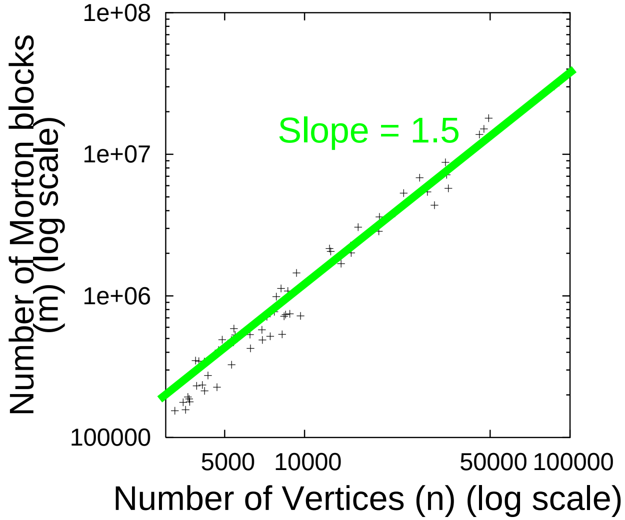




Space Complexity Analysis of Shortest-Path Quadtrees

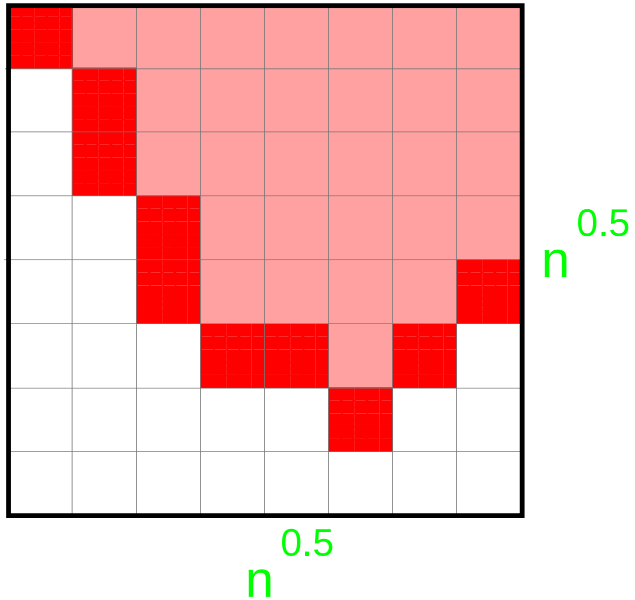
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u
- Total storage complexity of the SILC framework is $O(N\sqrt{N})$; closely follows empirical results

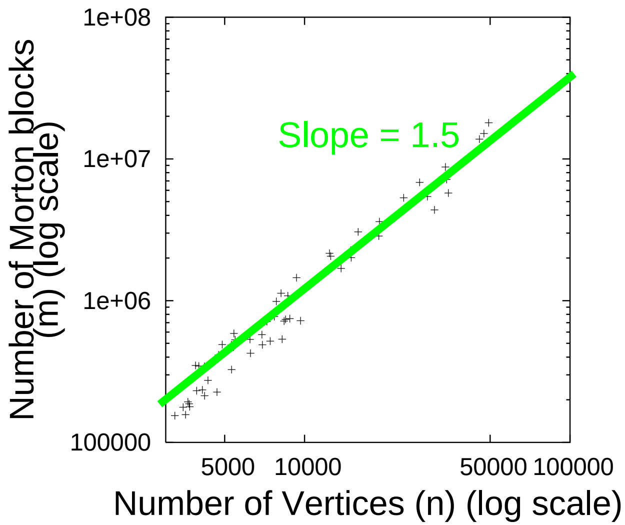




Space Complexity Analysis of Shortest-Path Quadtrees

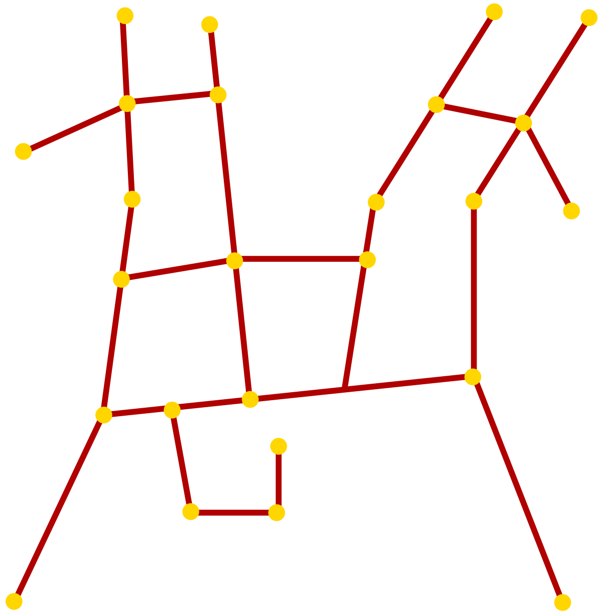
- Consider a spatial network containing N vertices in a square grid of size $N^{0.5} \times N^{0.5}$ and embed it
- Perimeter of a region with monotonic boundary on one of its coordinates is of size $O(N^{0.5})$
- Perimeter of a region with a non-monotonic boundary can be of size $O(N)$
- Assumption: Regions of the shortest-path quadtree have *monotonic* boundaries
- Size of a shortest-path quadtree of a vertex u is $c\sqrt{N}$, where c is a function of the outdegree of u
- Total storage complexity of the SILC framework is $O(N\sqrt{N})$; closely follows empirical results
- Contribution: A mechanism to capture shortest paths in spatial networks based solely on geometry and independent of topology or connectivity





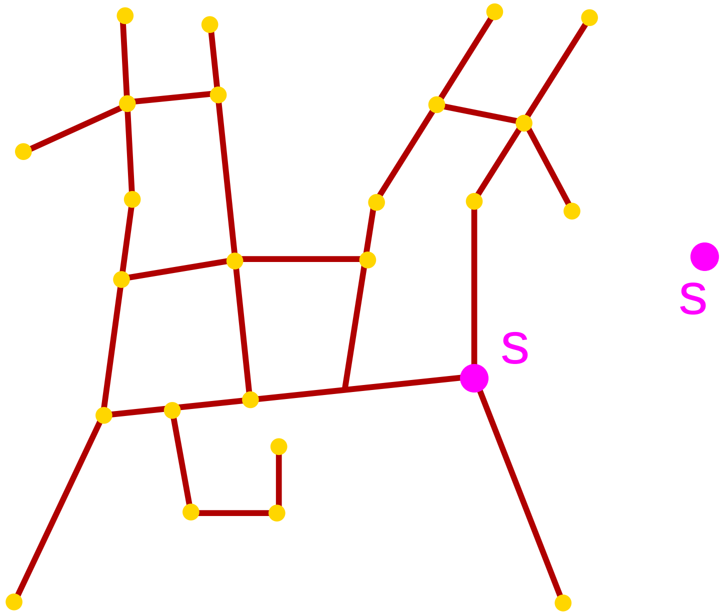
Path Retrieval

- Problem: How to retrieve the shortest path from a



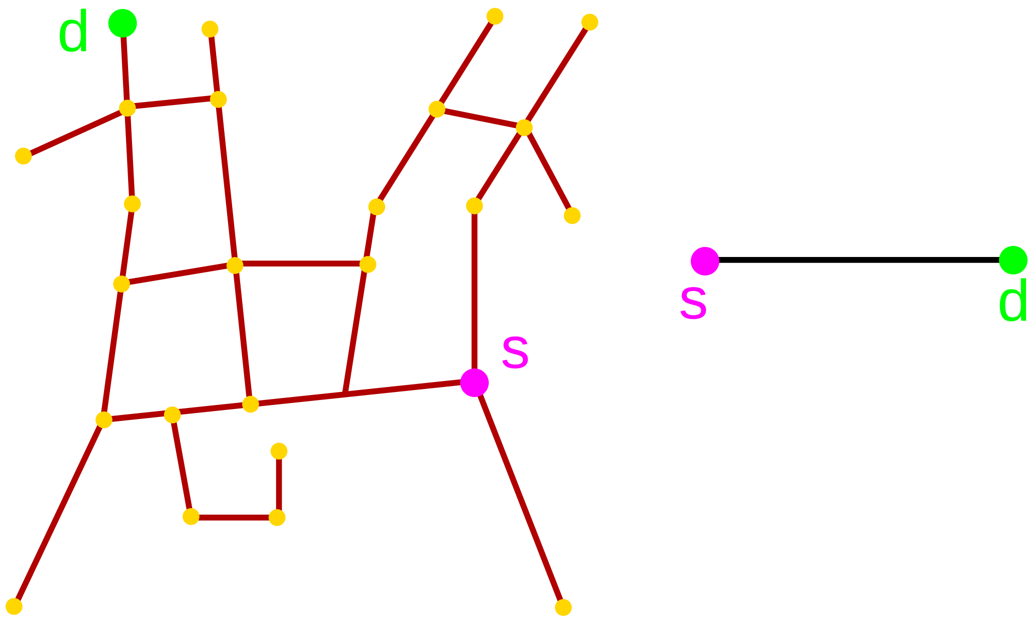
Path Retrieval

- Problem: How to retrieve the shortest path from a source s



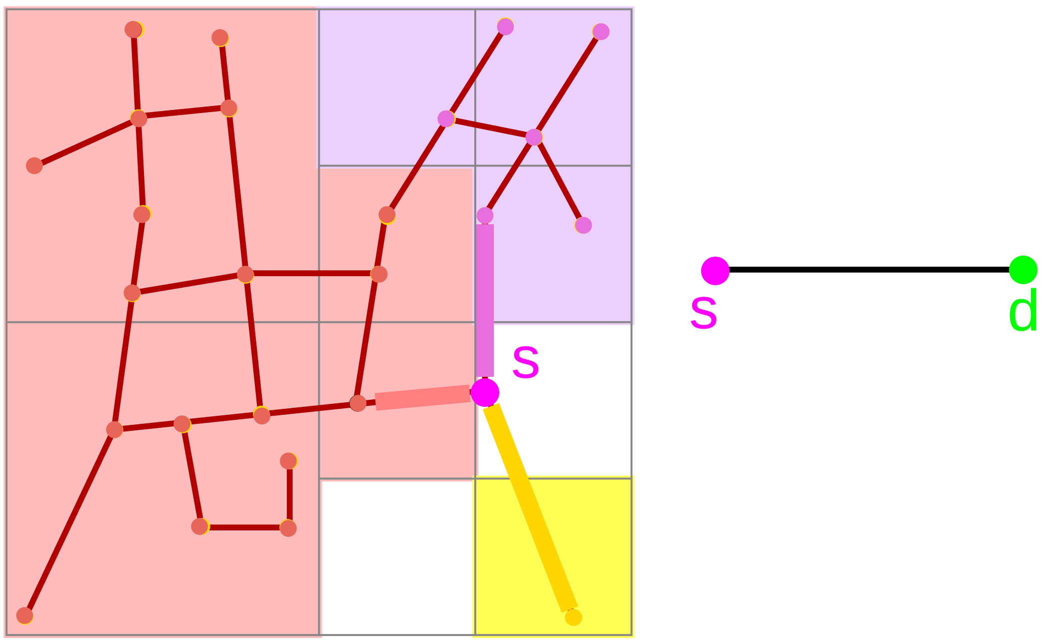
Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?



Path Retrieval

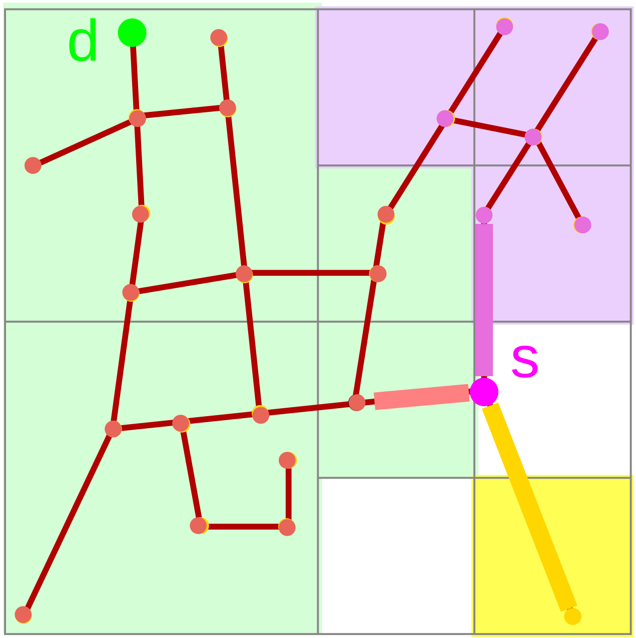
- Problem: How to retrieve the shortest path from a source s to a destination d?



- Retrieve the shortest-path quadtree Q_s corresponding to s

Path Retrieval

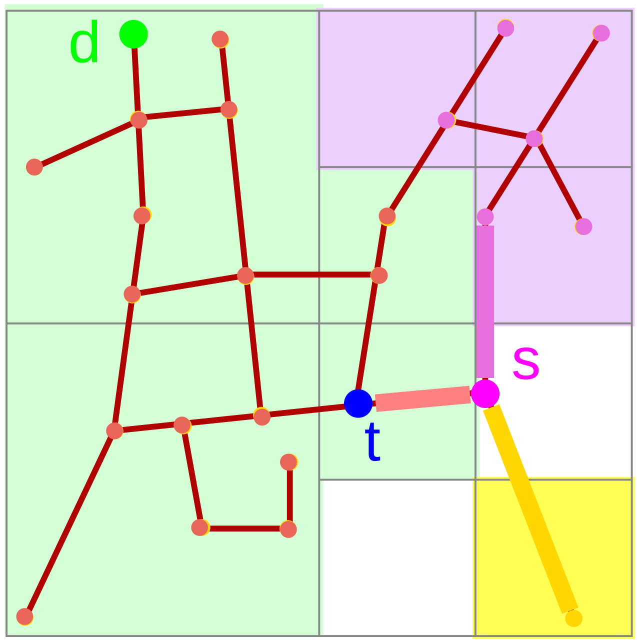
- Problem: How to retrieve the shortest path from a source s to a destination d?

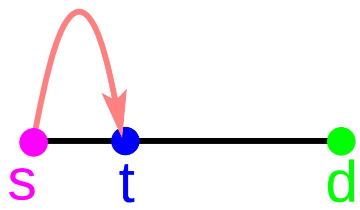


  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

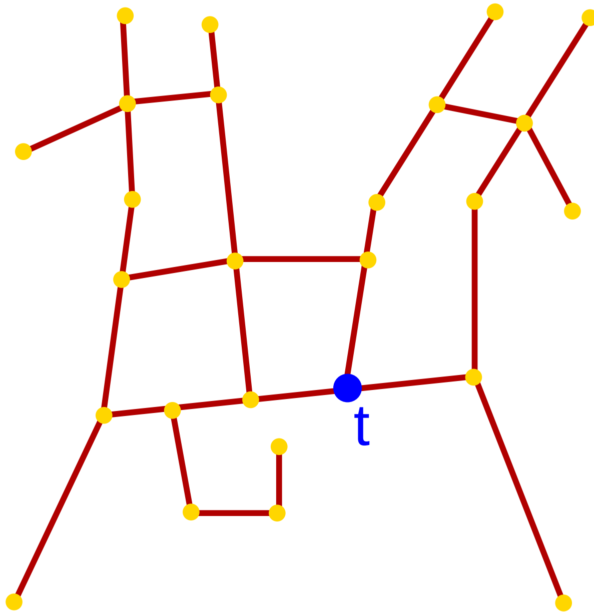


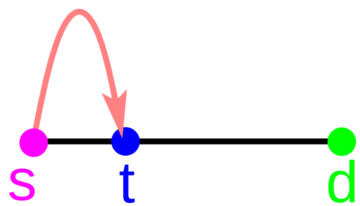


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

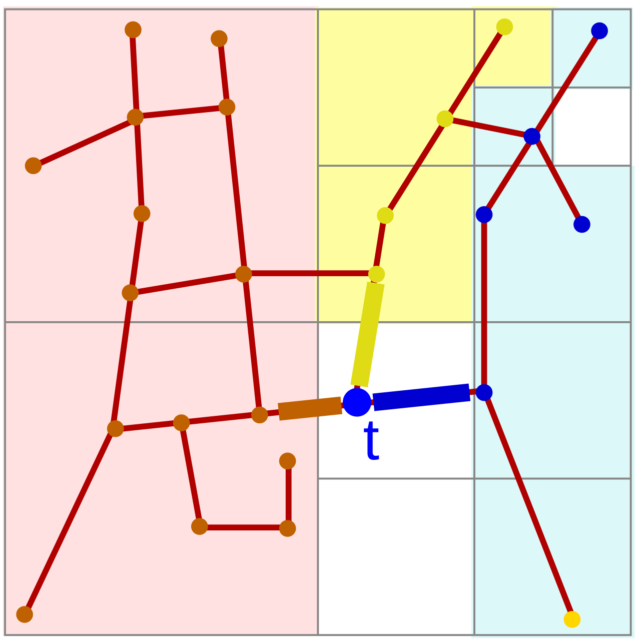


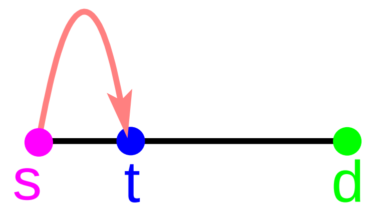


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

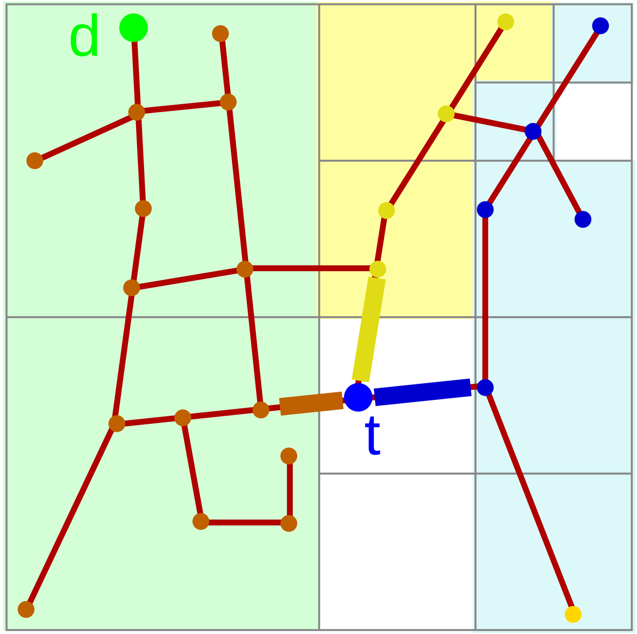


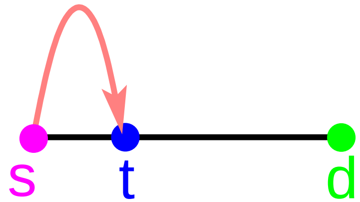


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s
- Retrieve the shortest-path quadtree Q_t corresponding to t

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

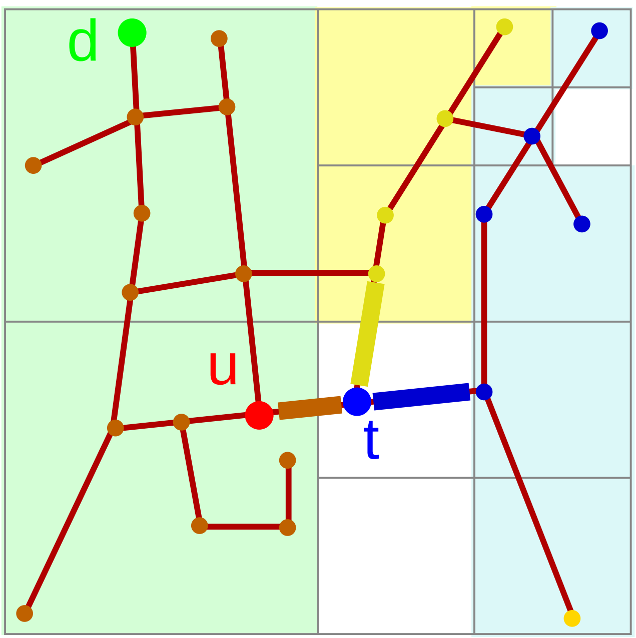


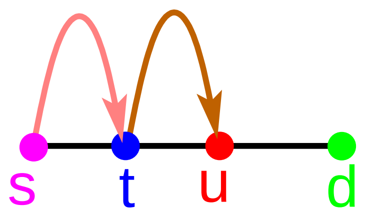


  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s
  - Retrieve the vertex t connected to s in the region containing d in Q_s
  - Retrieve the shortest-path quadtree Q_t corresponding to t
  - Find the colored region that contains d in Q_t

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

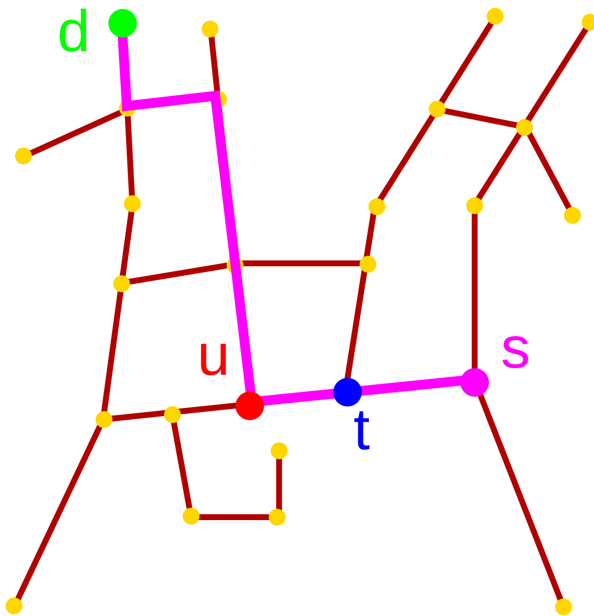


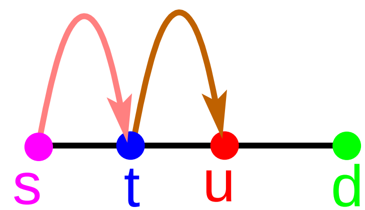


- Retrieve the shortest-path quadtree Q_s corresponding to s
- Find the colored region that contains d in Q_s
- Retrieve the vertex t connected to s in the region containing d in Q_s
- Retrieve the shortest-path quadtree Q_t corresponding to t
- Find the colored region that contains d in Q_t
- Retrieve the vertex u connected to t in the region containing d in Q_t

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?

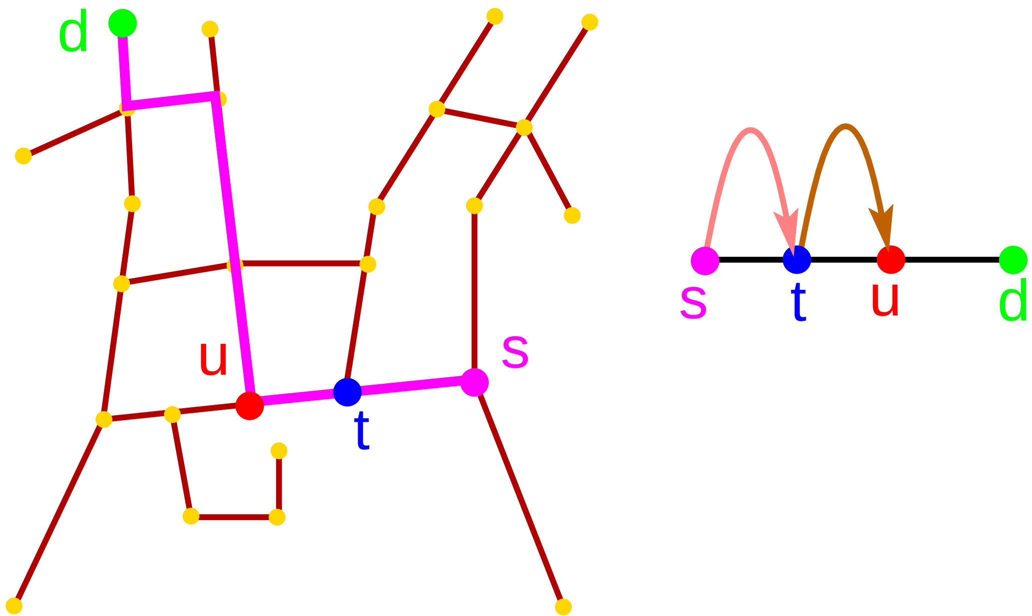




  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s
  - Retrieve the vertex t connected to s in the region containing d in Q_s
  - Retrieve the shortest-path quadtree Q_t corresponding to t
  - Find the colored region that contains d in Q_t
  - Retrieve the vertex u connected to t in the region containing d in Q_t
- Entire shortest path can be retrieved in size-of-path steps

Path Retrieval

- Problem: How to retrieve the shortest path from a source s to a destination d?



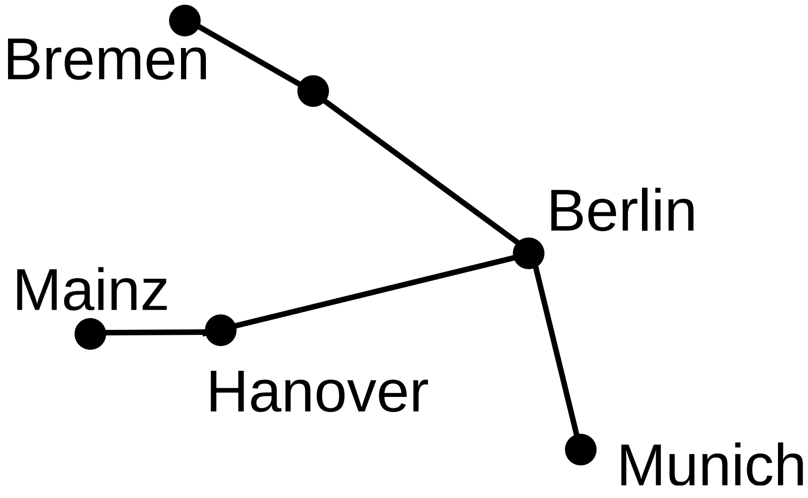
  - Retrieve the shortest-path quadtree Q_s corresponding to s
  - Find the colored region that contains d in Q_s
  - Retrieve the vertex t connected to s in the region containing d in Q_s
  - Retrieve the shortest-path quadtree Q_t corresponding to t
  - Find the colored region that contains d in Q_t
  - Retrieve the vertex u connected to t in the region containing d in Q_t
- Entire shortest path can be retrieved in size-of-path steps
- Network distance between s and d is immediately obtained from shortest path

Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives

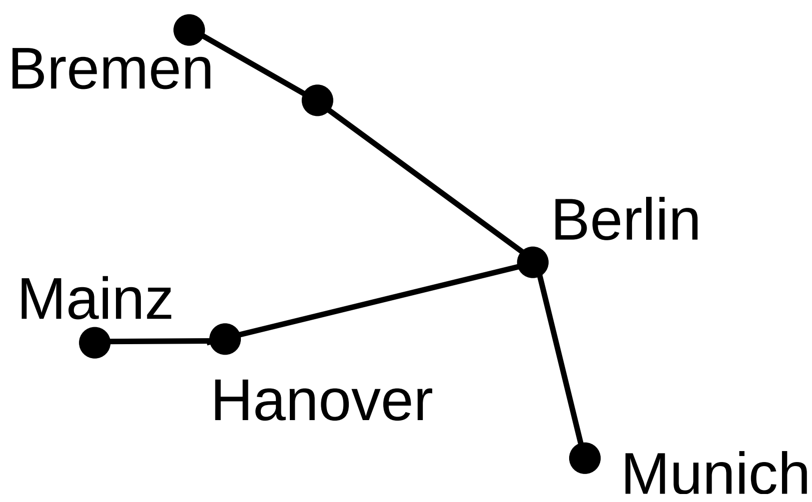
Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



Progressive Refinement of Distances

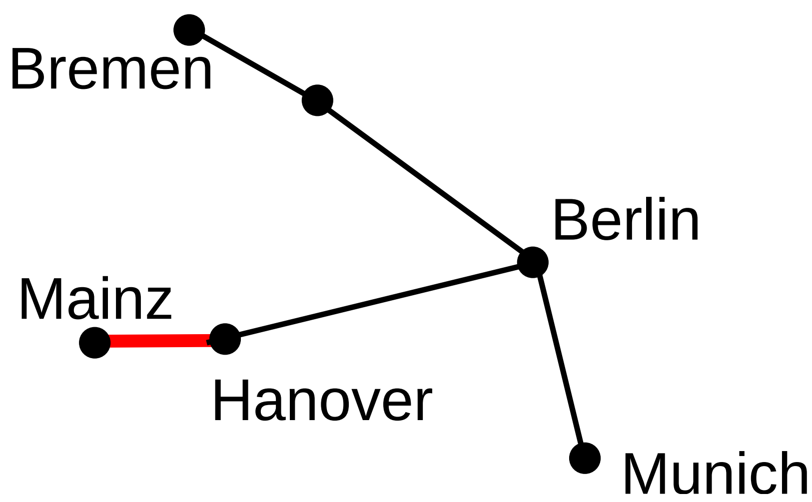
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]

Progressive Refinement of Distances

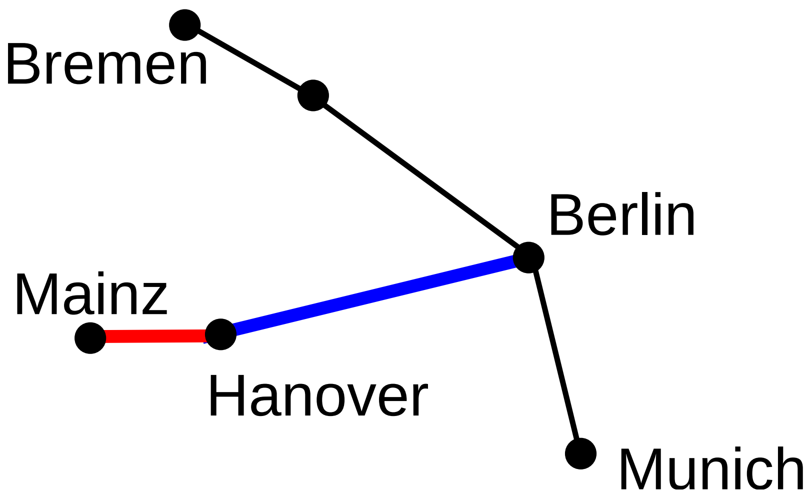
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]

Progressive Refinement of Distances

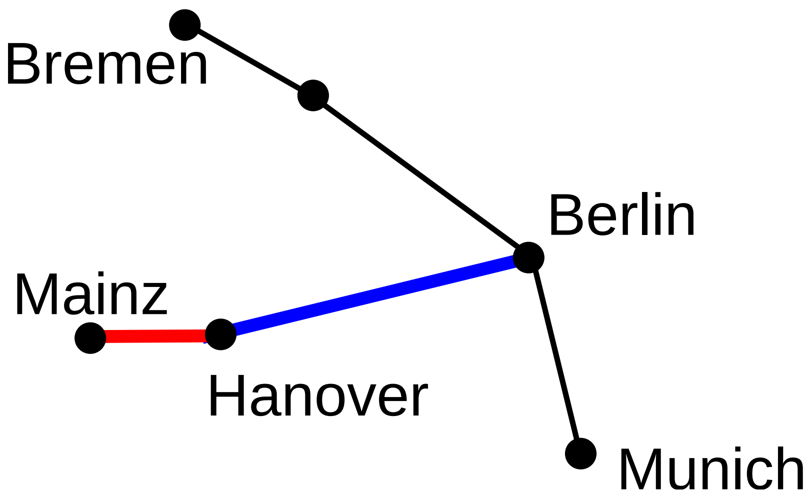
- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]
Berlin	[13,15]	[18,19]

Progressive Refinement of Distances

- Avoid full shortest path retrievals using progressive refinement
- Idea: Use distance intervals instead of the exact distance
- Progressive refinement: *Improve* interval if query cannot be answered
  - Associate Min/Max distance information with each Morton block
  - Refinement involves finding the next link in the shortest path
  - Worst case: retrieve entire shortest path to answer query
  - Many queries require distance comparison primitives
- Example: Is Munich closer to Mainz than Bremen?



	Munich	Bremen
Mainz	[10,20]	[15,30]
Hanover	[12,18]	[17,20]
Berlin	[13,15]	[18,19]

- Munich is closer as distance interval via Berlin does not intersect distance interval to Bremen via Berlin

Outline

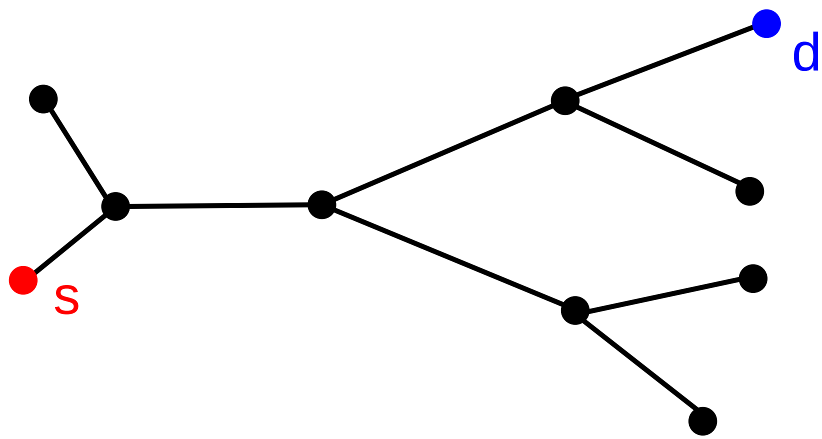
1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

Foundations of k Nearest Neighbor Finding (kNN)

- A non-incremental best-first algorithm
  - Set of objects (with spatial information)
  - A spatial data structure (e.g., a quadtree or R-tree) on objects
  - Shortest-path quadtrees for the spatial network
  - Note decoupling of data (objects) from domain (spatial network)
  - Cost Justification for Precomputing: Provision to reuse computations across queries and across datasets
- Primitive operations using Progressive Refinement

Foundations of k Nearest Neighbor Finding (kNN)

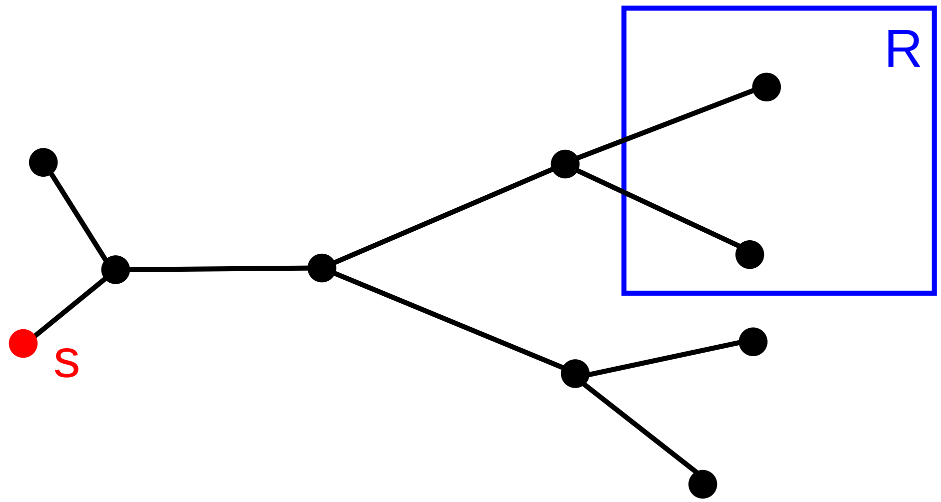
- A non-incremental best-first algorithm
  - Set of objects (with spatial information)
  - A spatial data structure (e.g., a quadtree or R-tree) on objects
  - Shortest-path quadtrees for the spatial network
  - Note decoupling of data (objects) from domain (spatial network)
  - Cost Justification for Precomputing: Provision to reuse computations across queries and across datasets
- Primitive operations using Progressive Refinement



  - DISTANCE_INTERVAL(object,object)

Foundations of k Nearest Neighbor Finding (kNN)

- A non-incremental best-first algorithm
  - Set of objects (with spatial information)
  - A spatial data structure (e.g., a quadtree or R-tree) on objects
  - Shortest-path quadtrees for the spatial network
  - Note decoupling of data (objects) from domain (spatial network)
  - Cost Justification for Precomputing: Provision to reuse computations across queries and across datasets
- Primitive operations using Progressive Refinement



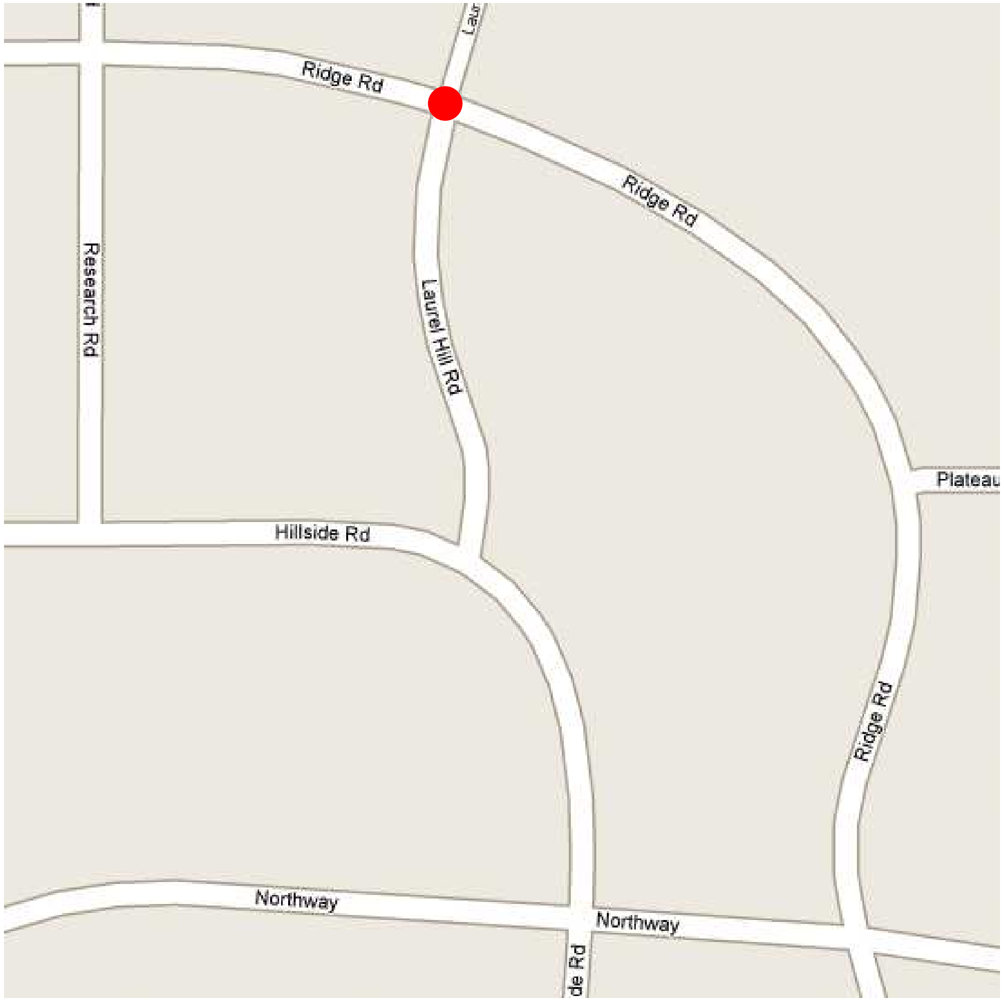
  - DISTANCE_INTERVAL(object,object)
  - DISTANCE_INTERVAL(object,Region)

Type of Input Objects

- Types of objects on spatial networks

Type of Input Objects

- Types of objects on spatial networks
  - Vertex object

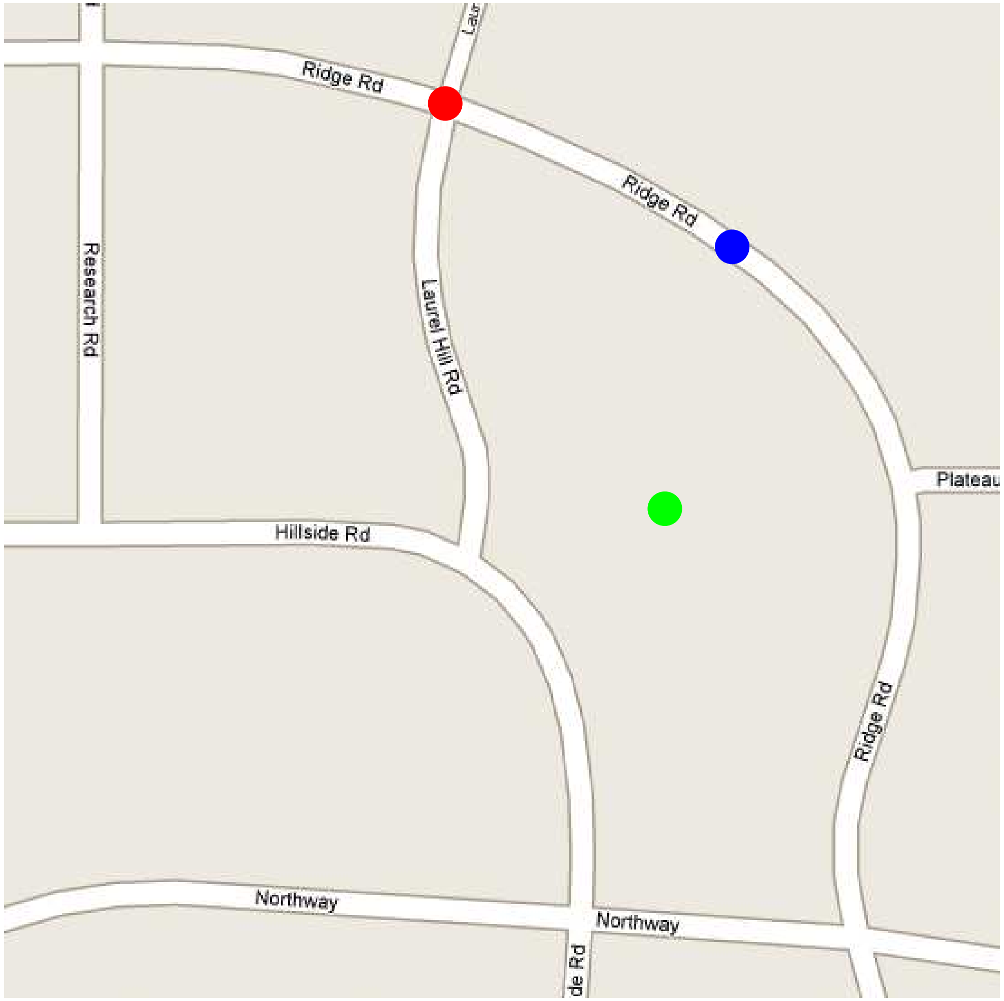


Type of Input Objects

- Types of objects on spatial networks
  - Vertex object
  - Edge object

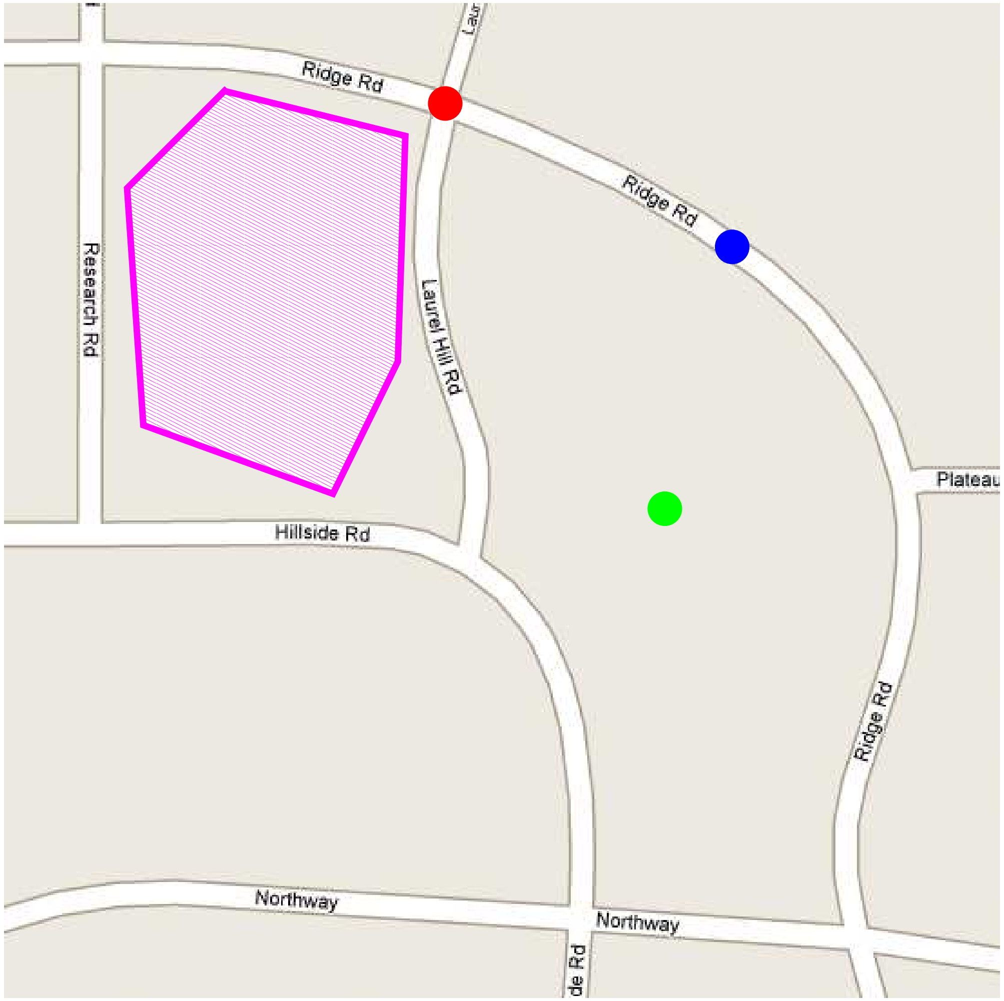
Type of Input Objects

- Types of objects on spatial networks
  - Vertex object
  - Edge object
  - Face object



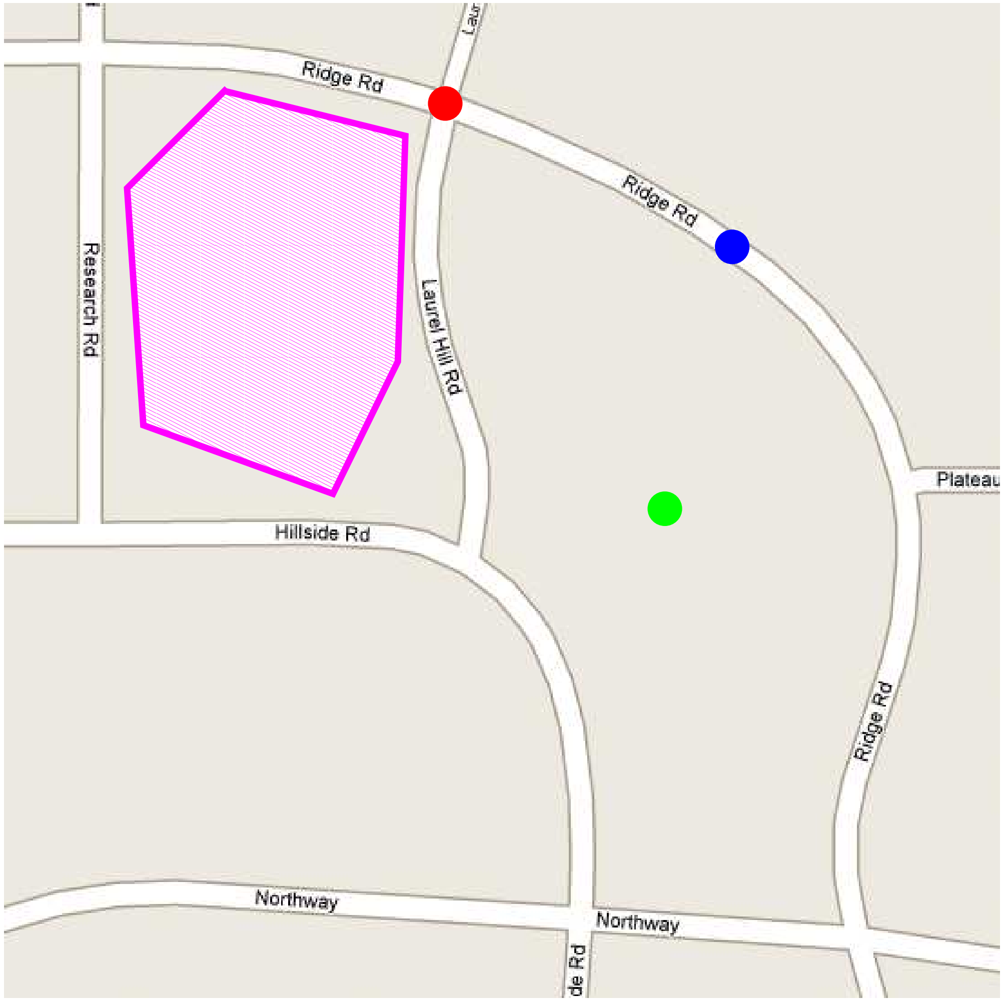
Type of Input Objects

- Types of objects on spatial networks
  - Vertex object
  - Edge object
  - Face object
  - Object with extents



Type of Input Objects

- Types of objects on spatial networks
  - Vertex object
  - Edge object
  - Face object
  - Object with extents
  - Any combination of the above



Properties of kNN Algorithm

- Neighbors produced in increasing order of distance from q
- Use a priority queue Q of objects and blocks
- Q contains network distance interval $[\delta^-, \delta^+]$ of objects from q
- Additional information stored with each object o in Q
  1. An intermediate vertex u in shortest path from q to u
  2. network distance d from q to u
- Uses another priority queue L in addition to Q
  - Stores k objects found so far in increasing order of δ^+
  - D_k is the maximum of the distance interval of the $k\mathrm{th}$ element in L
  - Idea: Prune elements e from Q such that $\delta_e^- \geq D_k$
- Elements are removed from Q in increasing order of the minimum of their distance interval δ^- from q
  - Objects may be reinserted in Q if $\delta^- < D_k$
  - Terminate when $\delta^- \geq D_k$
- Advantages over Incremental best-first kNN (INN)
  - Smaller size of Q
  - Faster than INN

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:

kNN Algorithm

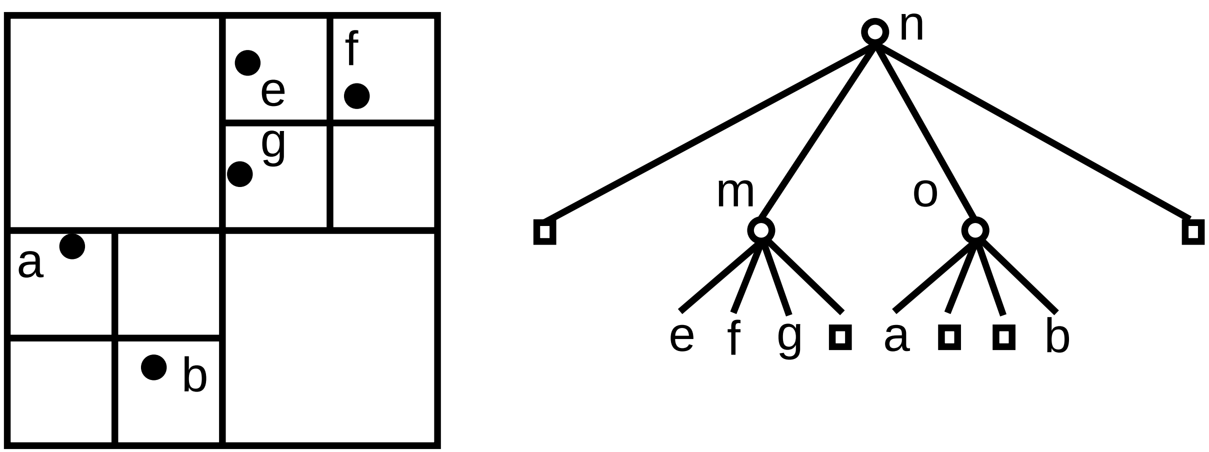
1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:
     - Remove p from L if $\delta^+ \leq D_k$
     - Apply refinement to improve distance interval of p and reinsert p in L if $\delta^+ \leq D_k$ and in Q if $\delta^- < D_k$ and go to Step 2
   - No collision:

kNN Algorithm

1. Initialize priority queue Q by inserting the root T
2. Retrieve top element p in Q at each iteration and halt if minimum distance from q is $> D_k$
3. If p is a LEAF block, then replace it with all objects contained within it for which $\delta^- < D_k$ along with their network distance interval from q
   - Also enqueue objects in L if $\delta^+ < D_k$
4. If p is a NONLEAF block, then replace it with all its children blocks for which the minimum distance from q is $< D_k$
5. If p is an OBJECT, then test the distance interval of p for possible collisions with the current top element of Q
   - A collision occurs if the distance interval of p intersects the distance interval of the current top element in Q
   - Collision:
     - Remove p from L if $\delta^+ \leq D_k$
     - Apply refinement to improve distance interval of p and reinsert p in L if $\delta^+ \leq D_k$ and in Q if $\delta^- < D_k$ and go to Step 2
   - No collision: p is already one of k nearest neighbors in L (Theorem 1) and go to Step 2

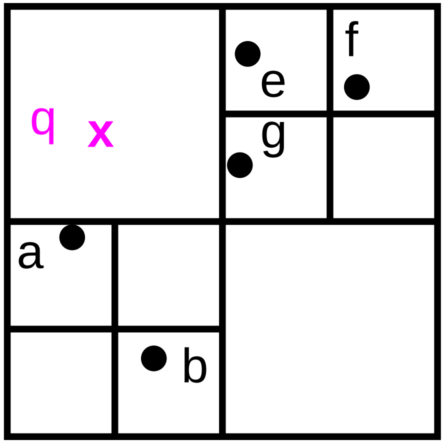
Example of an Non-incremental k Neighbor Search

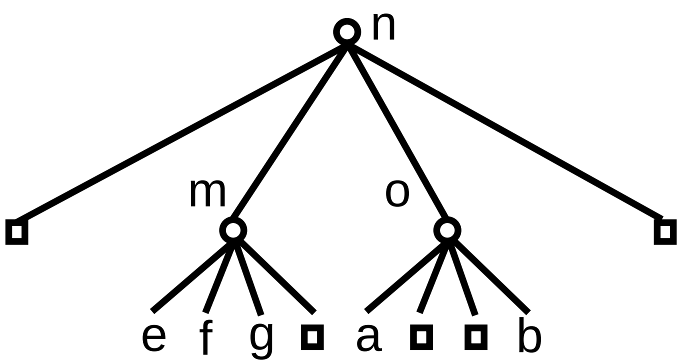
k = 2



Example of an Non-incremental k Neighbor Search

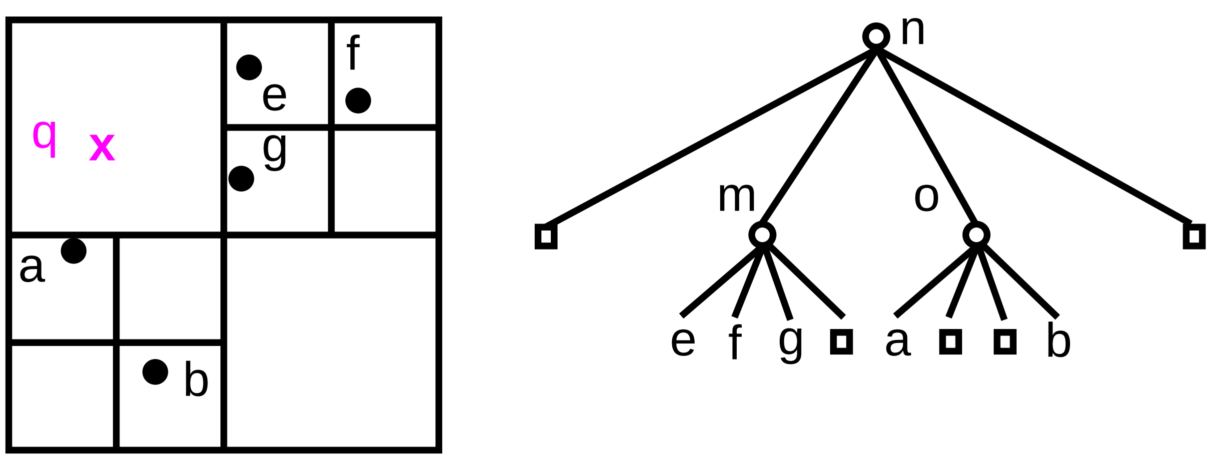
k = 2





Example of an Non-incremental k Neighbor Search

k = 2

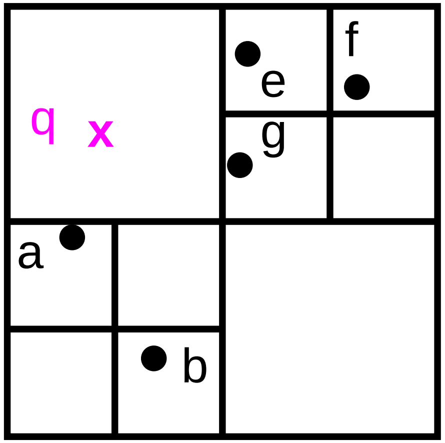


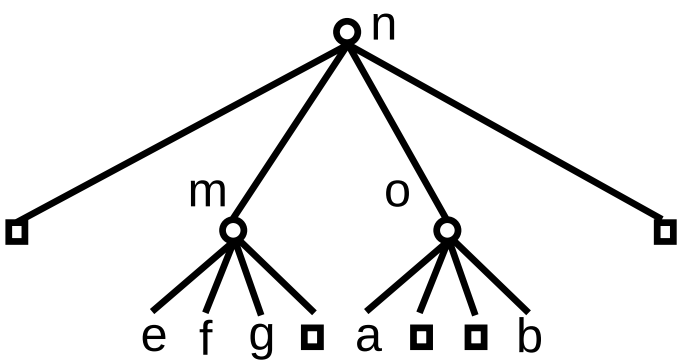
front

L Queue

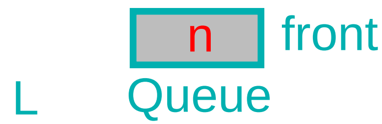
Example of an Non-incremental k Neighbor Search

k = 2



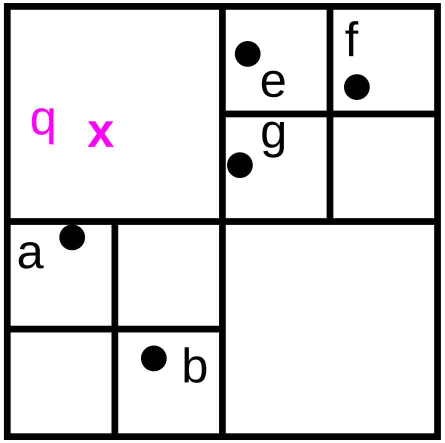


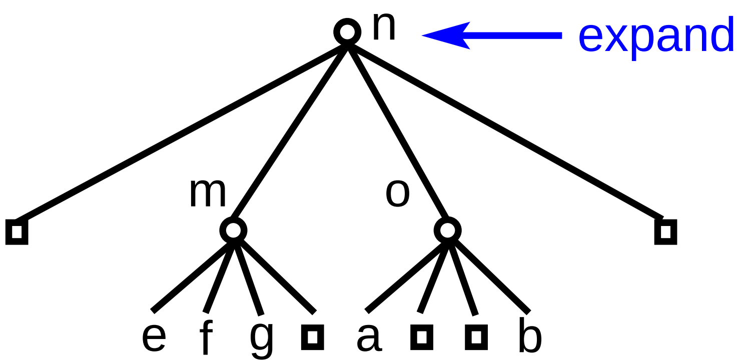
1. Insert n into Queue.



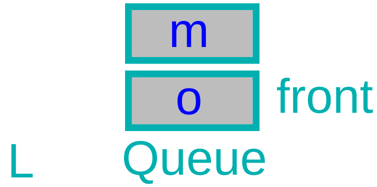
Example of an Non-incremental k Neighbor Search

k = 2



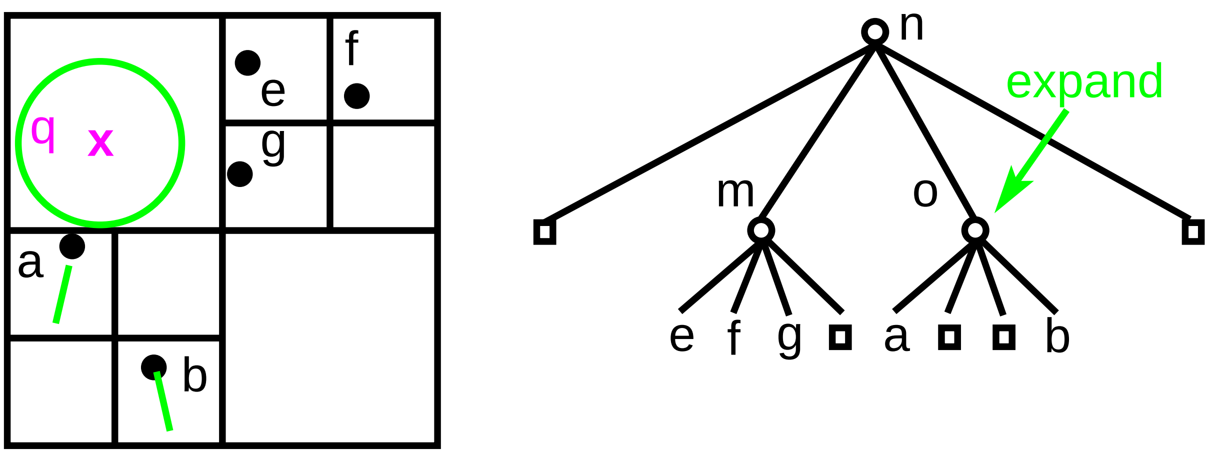


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.

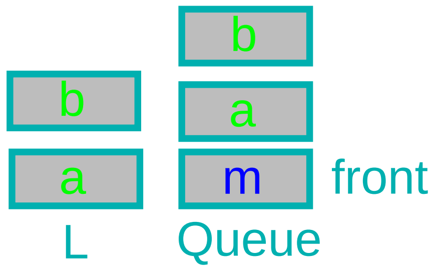


Example of an Non-incremental k Neighbor Search

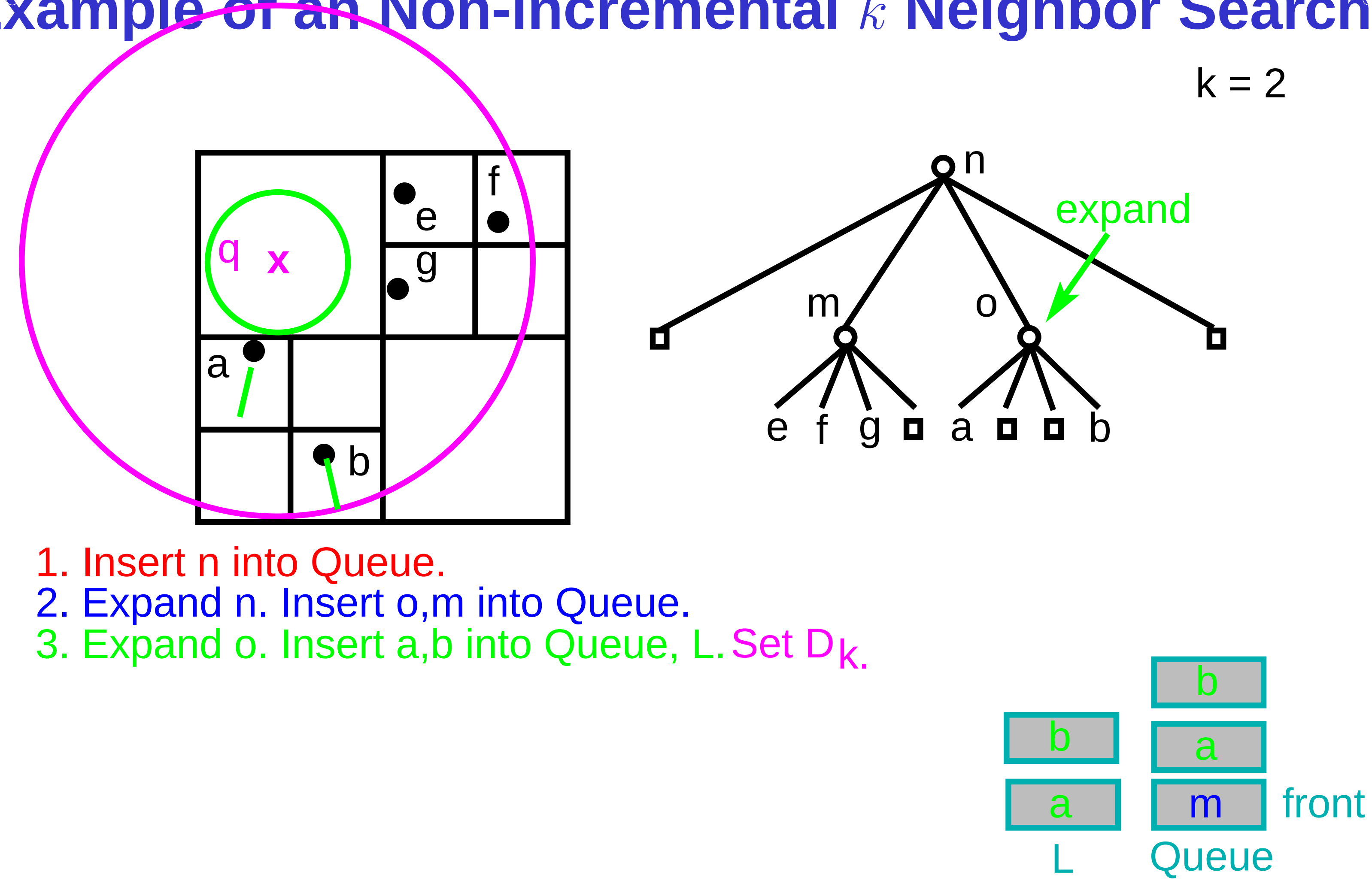
k = 2



1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L.



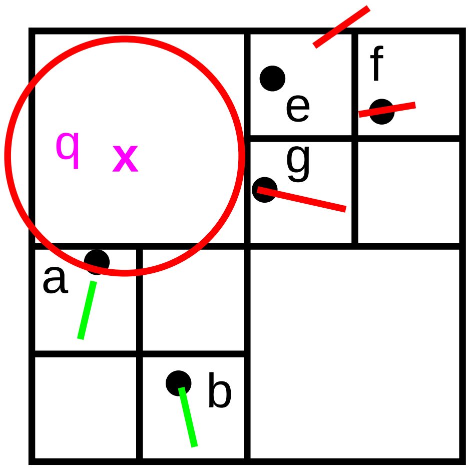
Example of an Non-incremental k Neighbor Search

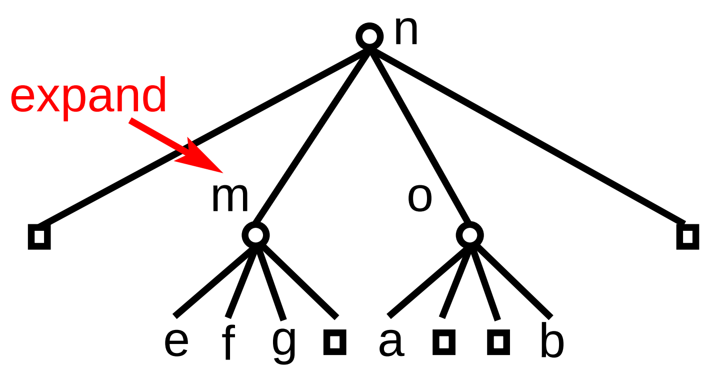


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.

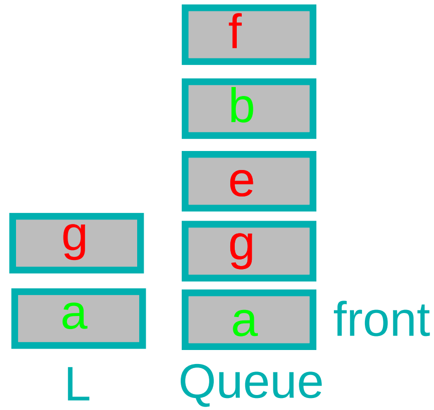
Example of an Non-incremental k Neighbor Search

k = 2



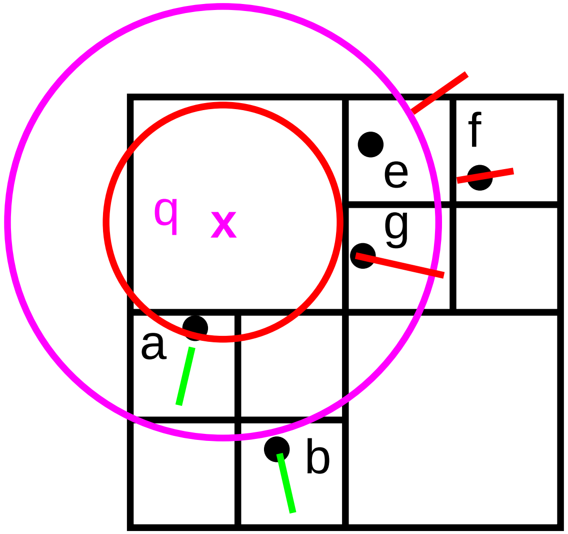


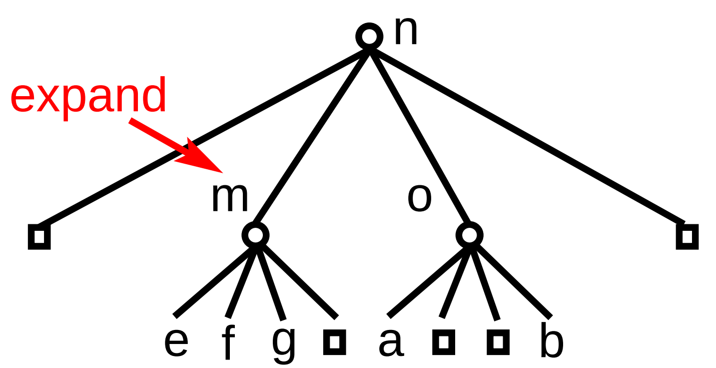
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.



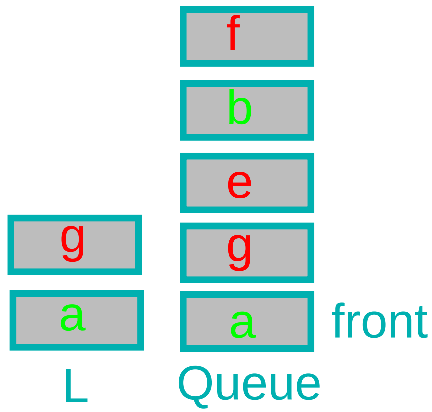
Example of an Non-incremental k Neighbor Search

k = 2



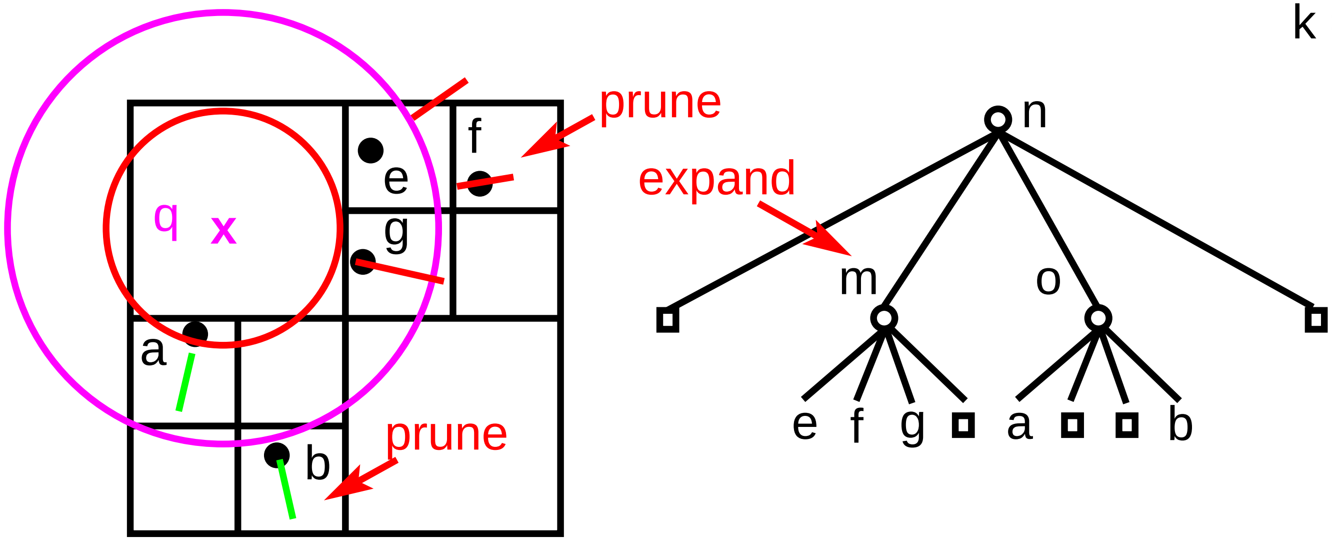


1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k.

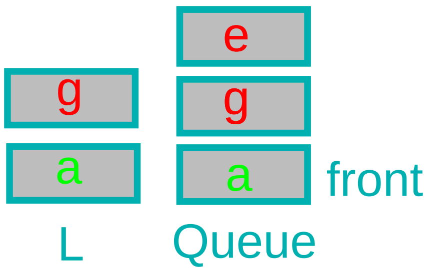


Example of an Non-incremental k Neighbor Search

k = 2

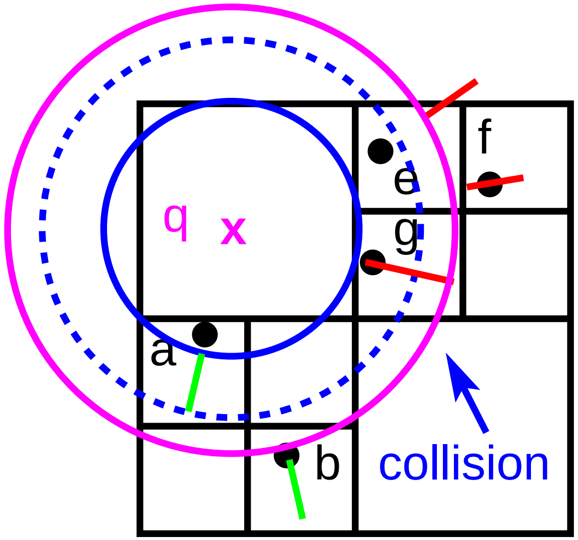


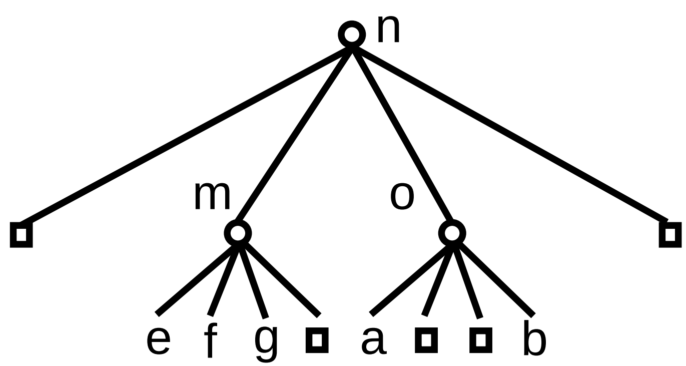
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.



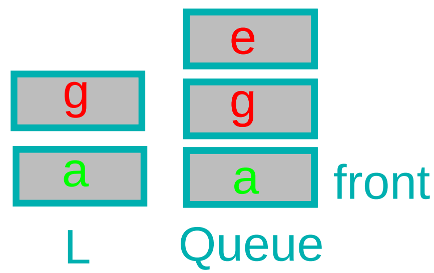
Example of an Non-incremental k Neighbor Search

k = 2



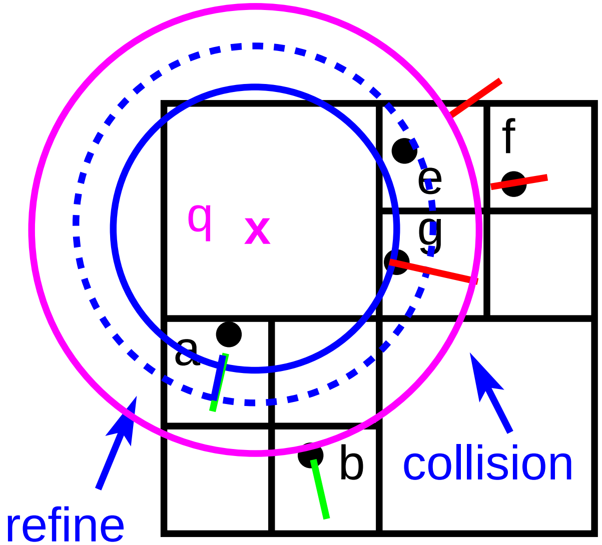


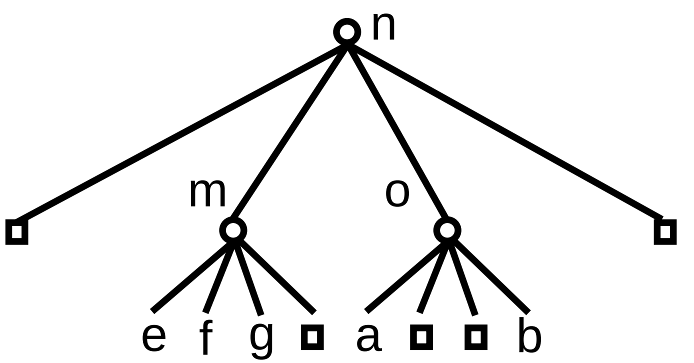
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.



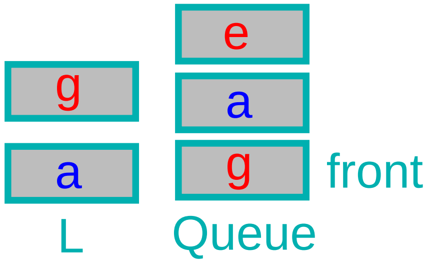
Example of an Non-incremental k Neighbor Search

k = 2



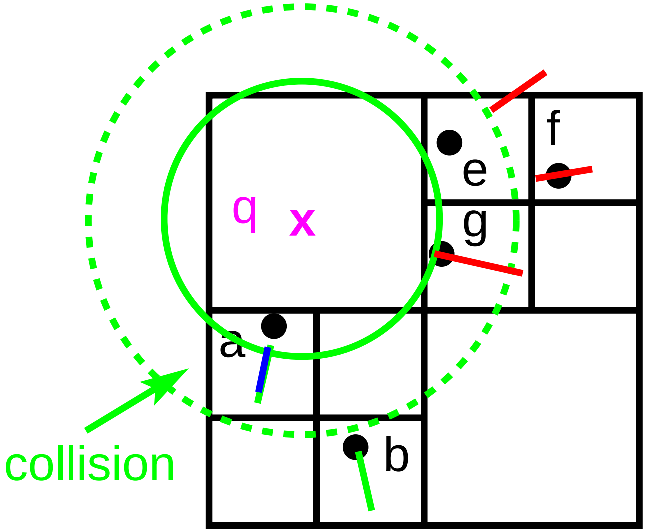


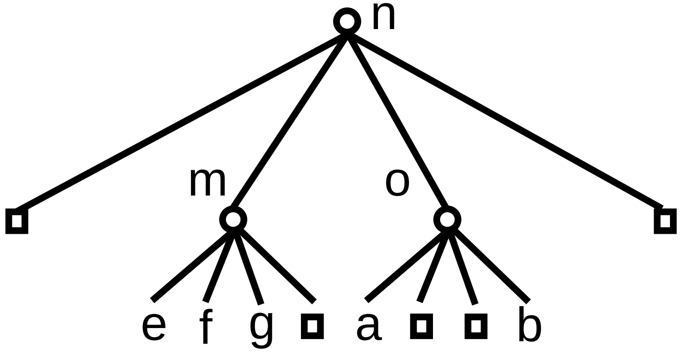
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.



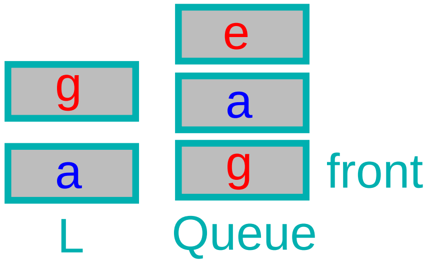
Example of an Non-incremental k Neighbor Search

k = 2



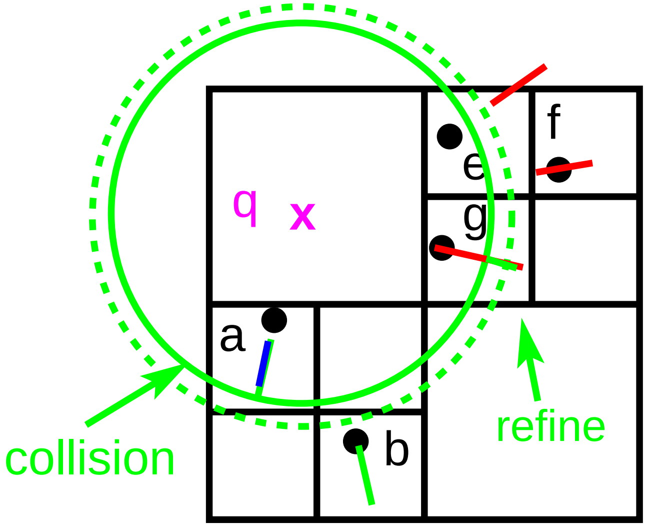


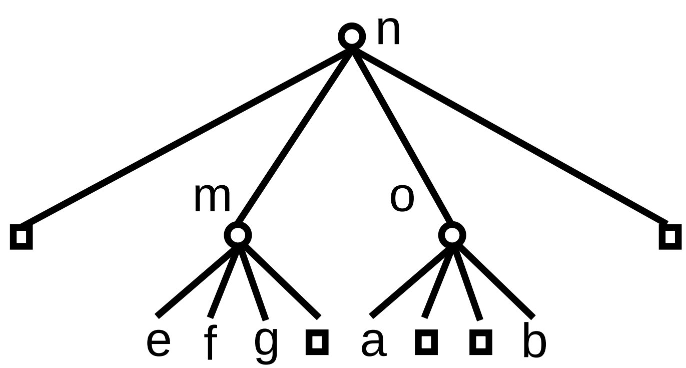
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.



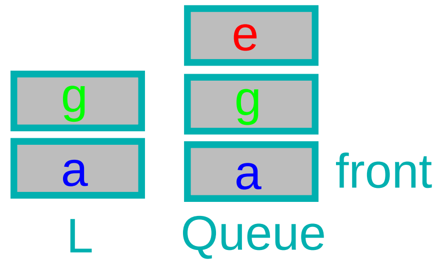
Example of an Non-incremental k Neighbor Search

k = 2



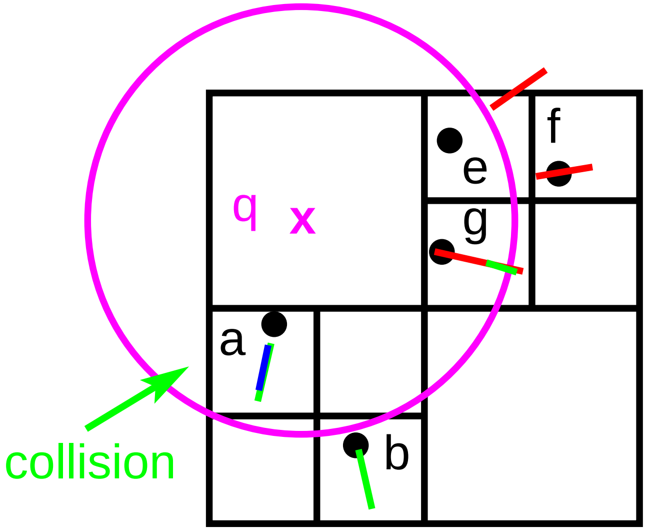


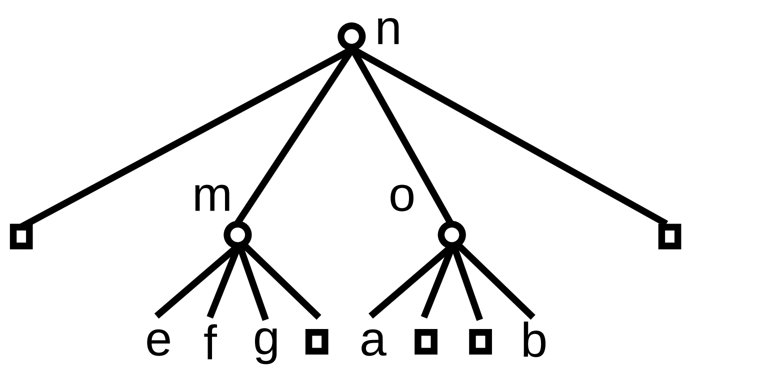
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L.



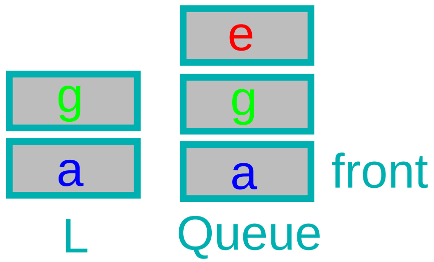
Example of an Non-incremental k Neighbor Search

k = 2



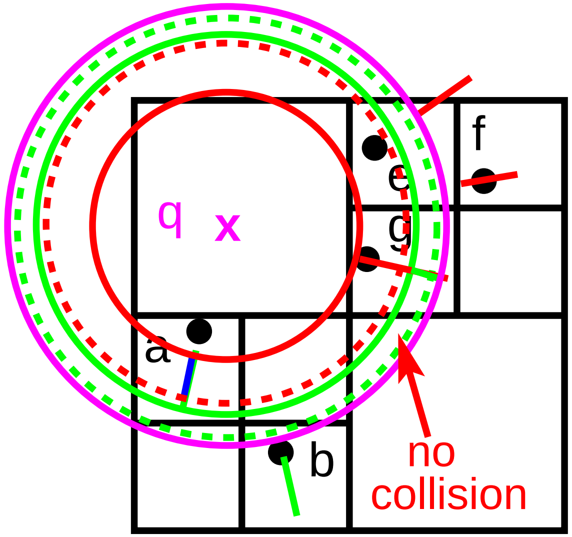


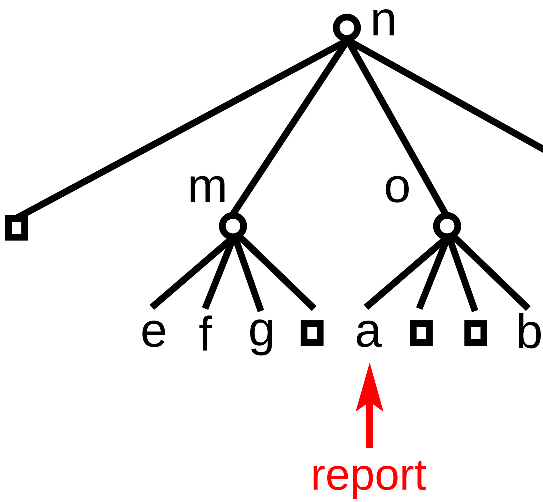
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.



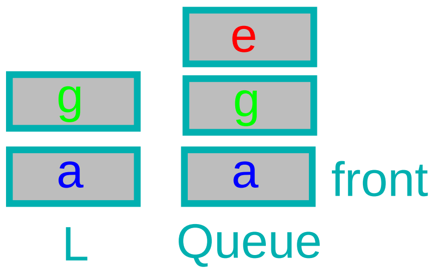
Example of an Non-incremental k Neighbor Search

k = 2

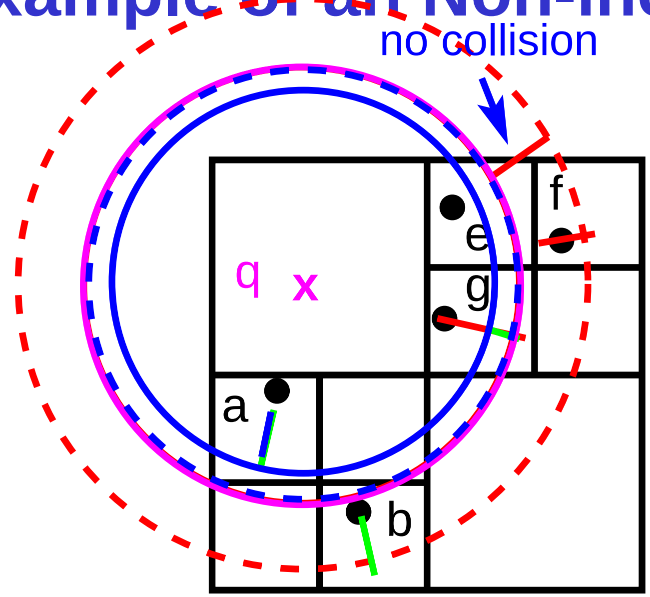




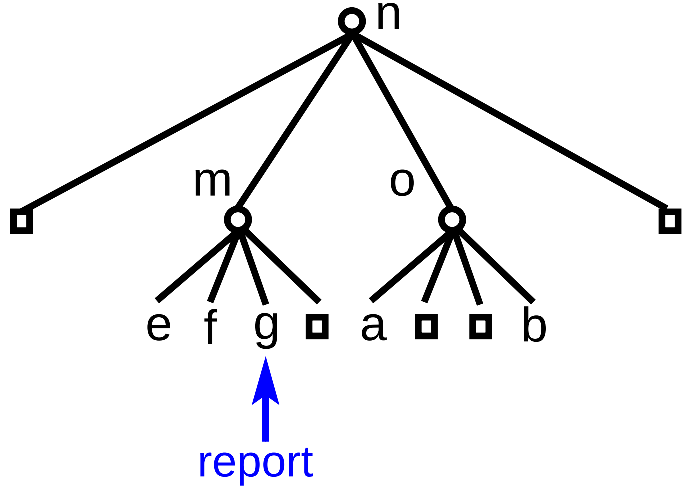
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.



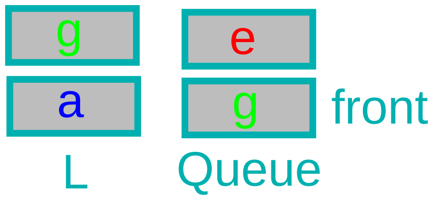
Example of an Non-incremental k Neighbor Search



k = 2

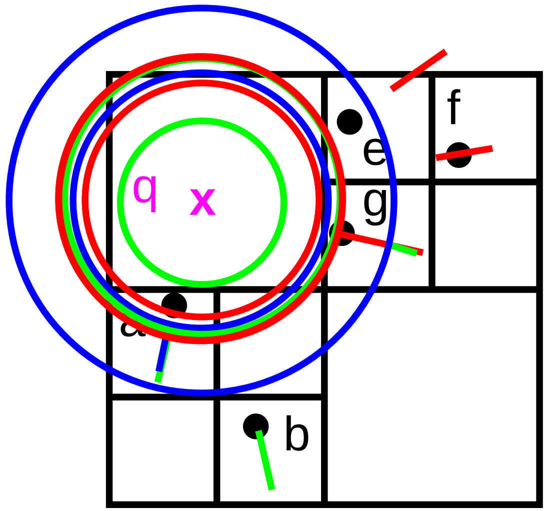


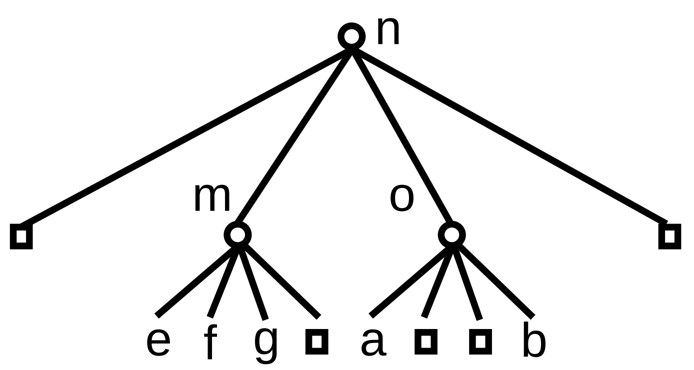
1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.
8. Process g. No collision of g with e.
   No need to refine g further. Report L.



Example of an Non-incremental k Neighbor Search

k = 2





1. Insert n into Queue.
2. Expand n. Insert o,m into Queue.
3. Expand o. Insert a,b into Queue, L. Set D_k.
4. Expand m. Insert g,e,f into Queue and g into L.
   Update D_k. Prune f and b from Queue.
5. Process a. Collision of a with g.
   Refine a. Reinsert a into Queue and L.
6. Process g. Collision of g with a.
   Refine and Reinsert g into Queue and L. Update D_k.
7. Process a. No collision of a with g. No need to refine a further.
8. Process g. No collision of g with e.
   No need to refine g further. Report L.
   Example of a best−first nearest neighbor algorithm.
   (Search radius to first element in Queue)

front

L Queue

Other Nearest Neighbor Methods

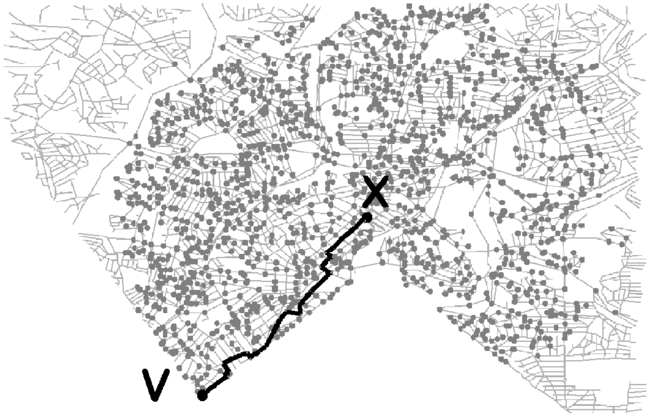
1. IER ("Incremental" Euclidean Restriction") method [Papa03]: not an incremental network distance algorithm
   - Use incremental nearest neighbor algorithm to find k nearest neighbors using Euclidean distance
   - Find the network distance of these k nearest neighbors using Dijkstra's algorithm and sort in increasing order
   - Apply incremental nearest neighbor algorithm using Euclidean distance until obtaining an object whose Euclidean distance is greater than the current network distance to the $k\mathrm{th}$ nearest neighbor
   - Need to apply Dijkstra's algorithm to obtain network distance to each additional object until termination
2. INE ("Incremental" Network Expansion) method [Papa03]: k-nearest neighbor network distance algorithm
   - Really Dijkstra's algorithm with a buffer L containing the k nearest neighbors seen so far in terms of network distance
   - Halt: current neighbor is farther than current k nearest neighbors in L
3. Advantage of our method is that Dijkstra's algorithm is only applied once per vertex in building the shortest-path quadtrees regardless of the number of queries instead of once for each query

Worst Case Comparison between INE and kNN

- Given a set of objects S on a spatial network

Worst Case Comparison between INE and kNN

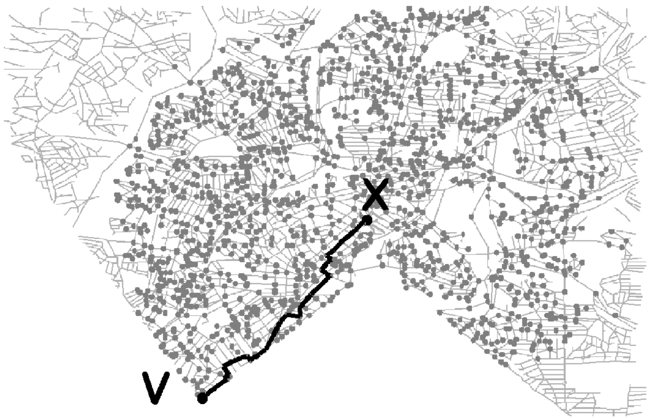
- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\mathrm{th}$ nearest neighbor



INE

Worst Case Comparison between INE and kNN

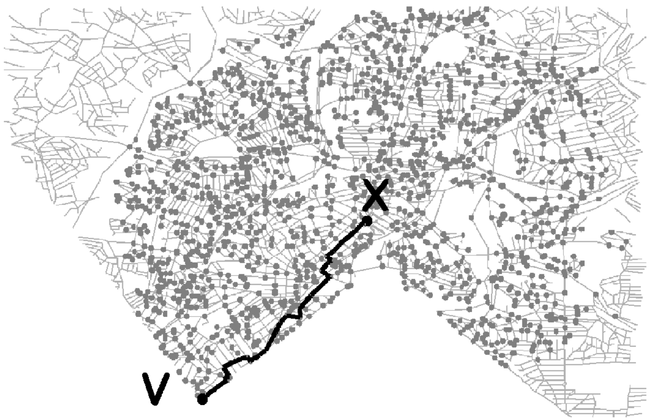
- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\text{th}$ nearest neighbor
    - INE visits every edge e that is closer to q than the $k\text{th}$ nearest neighbor



INE

Worst Case Comparison between INE and kNN

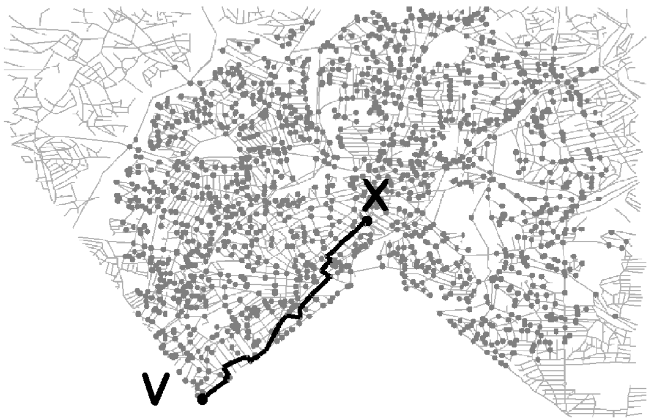
- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\mathrm{th}$ nearest neighbor
    - INE visits every edge e that is closer to q than the $k\mathrm{th}$ nearest neighbor
    - Number of queries to the spatial index is $O(M)$, which can be large



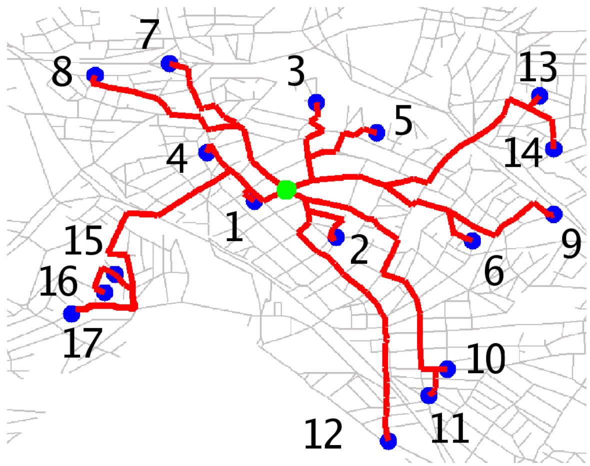
INE

Worst Case Comparison between INE and kNN

- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\mathrm{th}$ nearest neighbor
    - INE visits every edge e that is closer to q than the $k\mathrm{th}$ nearest neighbor
    - Number of queries to the spatial index is $O(M)$, which can be large
  - kNN's worst case is proportional to the number of objects examined and the number of links on the shortest paths to them from the query object q



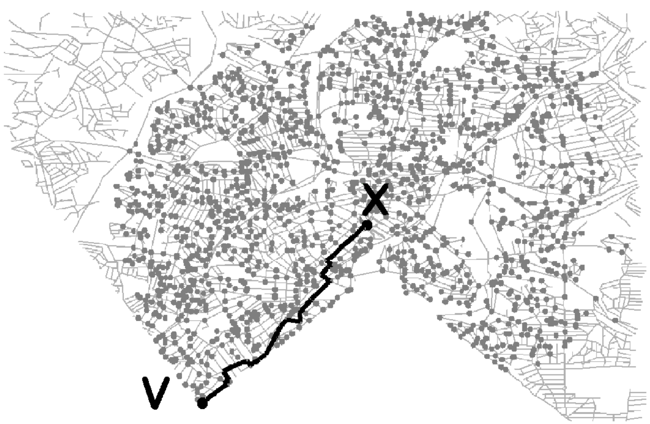
INE



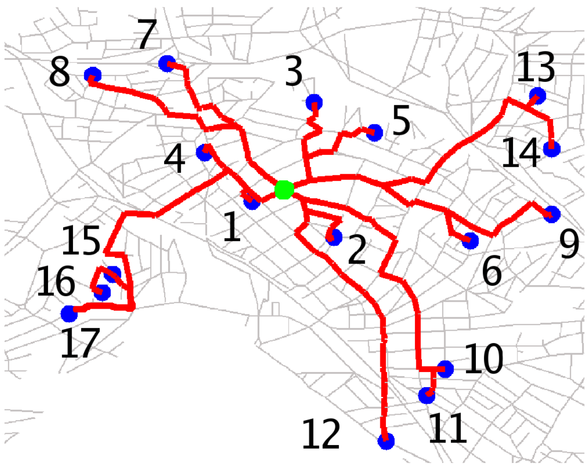
kNN

Worst Case Comparison between INE and kNN

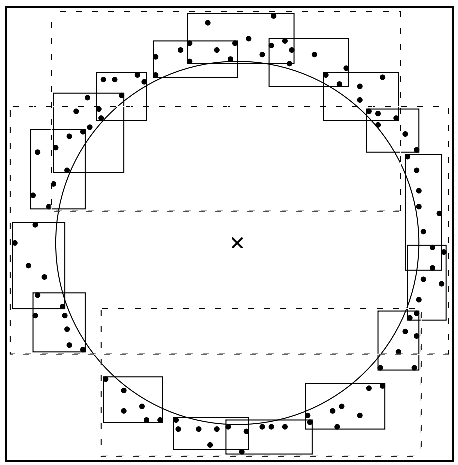
- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\mathrm{th}$ nearest neighbor
    - INE visits every edge e that is closer to q than the $k\mathrm{th}$ nearest neighbor
    - Number of queries to the spatial index is $O(M)$, which can be large
  - kNN's worst case is proportional to the number of objects examined and the number of links on the shortest paths to them from the query object q
    - kNN's worst case occurs when data objects are all nearly equidistant from query object



INE



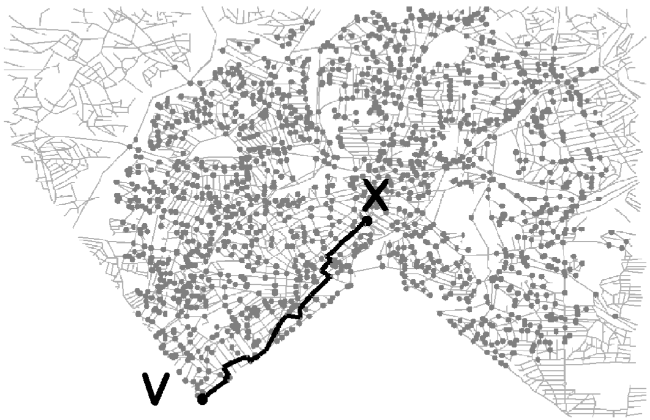
kNN



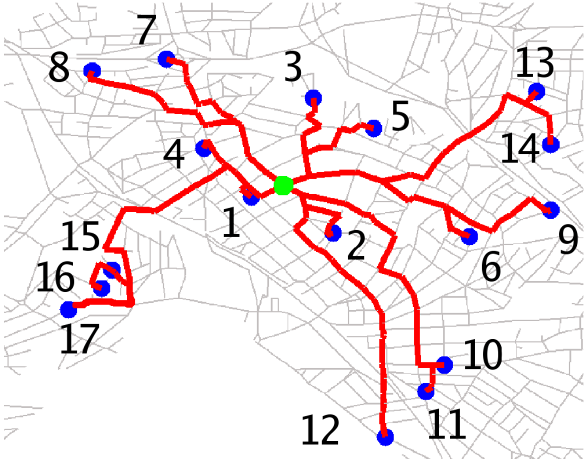
kNN

Worst Case Comparison between INE and kNN

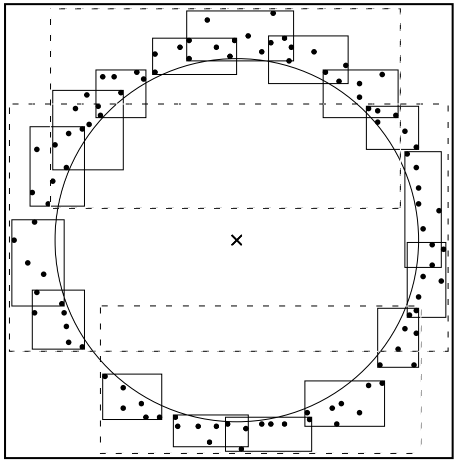
- Given a set of objects S on a spatial network
  - INE's worst case depends on the distance to the $k\mathrm{th}$ nearest neighbor
    - INE visits every edge e that is closer to q than the $k\mathrm{th}$ nearest neighbor
    - Number of queries to the spatial index is $O(M)$, which can be large
  - kNN's worst case is proportional to the number of objects examined and the number of links on the shortest paths to them from the query object q
    - kNN's worst case occurs when data objects are all nearly equidistant from query object
    - Probability of worst case is low, as it depends on a particular configuration of both the data objects and the query object



INE



kNN



kNN

Musings on How Realistic is the Approach

- How about a system for the whole US?
  - 24 million vertices x 10 seconds (say) per shortest path
    - Single machine = 2777 days
    - Google with 0.5 million machines = 480 seconds
    - Modest Cluster of 2000 machines = 1 day, 10 hours
  - Storage shown to be $cN\sqrt{N}$ Morton Blocks
    - N = 24 million vertices, 8 bytes per Morton block, c = 2 from empirical analysis = 1.8 TB
  - Easily Parallelizable: data parallelism
  - Mostly a one-time effort (decoupling)
- Open Challenge: Updates!
  - Changes to spatial network (e.g., road closure)
  - Dynamic traffic information
  - Strategy: How to localize changes to minimize recomputation?
- Approximation Strategies: location based services
  - Shortest-path quadtree on proximal vertices only (say, 100 miles around a vertex)
  - Multiresolution spatial networks
    - Full resolution around a source vertex that gets sparse gradually

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:

Path Coherence Beyond SILC

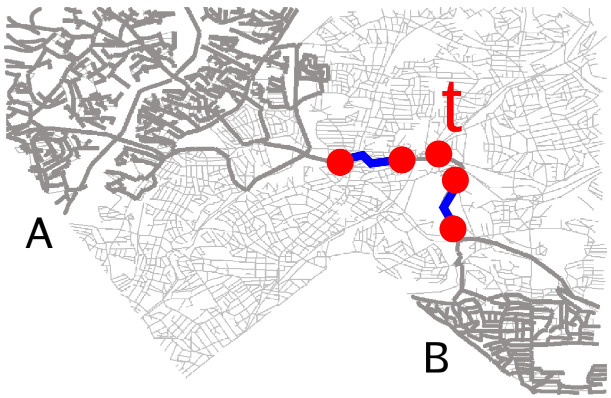
- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:
     - one or more vertices t in common, OR

Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:
     - one or more vertices t in common, OR
     - one or more edges in common

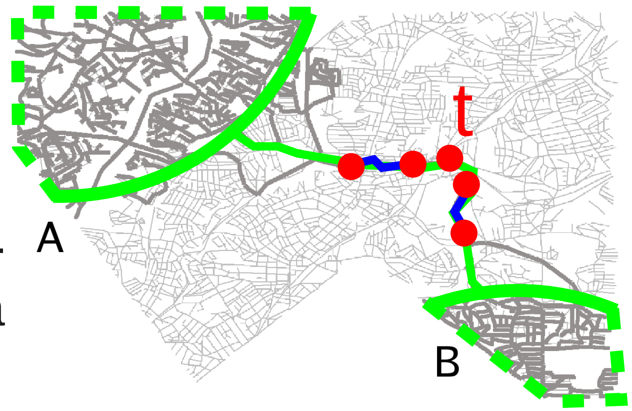
Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:
     - one or more vertices t in common, OR
     - one or more edges in common
  2. Shortest paths have a range of network distance values which can be expressed as a function of some approximation value ϵ



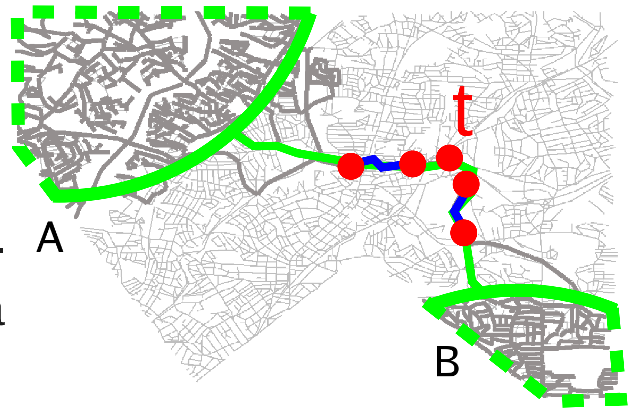
Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:
     - one or more vertices t in common, OR
     - one or more edges in common
  2. Shortest paths have a range of network distance values which can be expressed as a function of some approximation value ϵ
  3. Result has a structure of a dumbbell



Path Coherence Beyond SILC

- SILC framework focussed on facilitating nearest neighbor computation
- Not so efficient for shortest path and network distance as need to refine distances using an iterative process
- SILC captures the path coherence in the shortest paths
  - single source vertex to multiple destination vertices
  - Not captured: multiple source vertices to multiple destination vertices
- Introduce the Path Coherent Pair (PCP) framework
- A PCP is denoted by: (A,B,t)
  1. All shortest paths from A to B have either:
     - one or more vertices t in common, OR
     - one or more edges in common
  2. Shortest paths have a range of network distance values which can be expressed as a function of some approximation value ϵ
  3. Result has a structure of a dumbbell



- Goal: Decompose spatial network into PCPs so that all n^2 shortest paths are captured

Finding Path Coherent Pairs in Spatial Networks

Finding Path Coherent Pairs in Spatial Networks

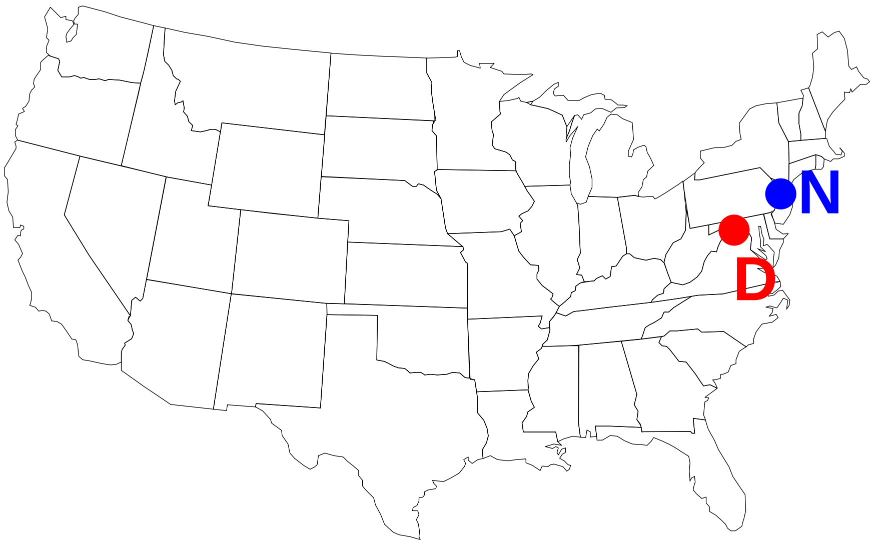
- Source Vertices:

Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D)

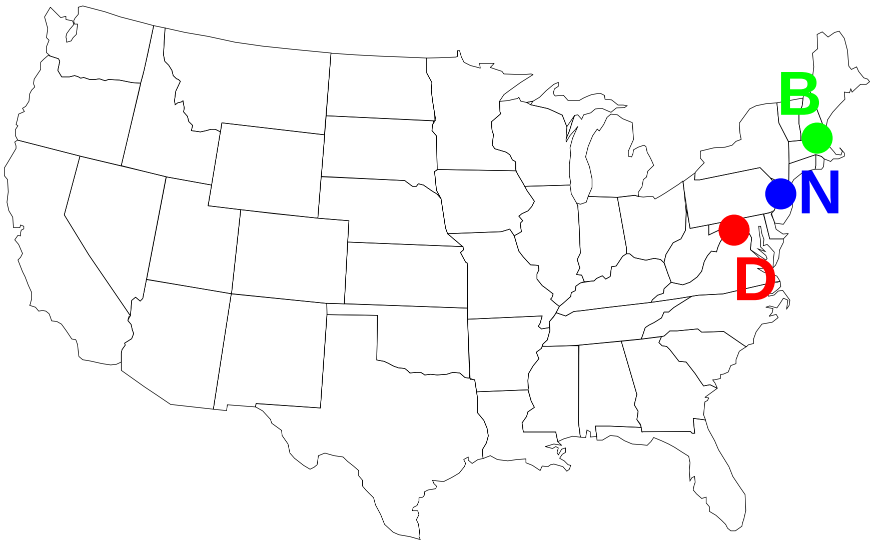
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N)



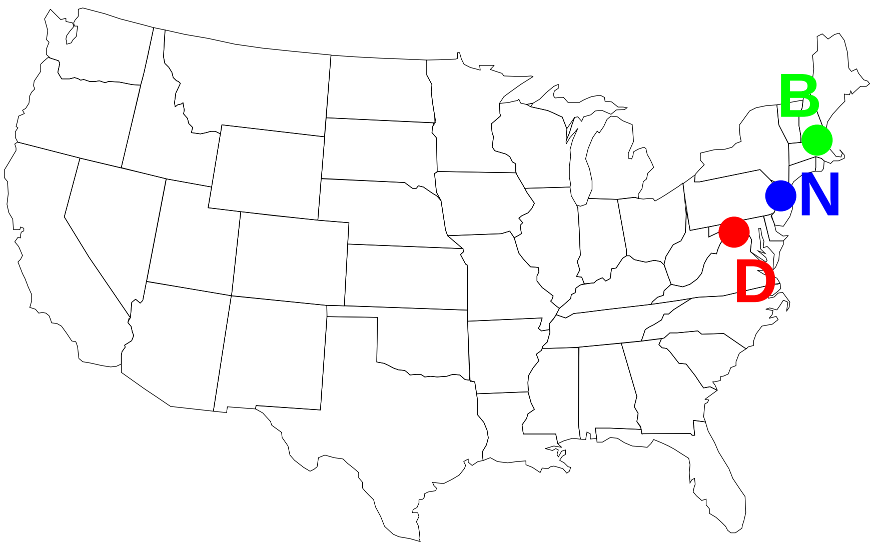
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)



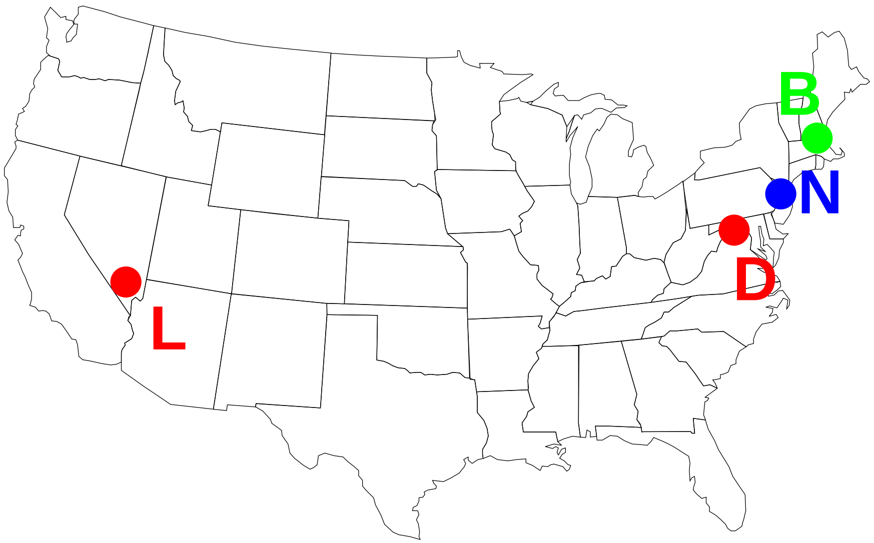
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices:



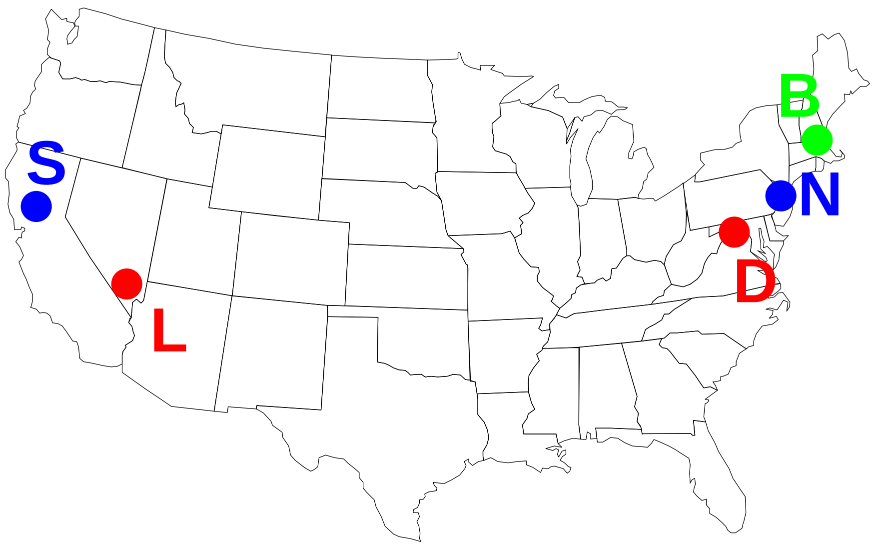
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L)



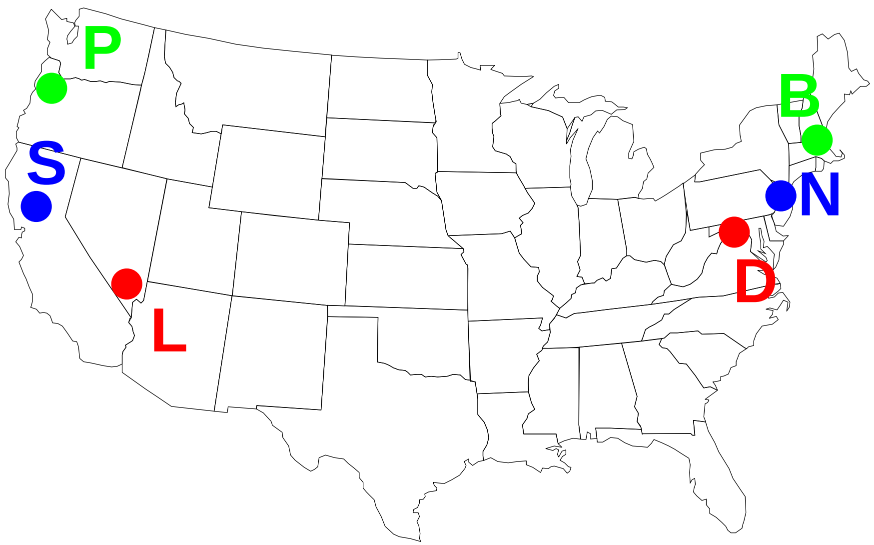
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S)



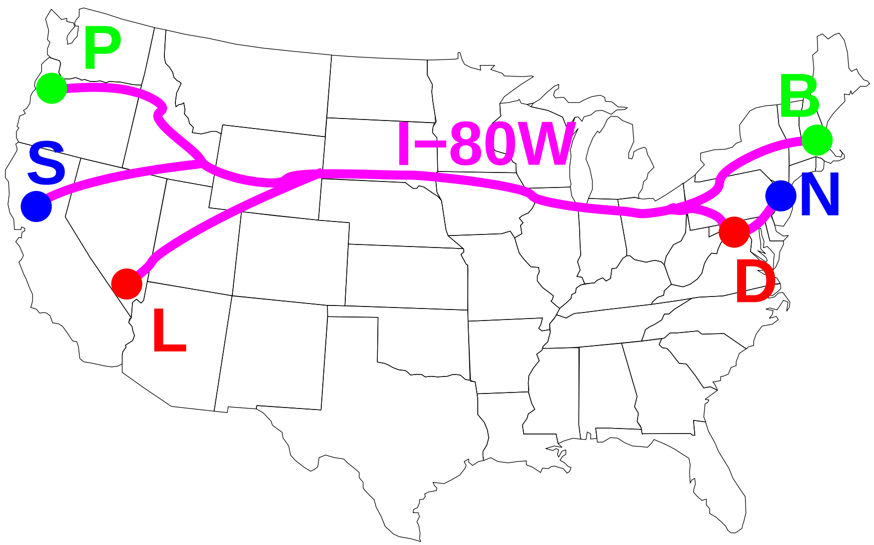
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)



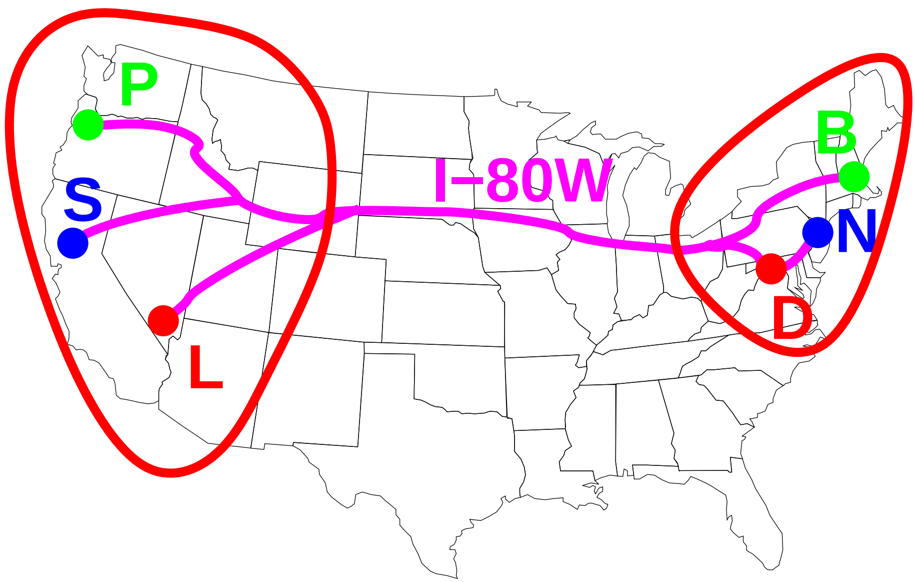
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W



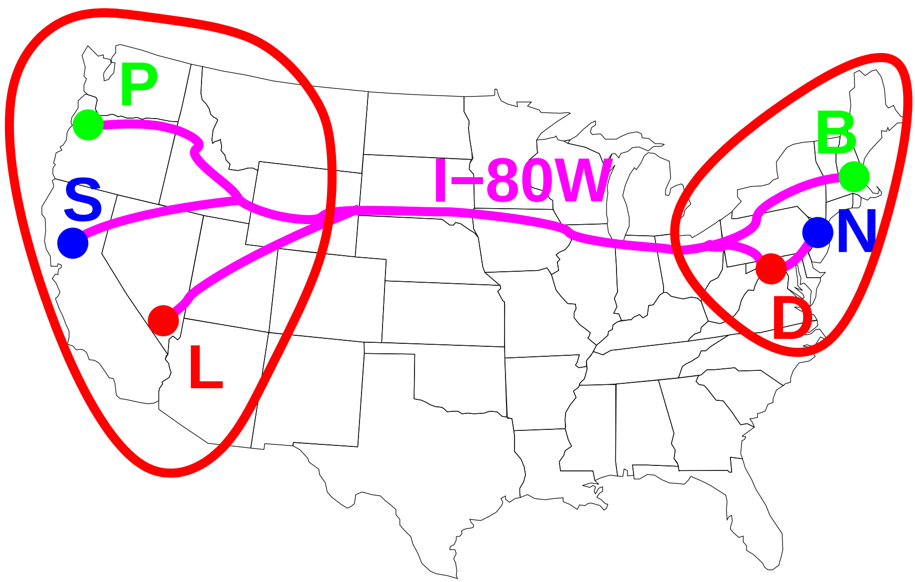
Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage



Finding Path Coherent Pairs in Spatial Networks

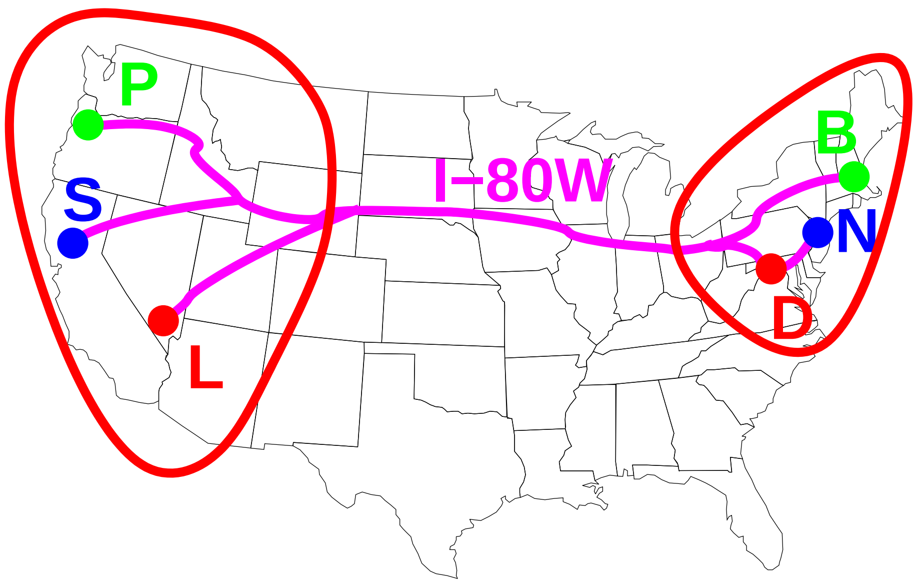
- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage
- Intuition: Sources “sufficiently far” from destinations share common vertices in their shortest paths



Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage
- Intuition: Sources “sufficiently far” from destinations share common vertices in their shortest paths

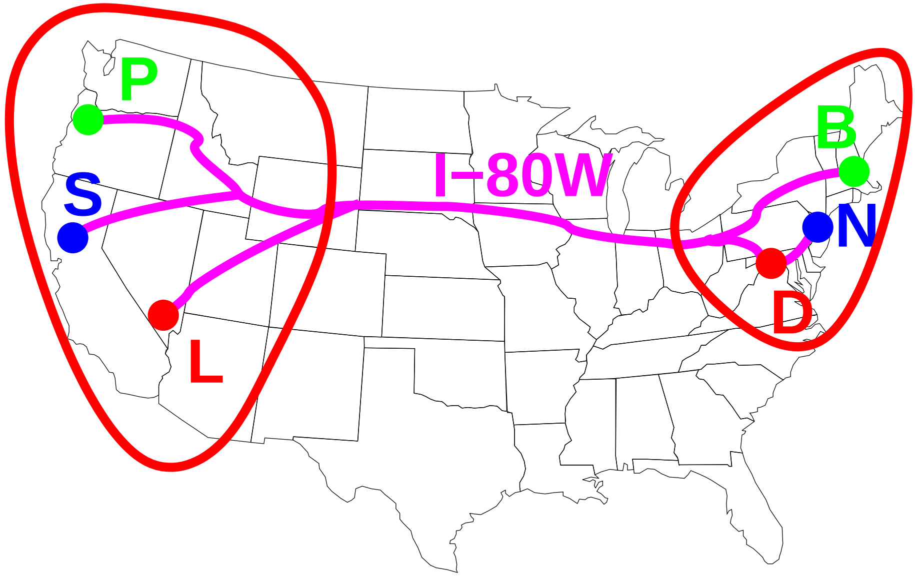
- Decompose road network into PCPs:
  - Any vertex pair is contained in exactly one PCP
  - All n^2 shortest paths are captured



Finding Path Coherent Pairs in Spatial Networks

- Source Vertices: Washington, DC (D) , New York (N) , Boston (B)
- Destination vertices: Las Vegas (L) , Sacramento (S) , Portland (P)
- Anyone driving from “North-East” to “North-West” US uses I-80W
- Capture shortest paths from one million (say) sources in “North-East” to one million (say) destinations in “North-West” using $O(1)$ storage
- Intuition: Sources “sufficiently far” from destinations share common vertices in their shortest paths

- Decompose road network into PCPs:
  - Any vertex pair is contained in exactly one PCP
  - All n^2 shortest paths are captured



- Key idea is the analogy to the well-separated pairs in computational geometry

Is SILC Still Useful?

1. Type of refinement
   - Refinement in SILC finds the next intermediate vertex
   - Refinement in oracles fetches some intermediate vertex
2. Quality of refinement
   - SILC is superior as the network distance between source and destination is always expressed as an exact network distance from source to some intermediate vertex plus the network distance interval from the intermediate vertex to the destination
   - While in the case of distance oracles, the network distance between source and destination is always expressed as the sum of two network distance intervals

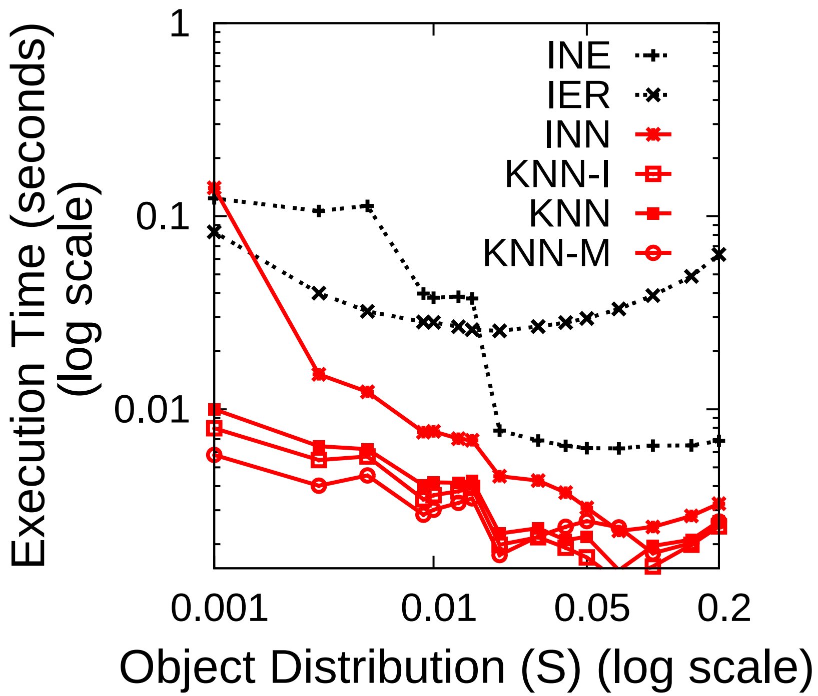
Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

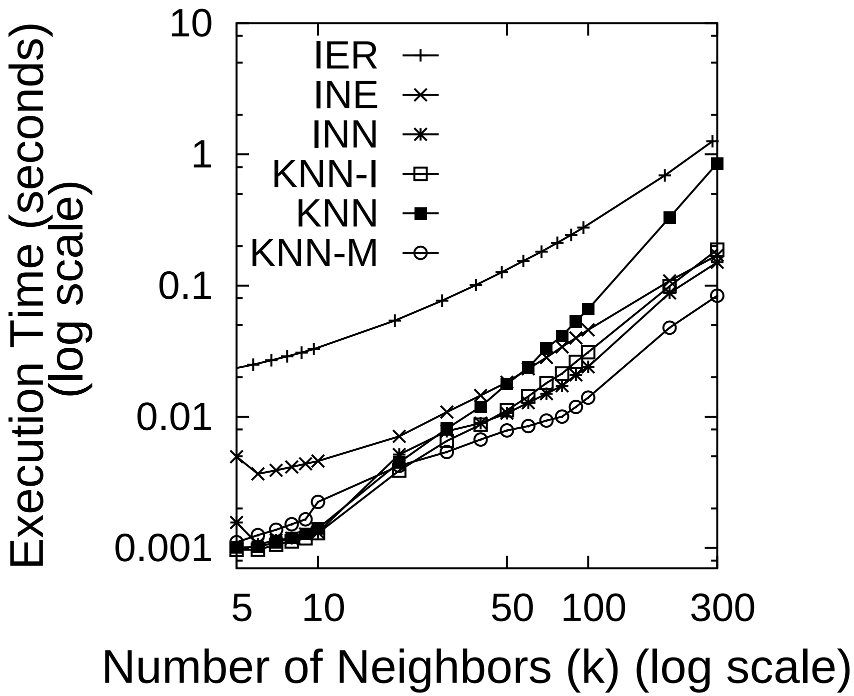
Experimental Evaluation

- Compared kNN with other algorithms including variants of kNN
  1. INE: Basically Dijkstra's algorithm [Papa03]
  2. IER: Using Euclidean distance as a filter [Papa03]
  3. INN: Incremental variant of kNN which invokes kNN k times
     - No priority queue, L, or D_k
  4. kNN-I: Use L to calculate D_k^0 using first k objects
     - Reduce size of *Queue* by not enqueueing elements with $\delta^- > D_k^0$
  5. kNN-M: Reduce number of refinements by dropping need for total ordering
     - KMINDIST keeps track of δ^- of object corresponding to D_k^0
     - *Queue*$_1$ contains all objects with $\delta^- \leq D_k^0$
     - Don't refine objects in *Queue*$_1$ with $\delta^+ <$ KMINDIST as automatically in L
- Linux (2.4.2 kernel), quad 2.4GHz Xeon server with 1GB of RAM, GNU C++
- LRU based cache that can hold 5% of the disk pages in main memory
- Test set is important roads on US eastern seaboard consisting of 91,113 vertices and 114,176 edges
- S is generated at random and stored in a PMR quadtree
- Each query run on at least 50 random input datasets of same size

Execution Time Comparison



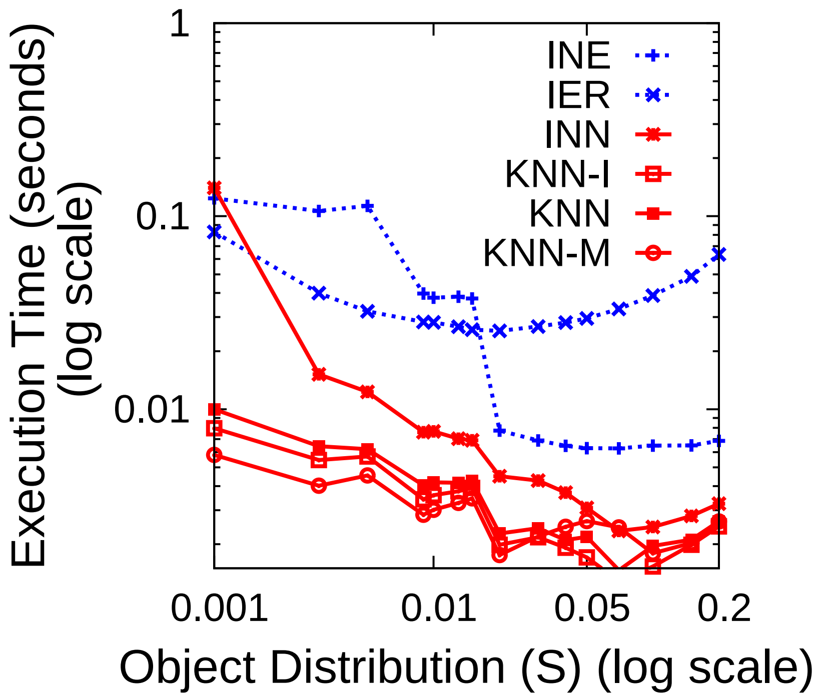
k=10 and varying sizes of S



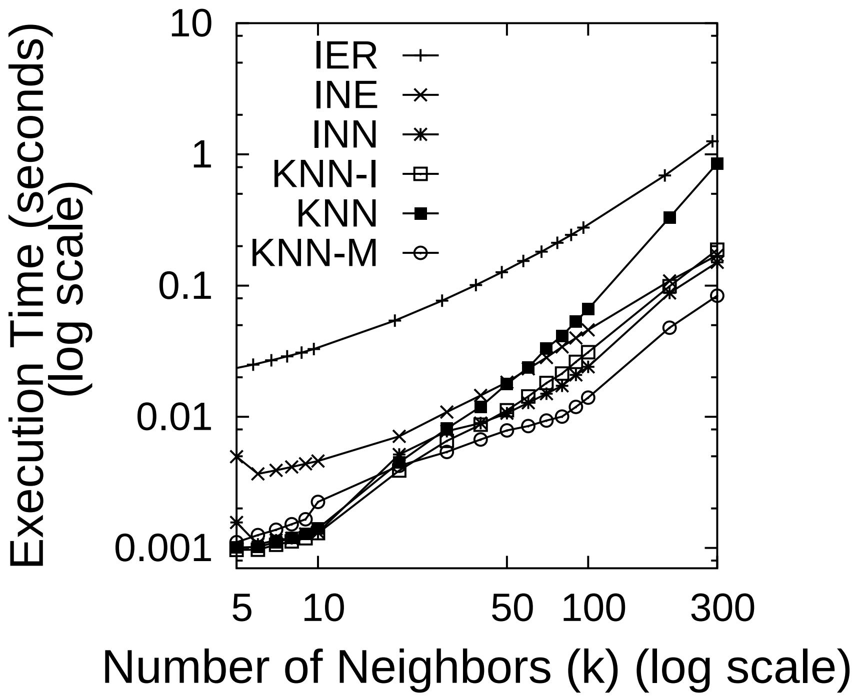
$S = 0.07N$ and varying k

- kNN and Variants

Execution Time Comparison



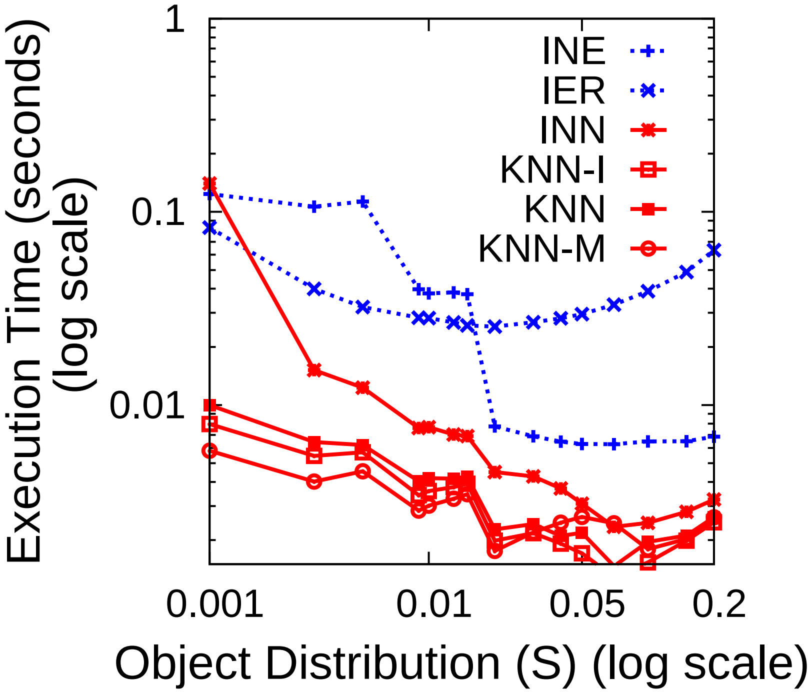
k=10 and varying sizes of S



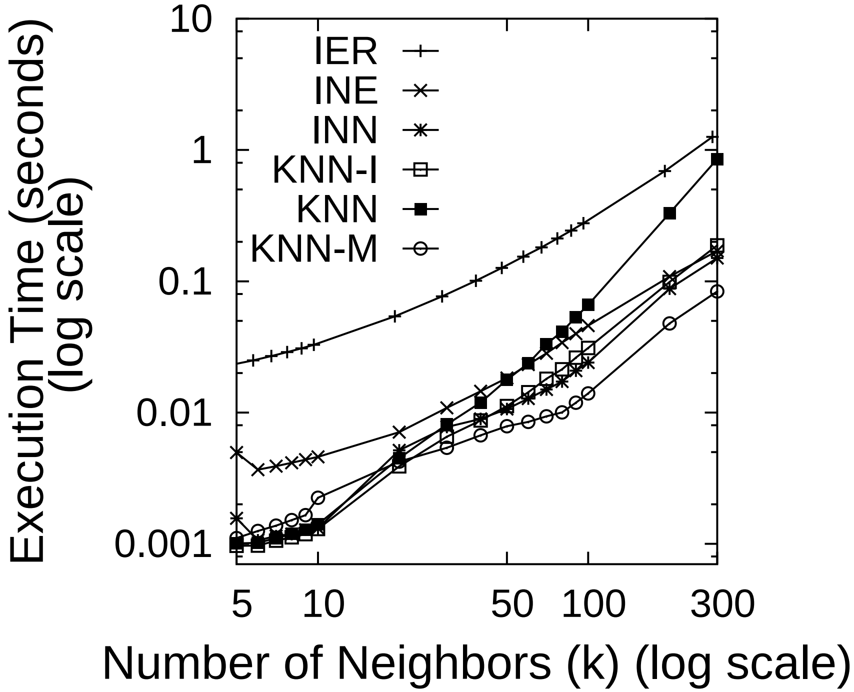
$S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S

Execution Time Comparison



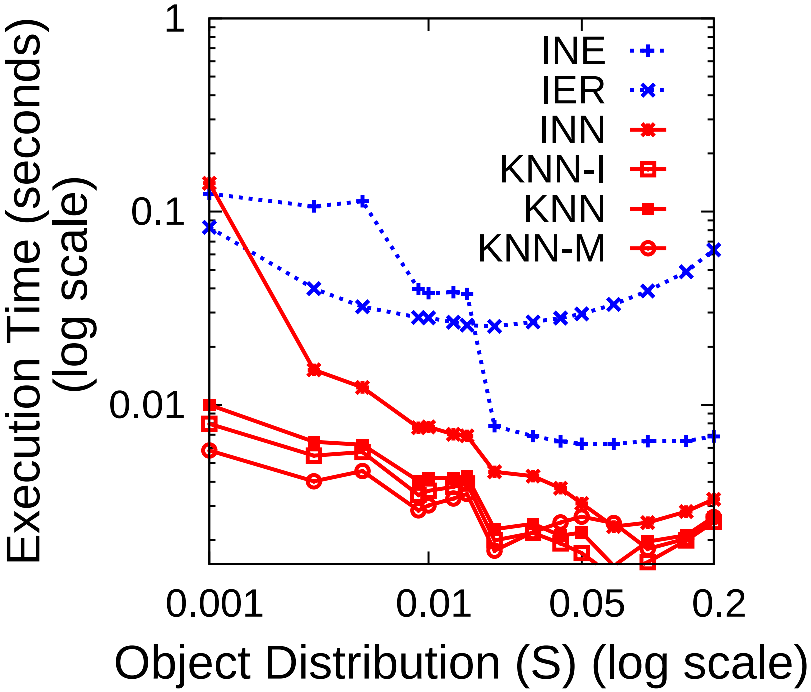
k=10 and varying sizes of S



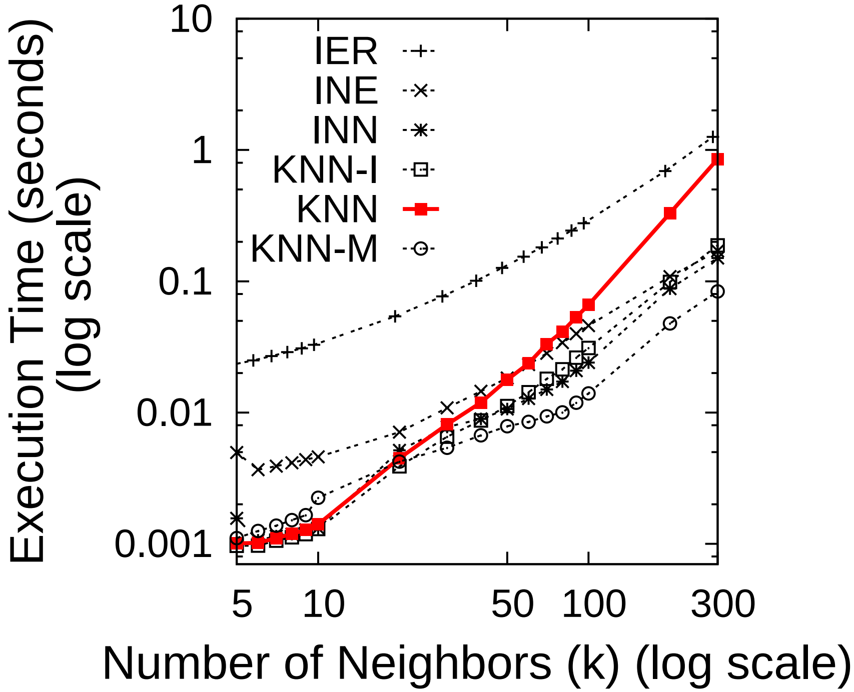
$S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S
- INE and IER improve relatively for large values of S as easy to find k neighbors around q

Execution Time Comparison



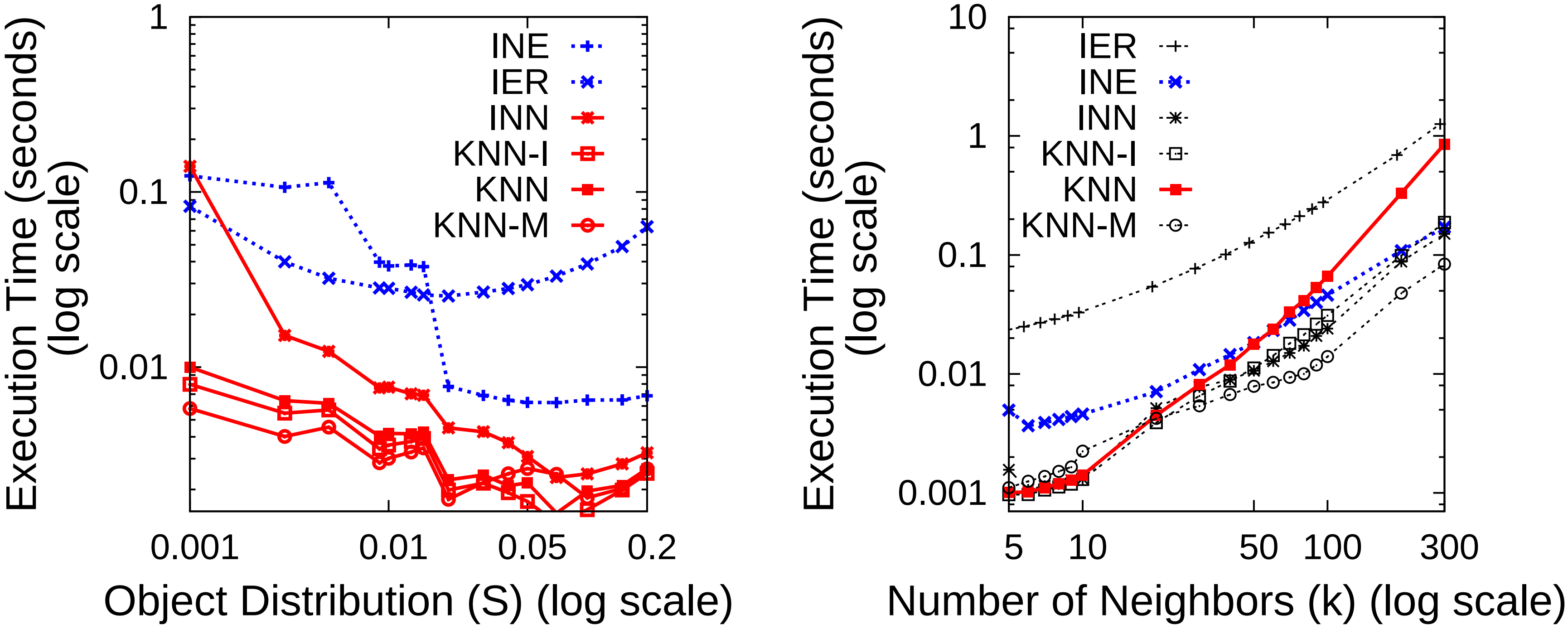
k=10 and varying sizes of S



$S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S
- INE and IER improve relatively for large values of S as easy to find k neighbors around q
- As k increases, priority queue operations take more time and kNN

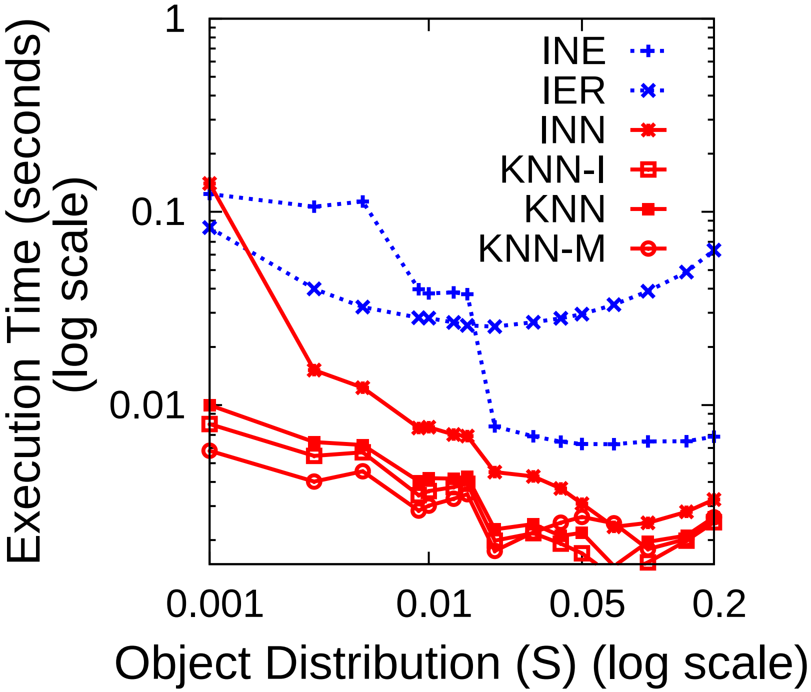
Execution Time Comparison



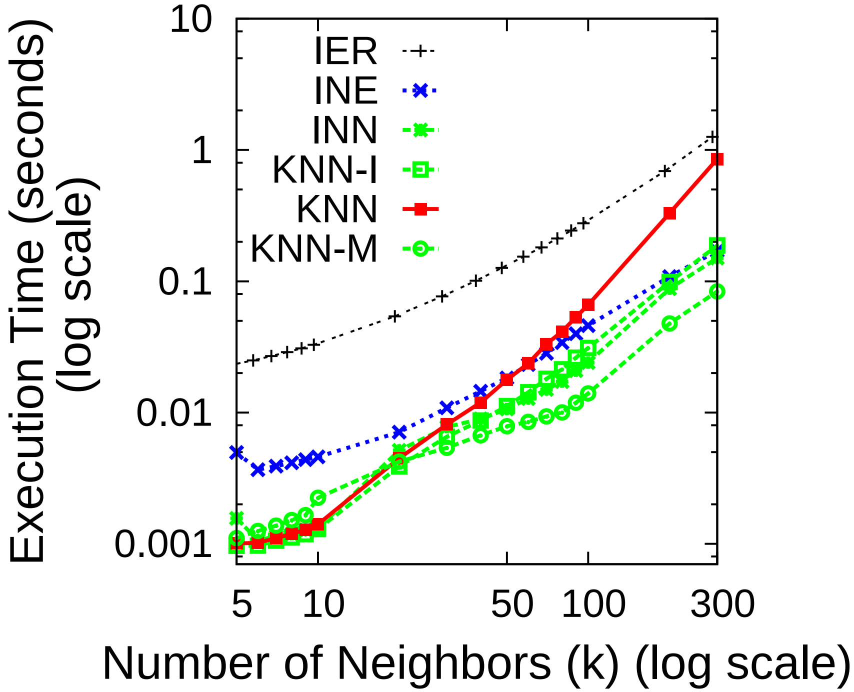
k=10 and varying sizes of S — $S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S
- INE and IER improve relatively for large values of S as easy to find k neighbors around q
- As k increases, priority queue operations take more time and kNN performs worse than INE

Execution Time Comparison



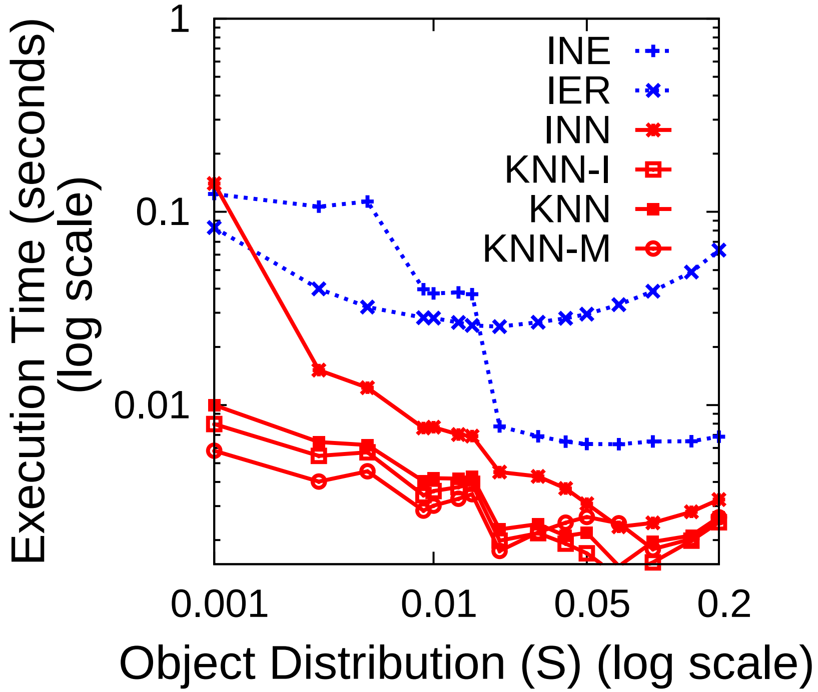
k=10 and varying sizes of S



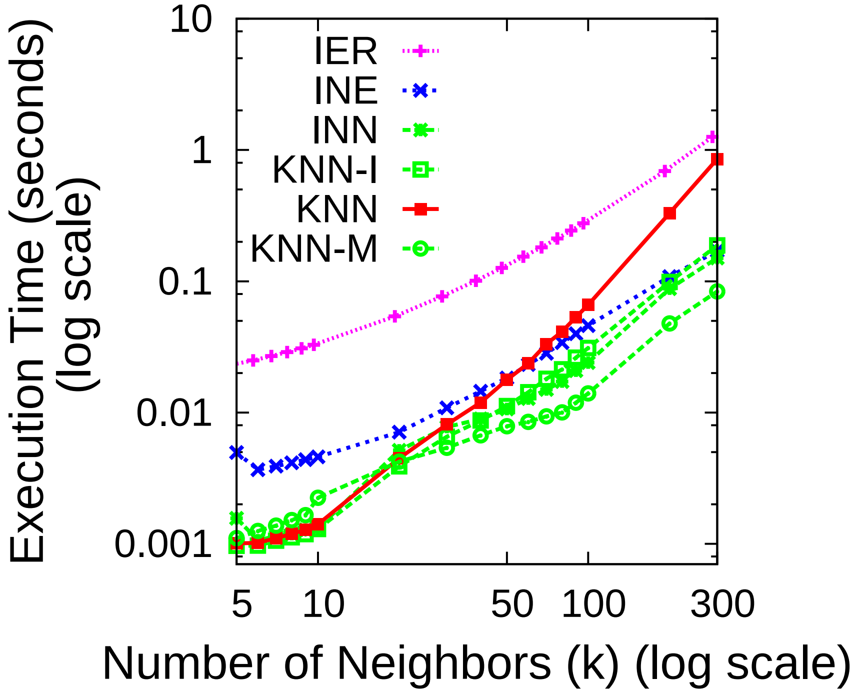
$S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S
- INE and IER improve relatively for large values of S as easy to find k neighbors around q
- As k increases, priority queue operations take more time and kNN performs worse than INE but not so for kNN variants (INN, kNN-I, kNN-M)

Execution Time Comparison



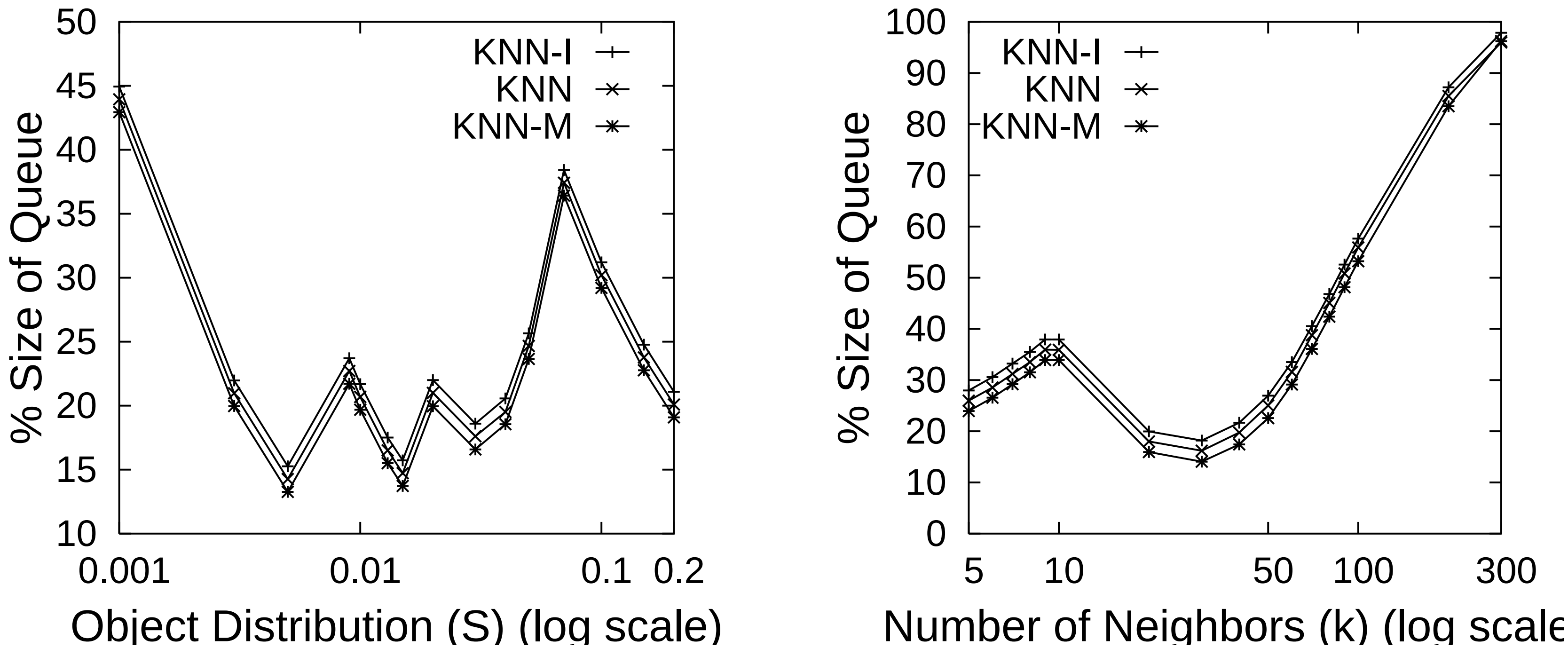
k=10 and varying sizes of S



$S = 0.07N$ and varying k

- kNN and Variants are at least one order of magnitude faster than INE and IER for small values of k and moderate values of S
- INE and IER improve relatively for large values of S as easy to find k neighbors around q
- As k increases, priority queue operations take more time and kNN performs worse than INE but not so for kNN variants (INN, kNN-I, kNN-M)
- IER always slowest

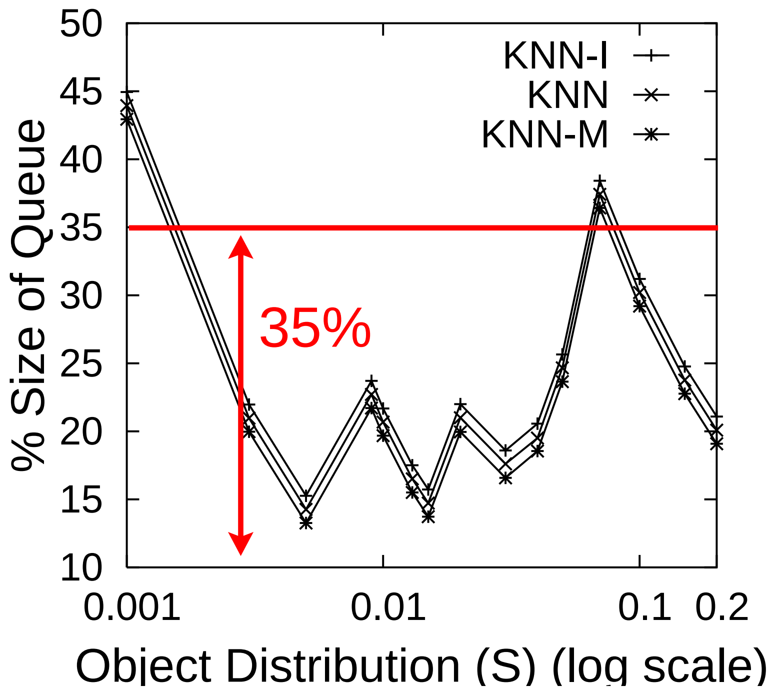
Maximum Priority Queue Size



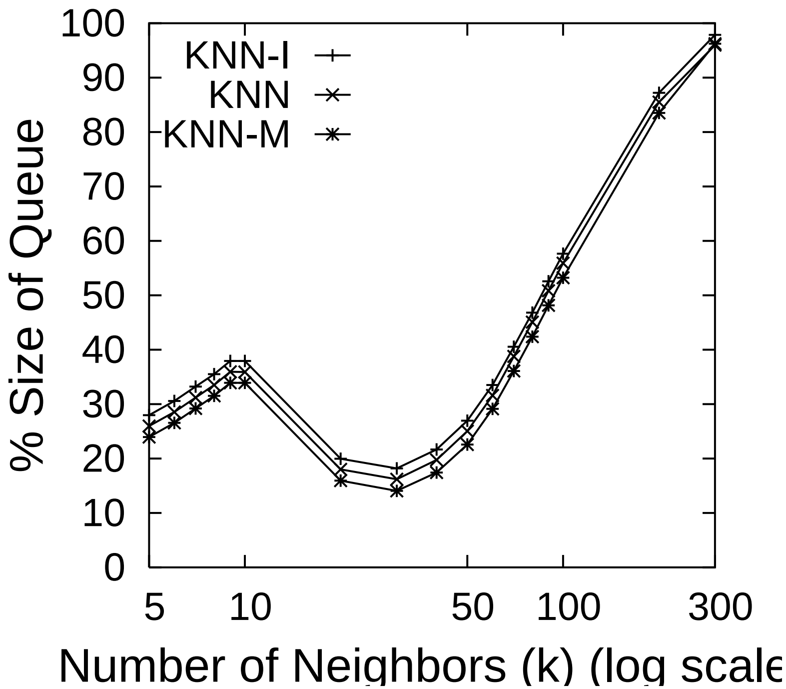
k=10 and varying sizes of S $S = 0.07N$ and varying k

- Compared maximum size of priority queue of kNN and variants with INN which cannot use D_k to reduce insertions

Maximum Priority Queue Size



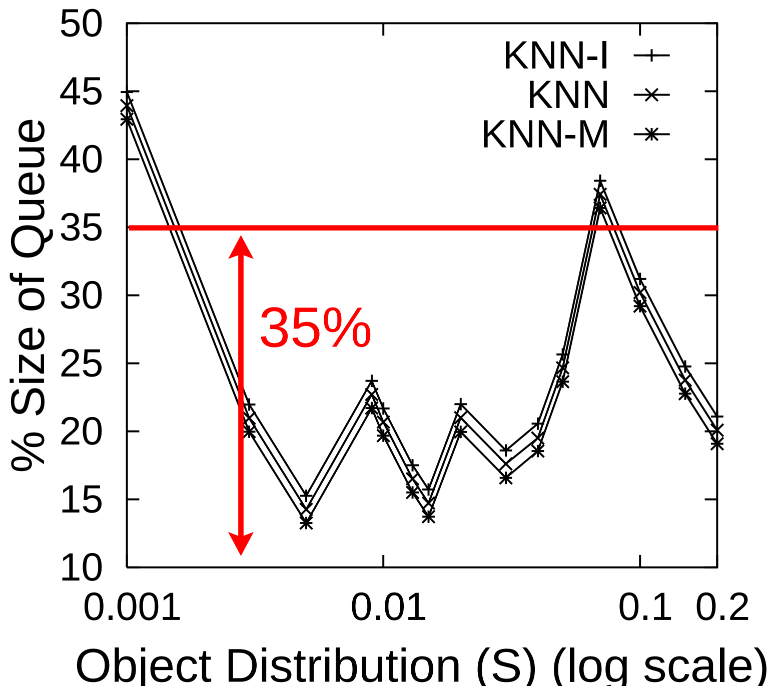
k=10 and varying sizes of S



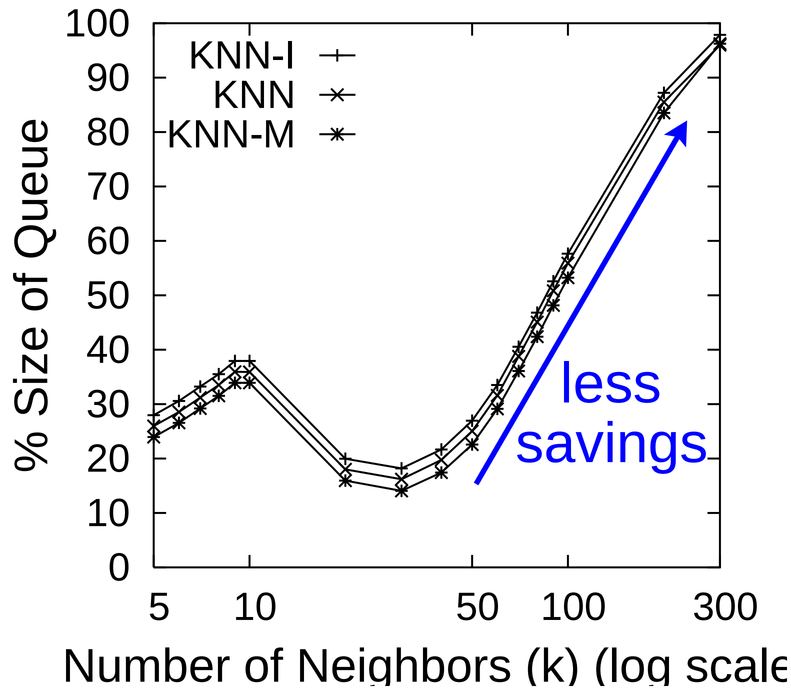
$S = 0.07N$ and varying k

- Compared maximum size of priority queue of kNN and variants with INN which cannot use D_k to reduce insertions
- 35% of INN on the average

Maximum Priority Queue Size



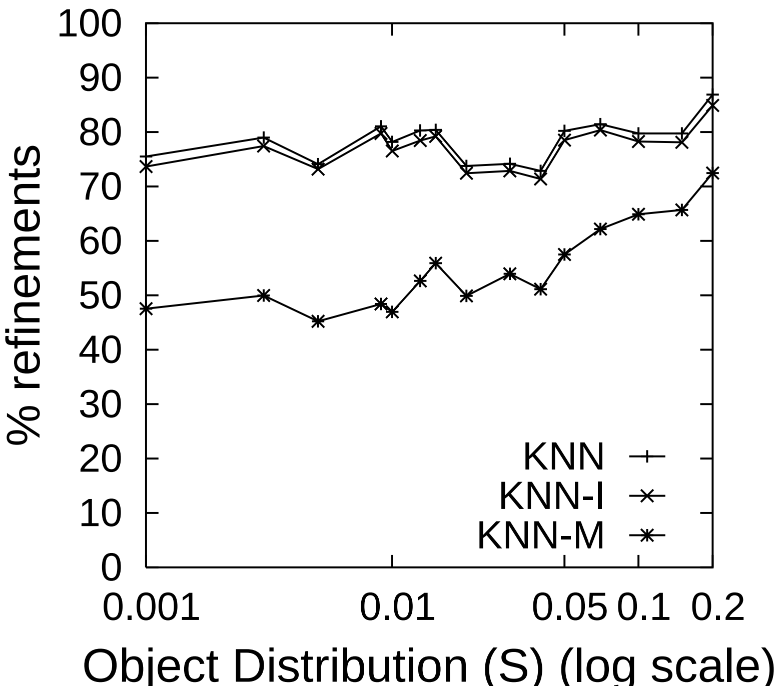
k=10 and varying sizes of S



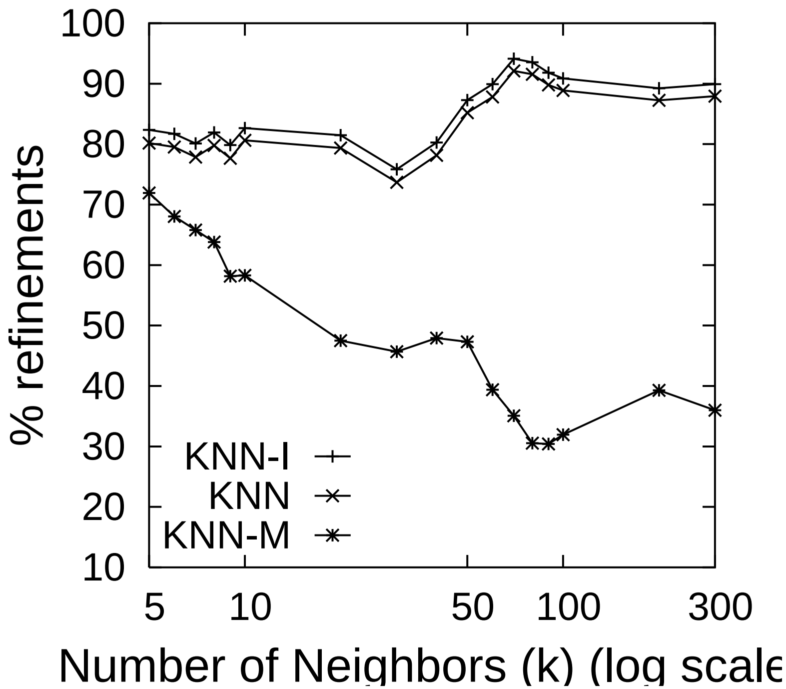
$S = 0.07N$ and varying k

- Compared maximum size of priority queue of kNN and variants with INN which cannot use D_k to reduce insertions
- 35% of INN on the average
- As k increases, savings in maximum queue size vanish
  - most likely due to an increase in the number of objects having overlapping distance intervals from q
  - Results in reducing pruning effectiveness of D_k

Number of Refinement Operations



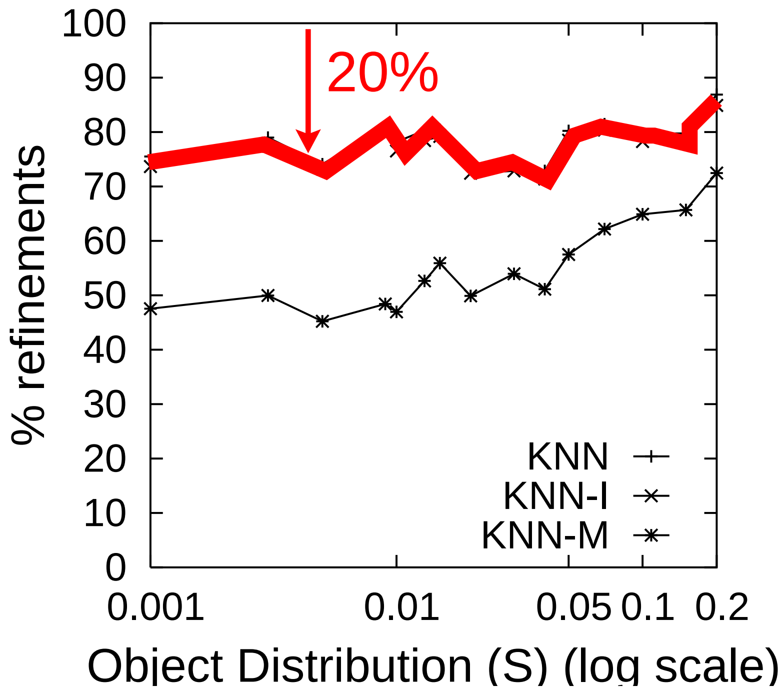
k=10 and varying sizes of S



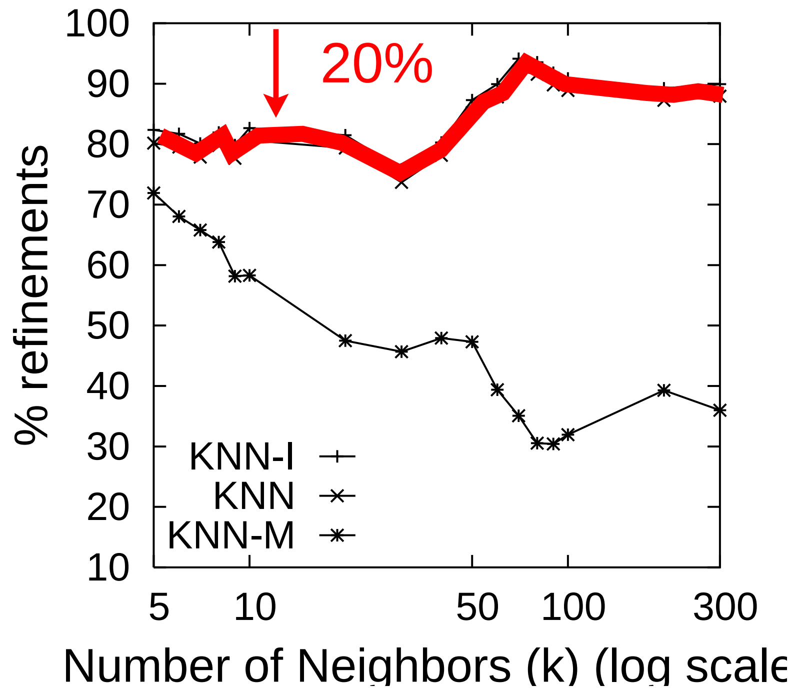
$S = 0.07N$ and varying k

- Compared reduction in refinements for

Number of Refinement Operations



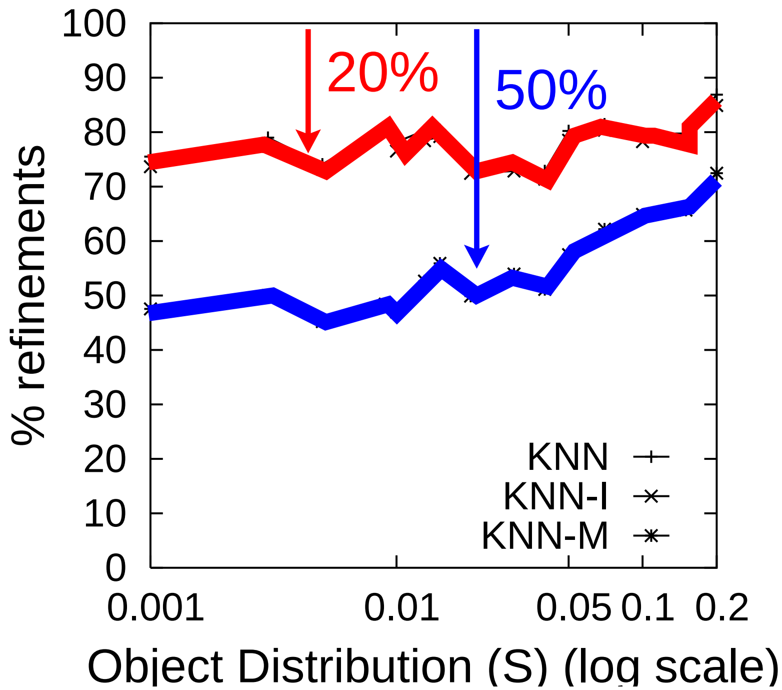
k=10 and varying sizes of S



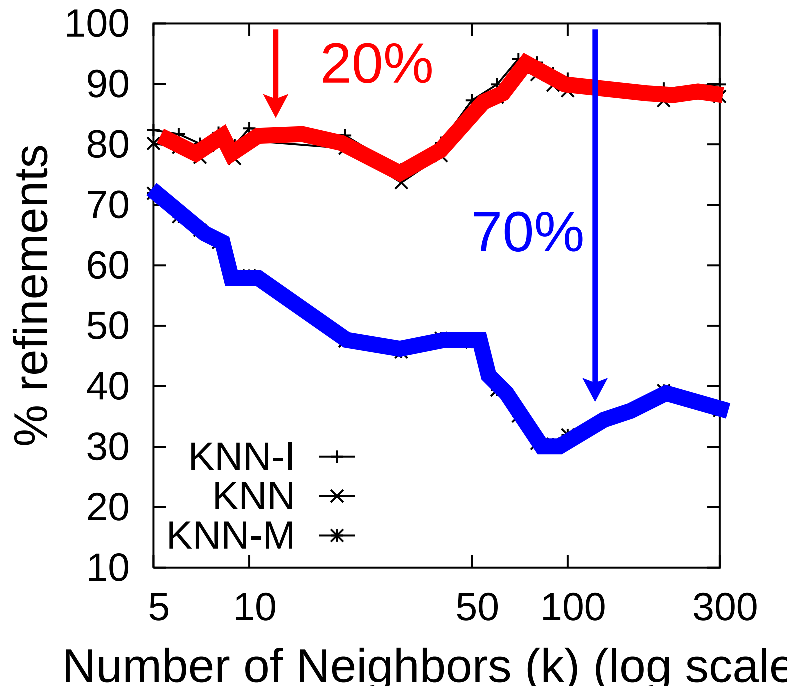
$S = 0.07N$ and varying k

- Compared reduction in refinements for kNN, kNN-I,

Number of Refinement Operations



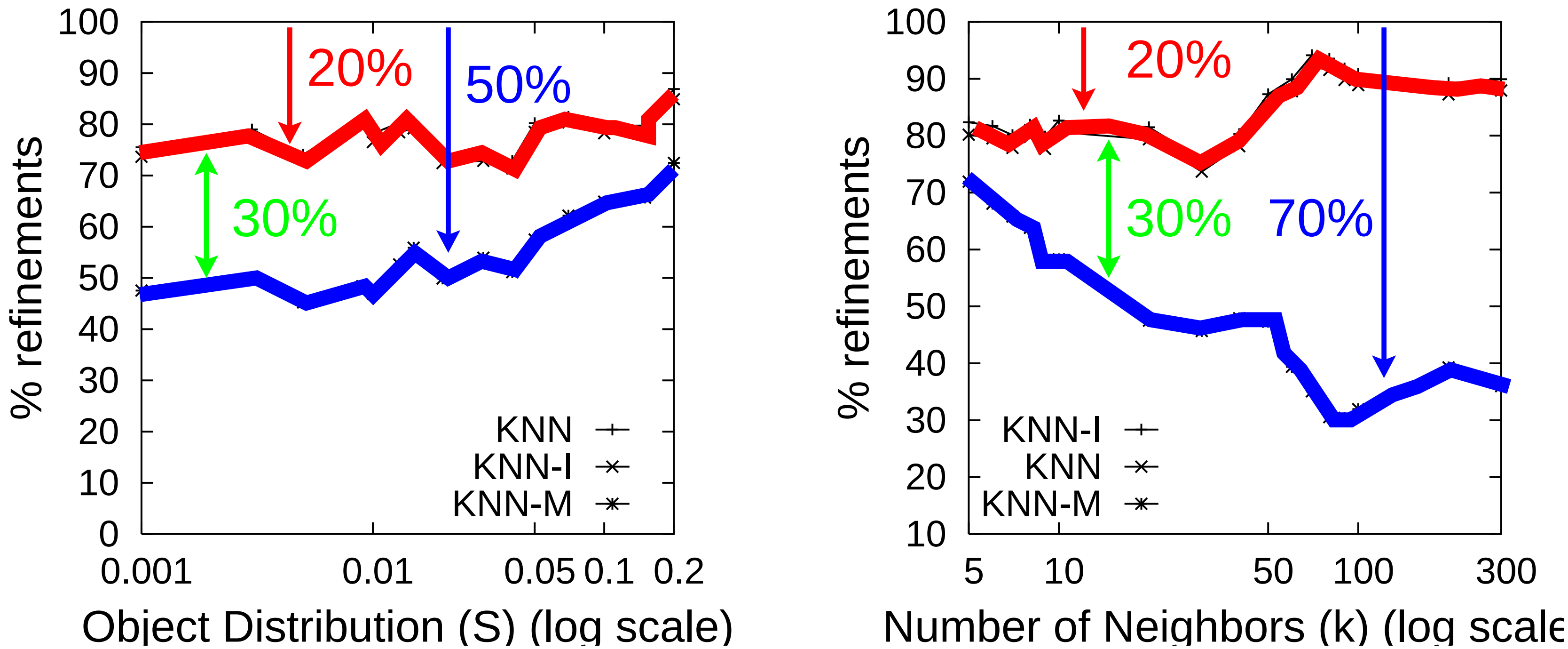
k=10 and varying sizes of S



$S = 0.07N$ and varying k

- Compared reduction in refinements for kNN, kNN-I, kNN-M with INN

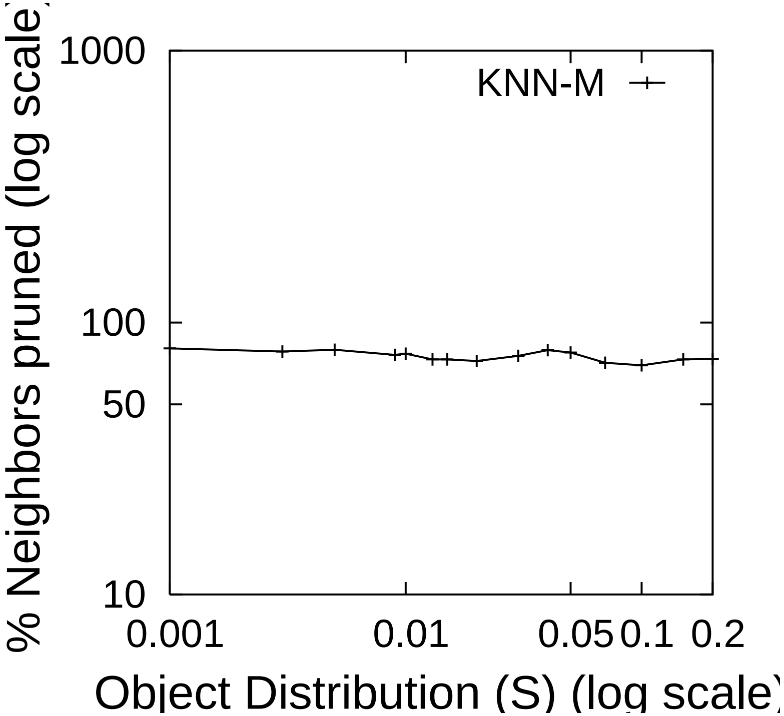
Number of Refinement Operations



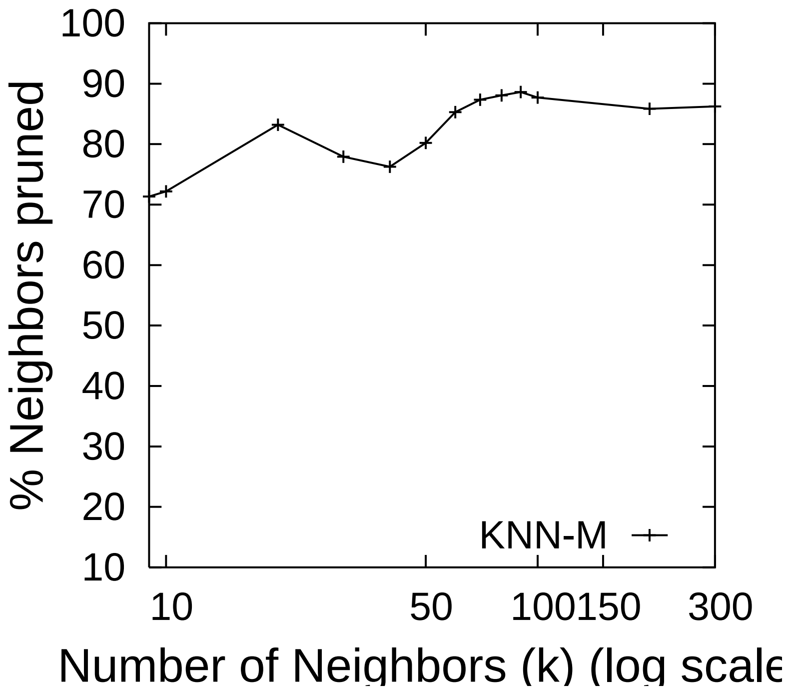
k=10 and varying sizes of S $\qquad$ $S = 0.07N$ and varying k

- Compared reduction in refinements for kNN, kNN-I, kNN-M with INN
- Large reduction for kNN-M which means that at least 30% of refinements in kNN are devoted to developing a total ordering

Roles of kMinDist in Pruning

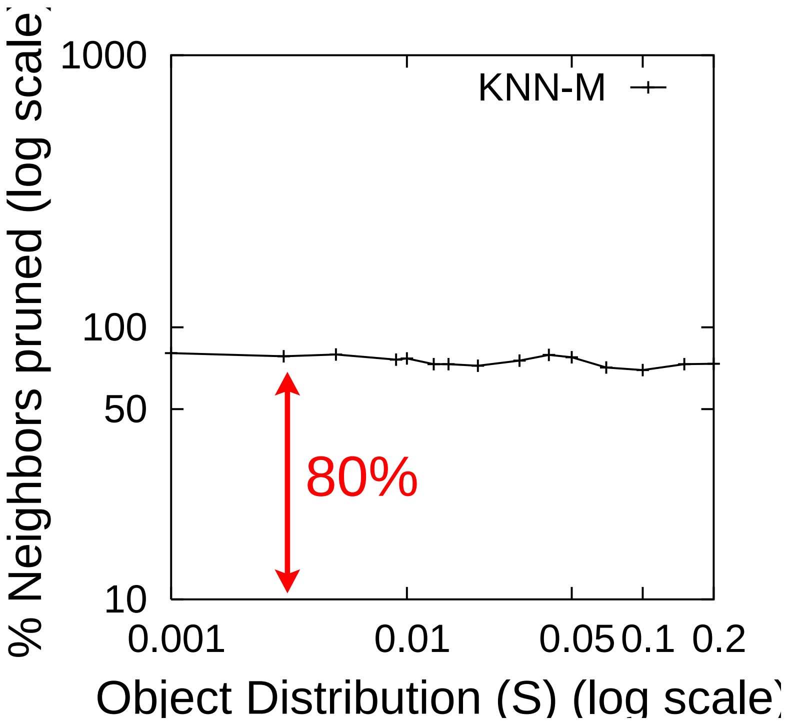


k=10 and varying sizes of S

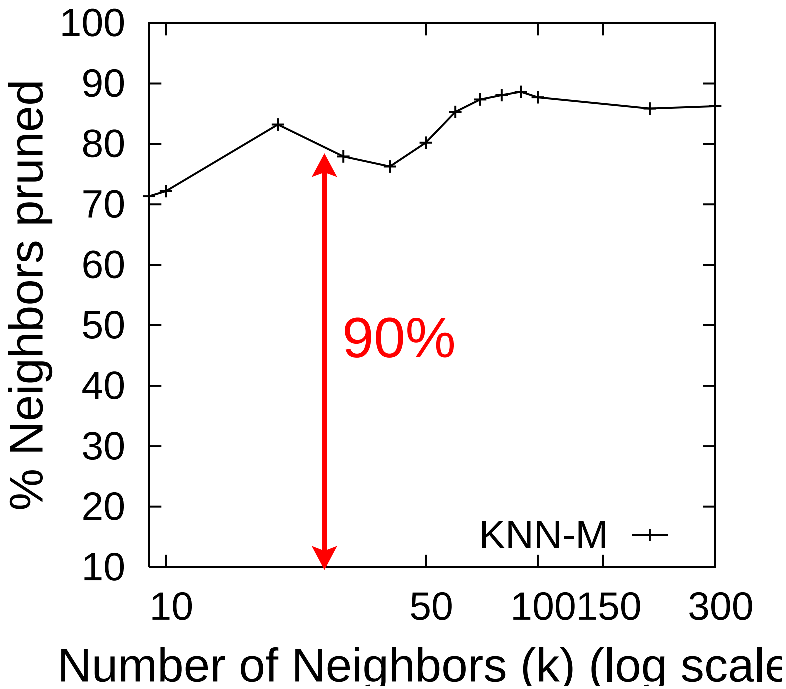


$S = 0.07N$ and varying k

Roles of kMinDist in Pruning



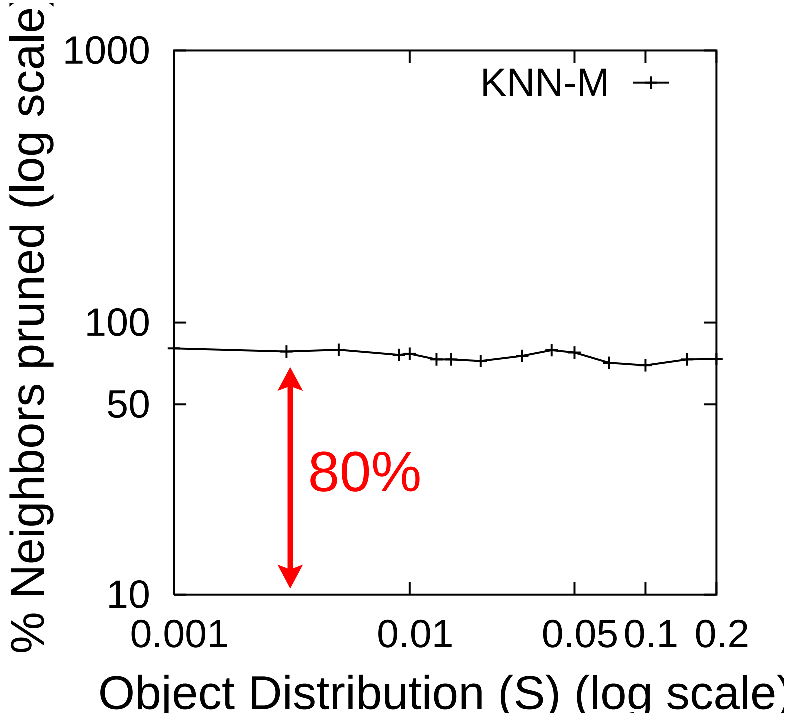
k=10 and varying sizes of S



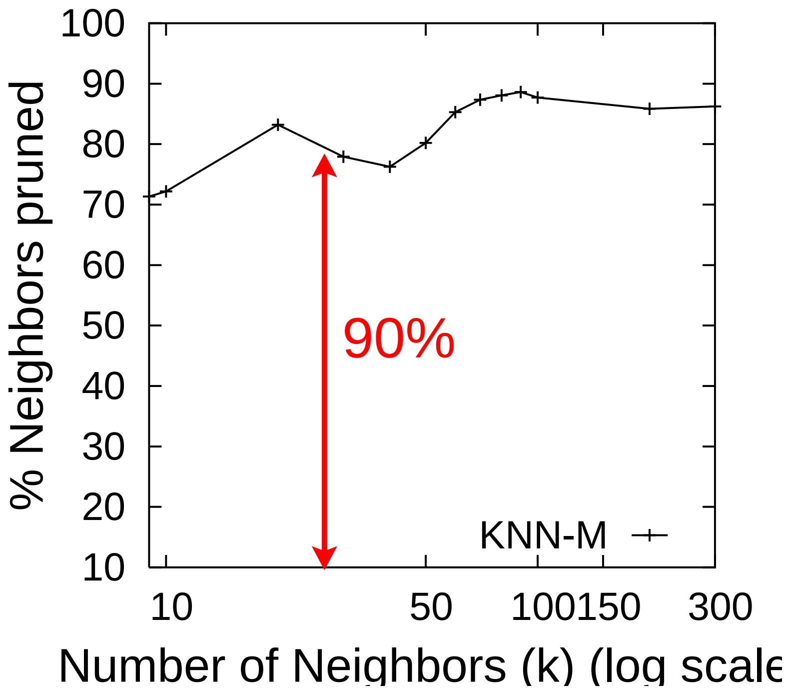
$S = 0.07N$ and varying k

- Up to 90% of nearest neighbors in kNN-M were pruned against kMinDist
  - kMinDist yields minimum possible distance of $k\text{th}$ nearest neighbor
  - Means any object with $\delta^+ \leq$ kMinDist can be added directly to result
  - But output is no longer sorted

Roles of kMinDist in Pruning



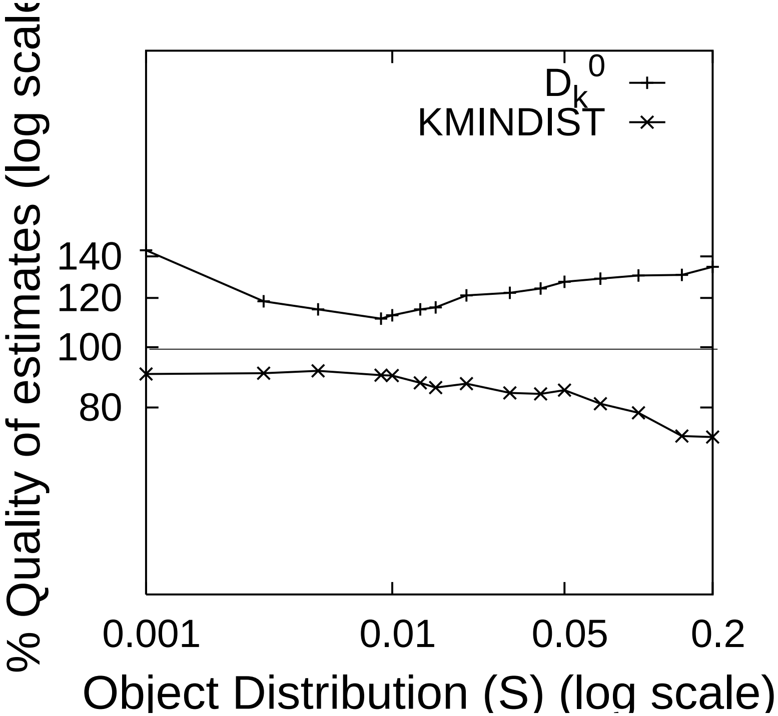
k=10 and varying sizes of S



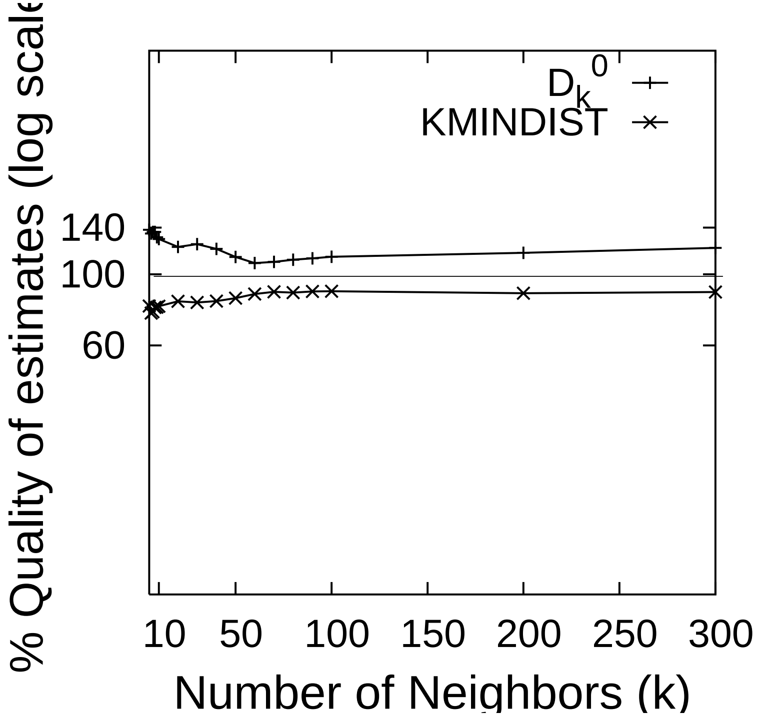
$S = 0.07N$ and varying k

- Up to 90% of nearest neighbors in kNN-M were pruned against kMinDist
  - kMinDist yields minimum possible distance of $k\mathrm{th}$ nearest neighbor
  - Means any object with $\delta^+ \leq$ kMinDist can be added directly to result
  - But output is no longer sorted
- However, does not eliminate an equivalent number of refinements as most refinements are usually performed before pruning could take place

Role of KMINDIST and D_k^0 Vis-a-Vis D_k

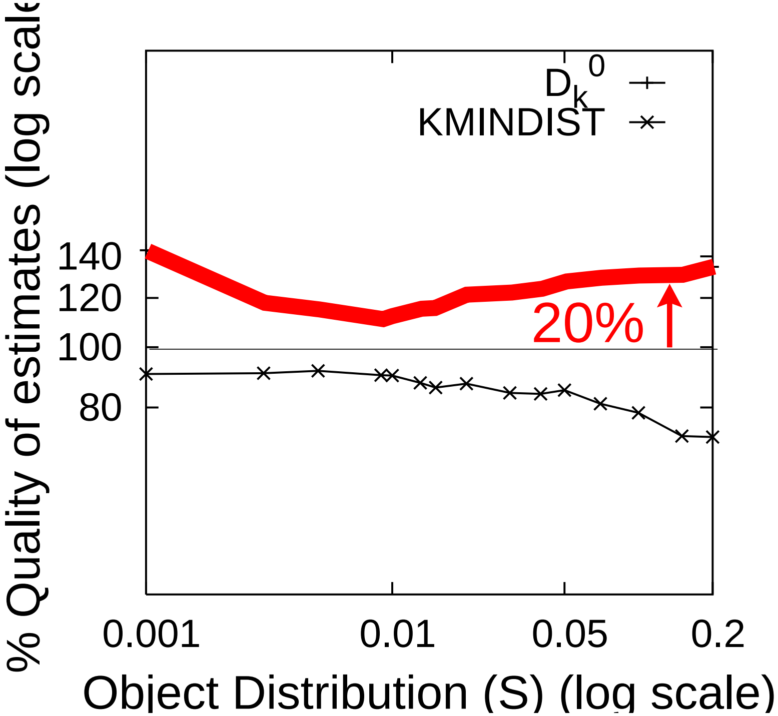


k=10 and varying sizes of S

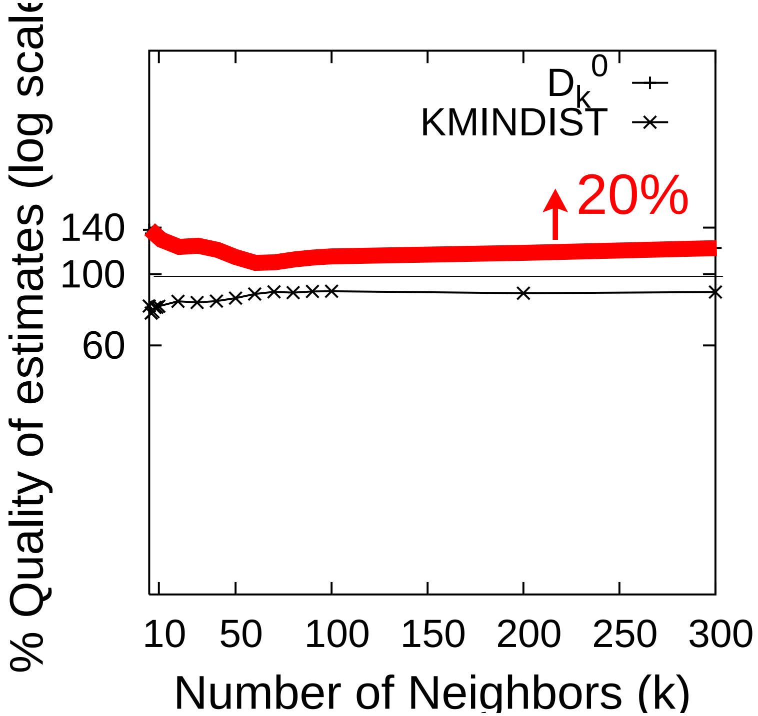


$S = 0.07N$ and varying k

Role of KMINDIST and D_k^0 Vis-a-Vis D_k



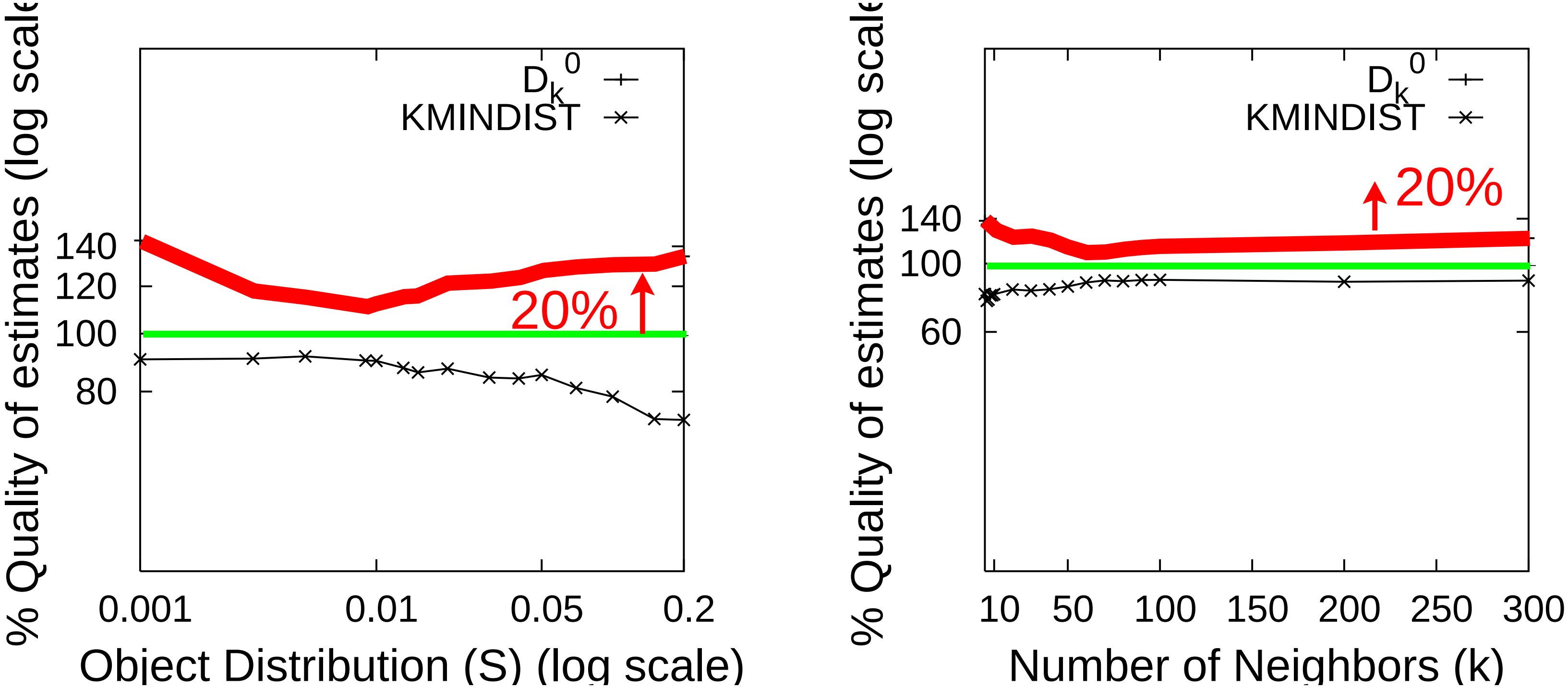
k=10 and varying sizes of S



$S = 0.07N$ and varying k

- D_k^0 is about 20% larger than

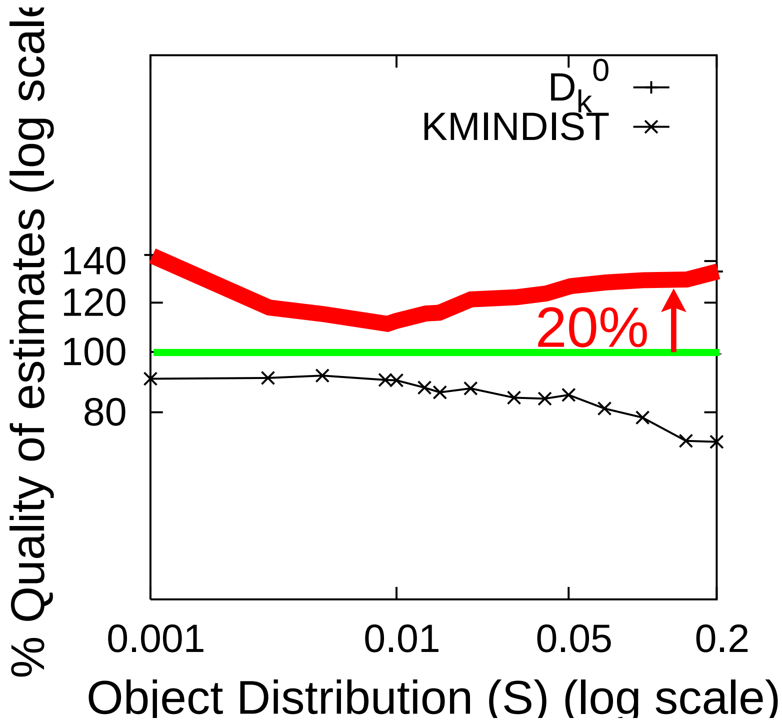
Role of KMINDIST and D_k^0 Vis-a-Vis D_k



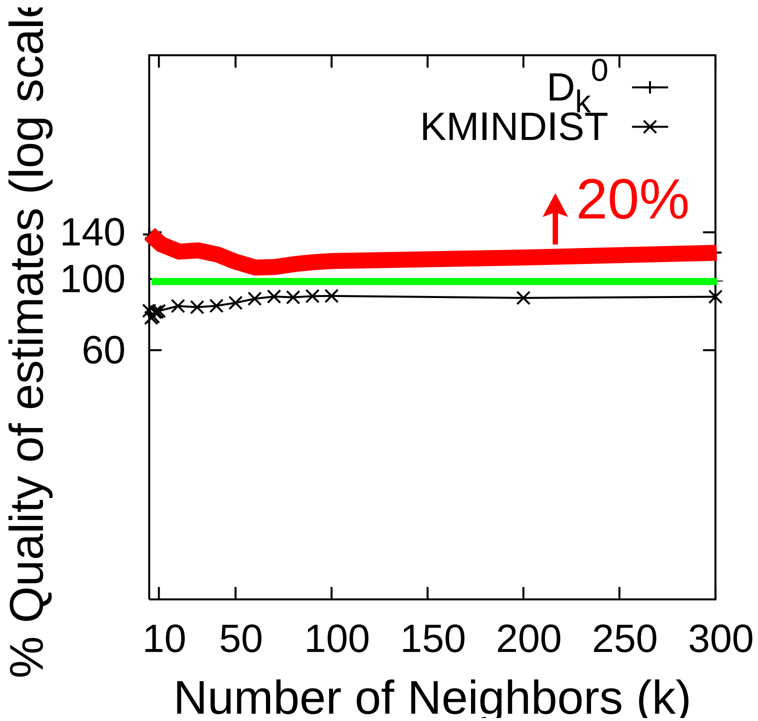
k=10 and varying sizes of S $\quad$ $S = 0.07N$ and varying k

- D_k^0 is about 20% larger than D_k

Role of KMINDIST and D_k^0 Vis-a-Vis D_k



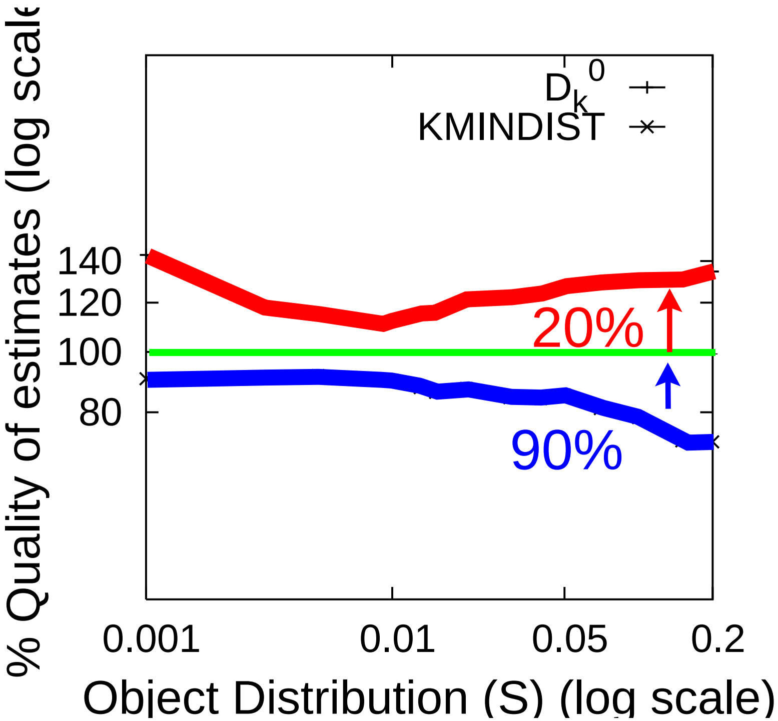
k=10 and varying sizes of S



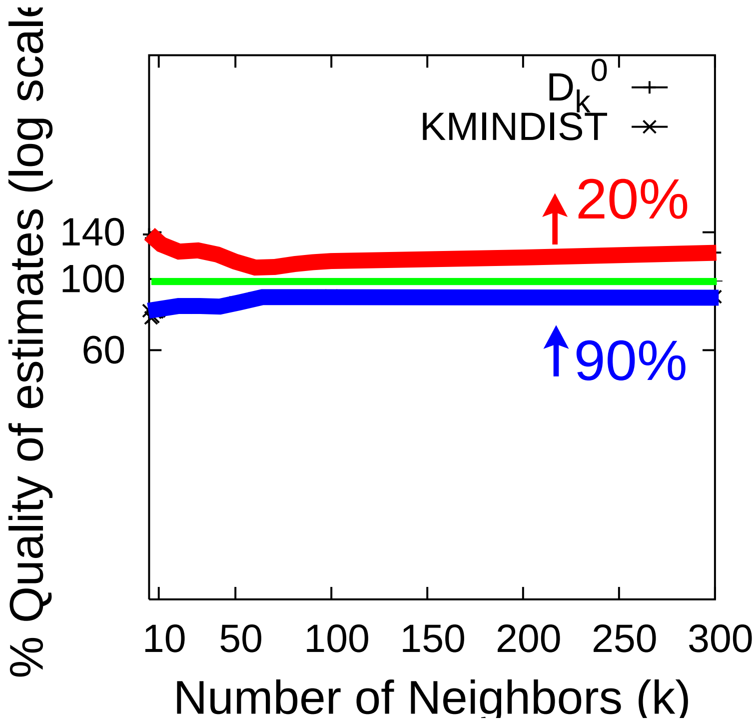
$S = 0.07N$ and varying k

- D_k^0 is about 20% larger than D_k
  - 20% means that D_k does not lead to much more pruning than D_k^0 explaining why the maximum sizes of the priority queues for kNN, kNN-I, and kNN-M are almost identical when compared to that for INN

Role of KMINDIST and D_k^0 Vis-a-Vis D_k



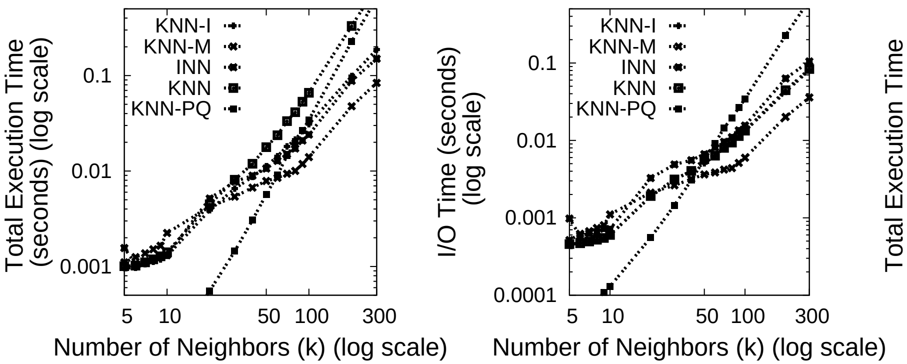
k=10 and varying sizes of S



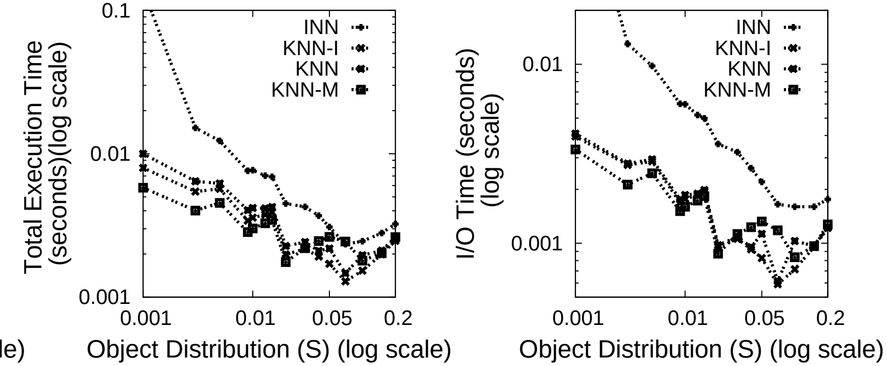
$S = 0.07N$ and varying k

- D_k^0 is about 20% larger than D_k
  - 20% means that D_k does not lead to much more pruning than D_k^0 explaining why the maximum sizes of the priority queues for kNN, kNN-I, and kNN-M are almost identical when compared to that for INN
- KMINDIST is about 90% of D_k which may explain why many objects in kNN-M are added directly to the result set without need for further refinements (said to *pruned* against KMINDIST)

Comparison of kNN and variants

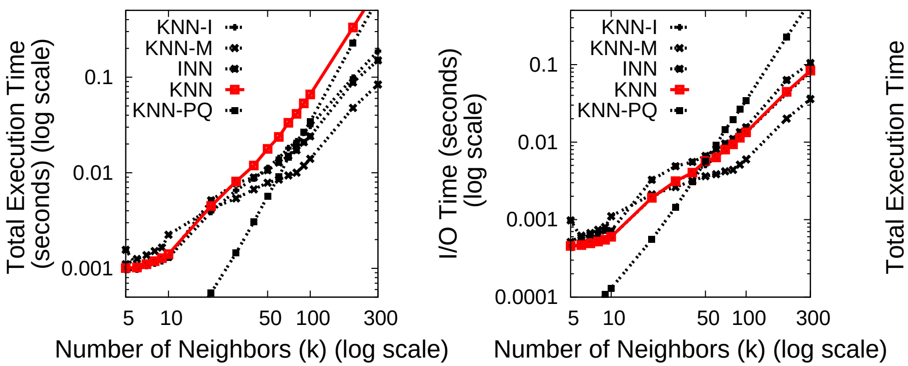


$S = 0.07N$ and varying k

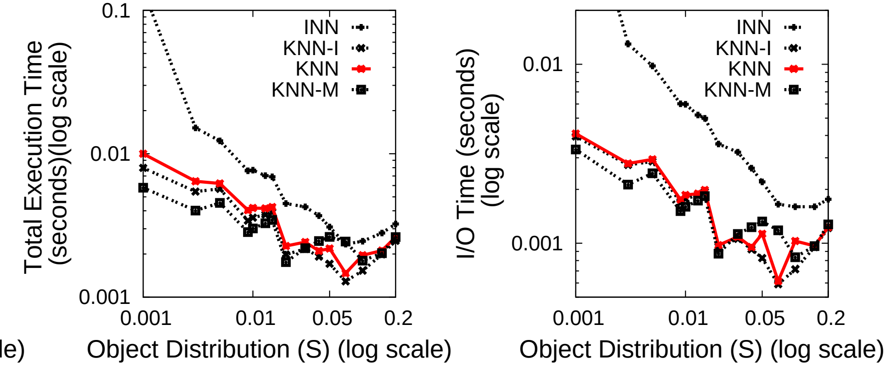


k=10 and varying sizes of S

Comparison of kNN and variants



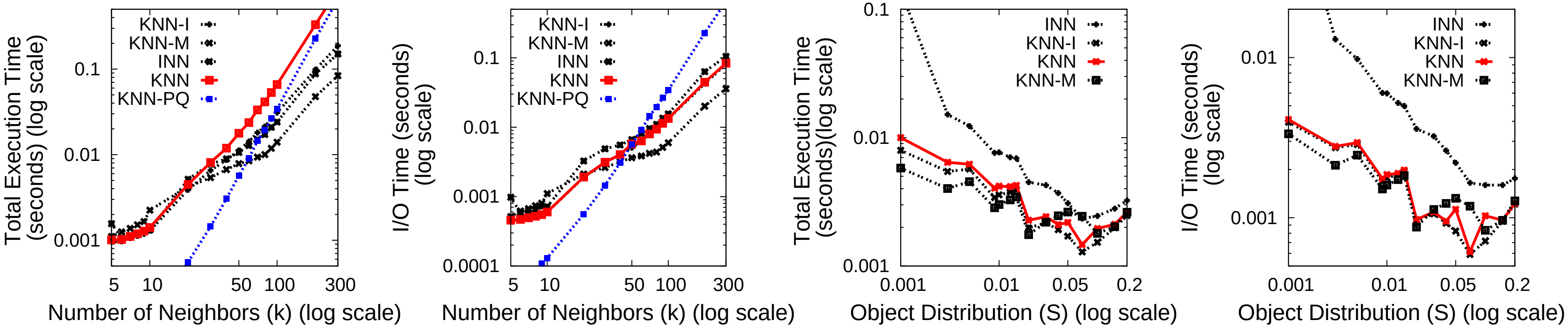
$S = 0.07N$ and varying k



k=10 and varying sizes of S

- Small value of k, kNN is best

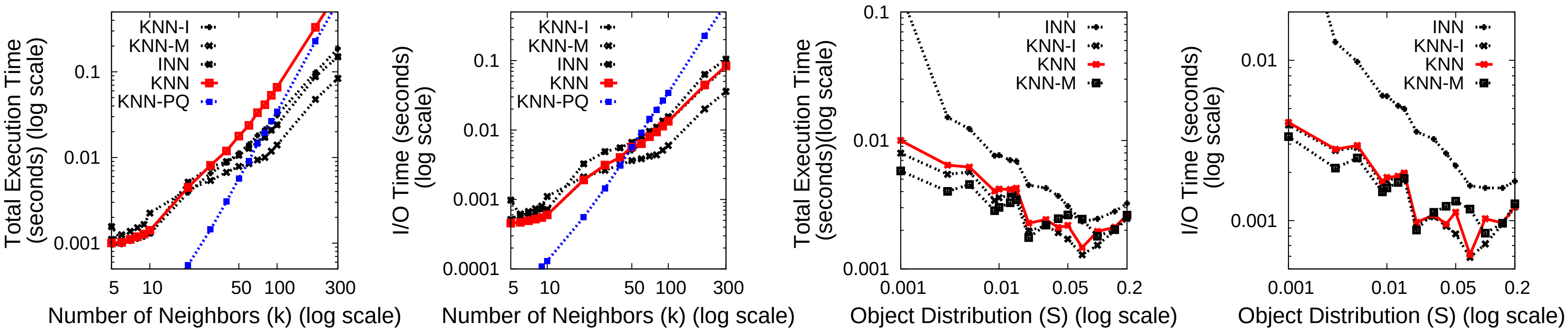
Comparison of kNN and variants



$S = 0.07N$ and varying k k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial

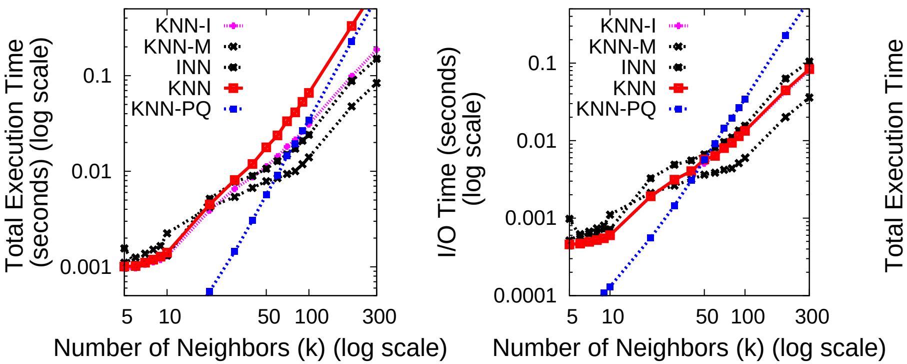
Comparison of kNN and variants



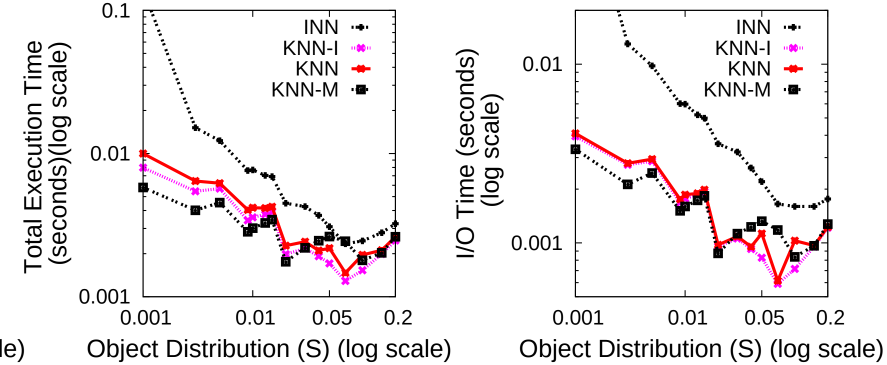
$S = 0.07N$ and varying k $\quad$ k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use

Comparison of kNN and variants



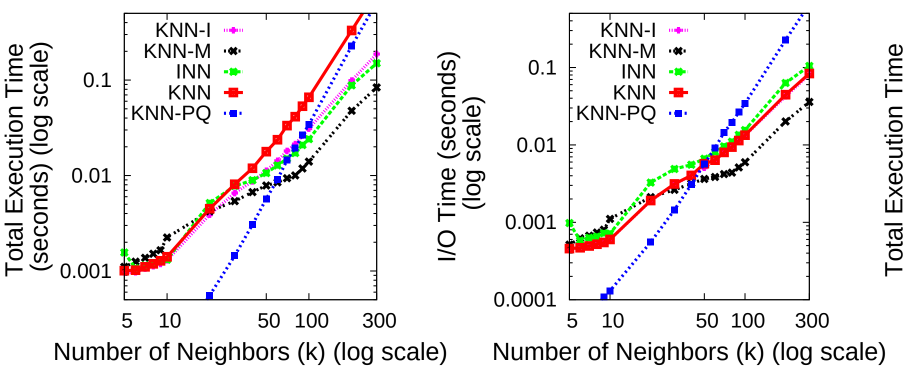
$S = 0.07N$ and varying k



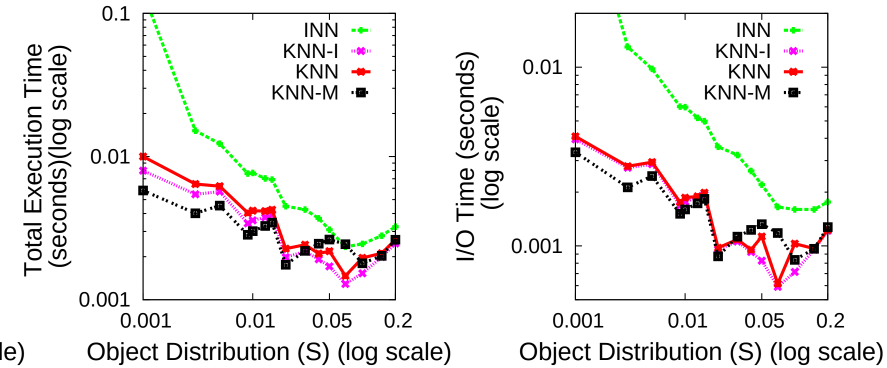
k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and

Comparison of kNN and variants



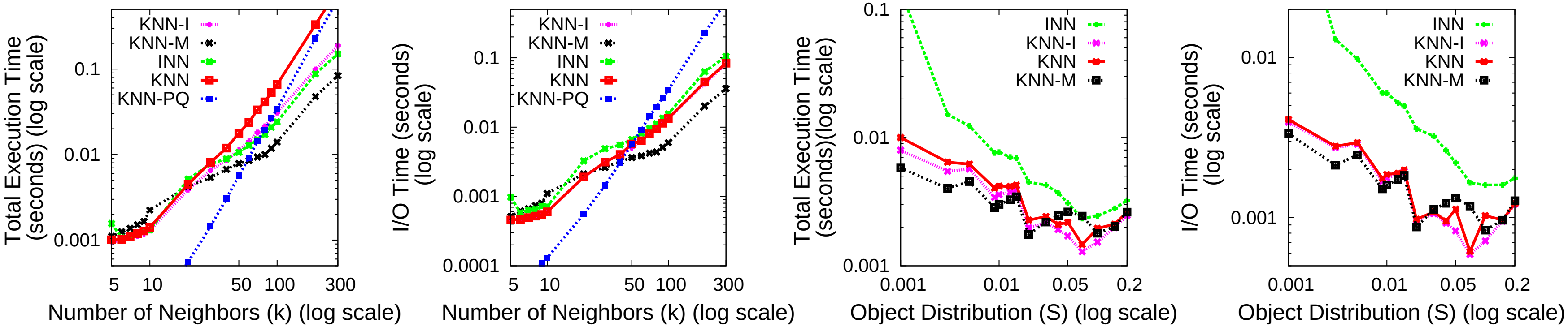
$S = 0.07N$ and varying k



k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and INN as less or no manipulation of L

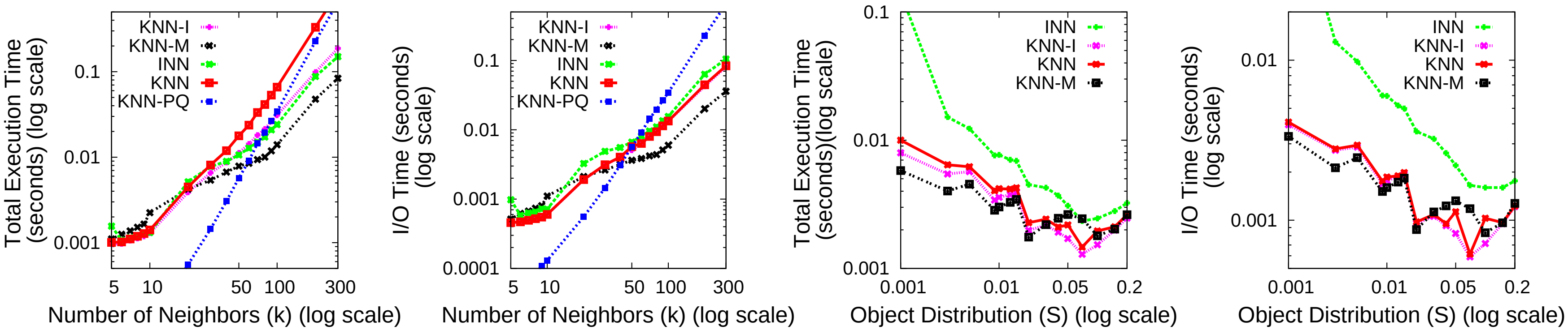
Comparison of kNN and variants



$S = 0.07N$ and varying k k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and INN as less or no manipulation of L
- I/O time dominates total execution time of kNN and variants

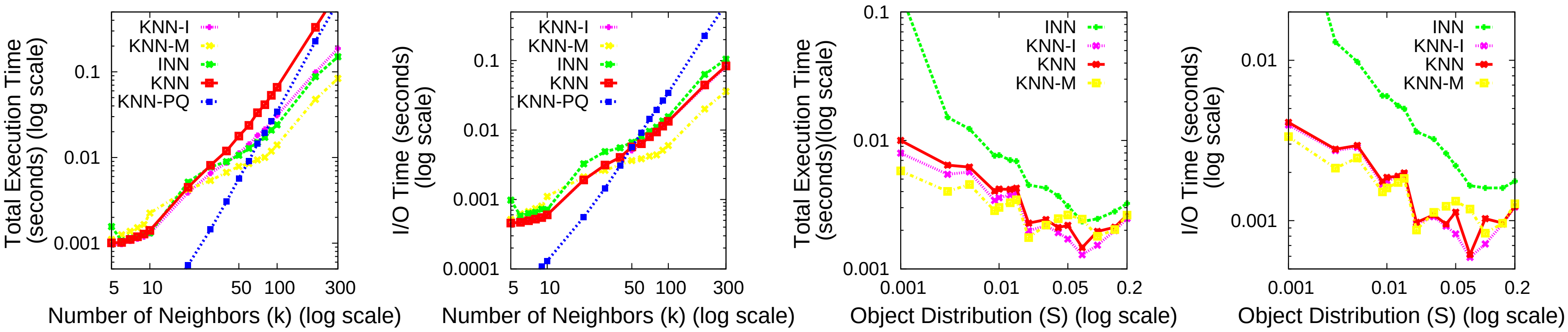
Comparison of kNN and variants



$S = 0.07N$ and varying k k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and INN as less or no manipulation of L
- I/O time dominates total execution time of kNN and variants
  - Each refinement may lead to a disk access

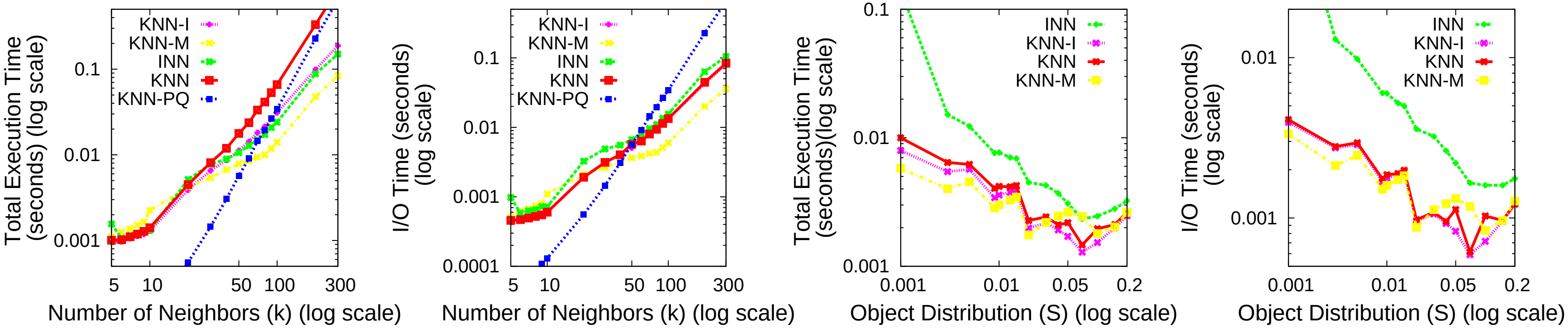
Comparison of kNN and variants



$S = 0.07N$ and varying k $\quad$ k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and INN as less or no manipulation of L
- I/O time dominates total execution time of kNN and variants
  - Each refinement may lead to a disk access
- Use kNN-M if no need to sort results

Comparison of kNN and variants



$S = 0.07N$ and varying k $\qquad$ k=10 and varying sizes of S

- Small value of k, kNN is best
- kNN-PQ: cost of priority queue L and D_k manipulations and is substantial
  - As k increases ($k > 20$), use kNN-I and INN as less or no manipulation of L
- I/O time dominates total execution time of kNN and variants
  - Each refinement may lead to a disk access
- Use kNN-M if no need to sort results
- Execution time for all variants of kNN decreases as size of S increases since objects are closer to query object and fewer refinement steps

Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

Contributions

1. Efficient k nearest neighbor computation in a spatial network

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
   - Permit reusing shortest path computations across different queries and different datasets

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
    - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
    - Permit reusing shortest path computations across different queries and different datasets
    - General framework for query processing in spatial networks
        - Not restricted to nearest neighbor queries
        - Can be easily integrated with a database

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
   - Permit reusing shortest path computations across different queries and different datasets
   - General framework for query processing in spatial networks
     - Not restricted to nearest neighbor queries
     - Can be easily integrated with a database
4. Avoid applying Dijkstra's algorithm for each query which visits all vertices on the shortest path to the destination

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
   - Permit reusing shortest path computations across different queries and different datasets
   - General framework for query processing in spatial networks
     - Not restricted to nearest neighbor queries
     - Can be easily integrated with a database
4. Avoid applying Dijkstra's algorithm for each query which visits all vertices on the shortest path to the destination
   - Transform solution from a graph-based combinatorial algorithm to a purely geometric one

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
   - Permit reusing shortest path computations across different queries and different datasets
   - General framework for query processing in spatial networks
     - Not restricted to nearest neighbor queries
     - Can be easily integrated with a database
4. Avoid applying Dijkstra's algorithm for each query which visits all vertices on the shortest path to the destination
   - Transform solution from a graph-based combinatorial algorithm to a purely geometric one
5. Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$

Contributions

1. Efficient k nearest neighbor computation in a spatial network
2. Precompute shortest paths between all vertices in spatial network
   - Shown to be a viable solution strategy
3. Decouple shortest path and nearest neighbor computation processes
   - Permit reusing shortest path computations across different queries and different datasets
   - General framework for query processing in spatial networks
     - Not restricted to nearest neighbor queries
     - Can be easily integrated with a database
4. Avoid applying Dijkstra's algorithm for each query which visits all vertices on the shortest path to the destination
   - Transform solution from a graph-based combinatorial algorithm to a purely geometric one
5. Reduce cost of storing shortest paths between all pairs of N vertices from $O(N^3)$ to $O(N^{1.5})$
   - Scalable

Outline

1. Overview
2. Spatial Networks
3. Precomputation and storage of shortest paths
4. k Nearest Neighbor Finding Algorithm
5. Experimental evaluation
6. Contributions
7. Future Work

Future Work

- Apply to other queries on a spatial network

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques
- Handle dynamic updates spatial networks

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques
- Handle dynamic updates spatial networks
  - Road networks with time varying traffic information

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques
- Handle dynamic updates spatial networks
  - Road networks with time varying traffic information
  - Route planning with link failures

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques
- Handle dynamic updates spatial networks
  - Road networks with time varying traffic information
  - Route planning with link failures
- Applicability of progressive refinement of distances to other domains where distance computations are expensive

Future Work

- Apply to other queries on a spatial network
- Approximate query processing on spatial networks
- Applicability of SILC framework to massive road networks
  - Use of *parallel* (e.g., GPUs) and *cloud computing* (e.g., Map-Reduce) techniques
- Handle dynamic updates spatial networks
  - Road networks with time varying traffic information
  - Route planning with link failures
- Applicability of progressive refinement of distances to other domains where distance computations are expensive
  - E.g., mesh and terrain models

References

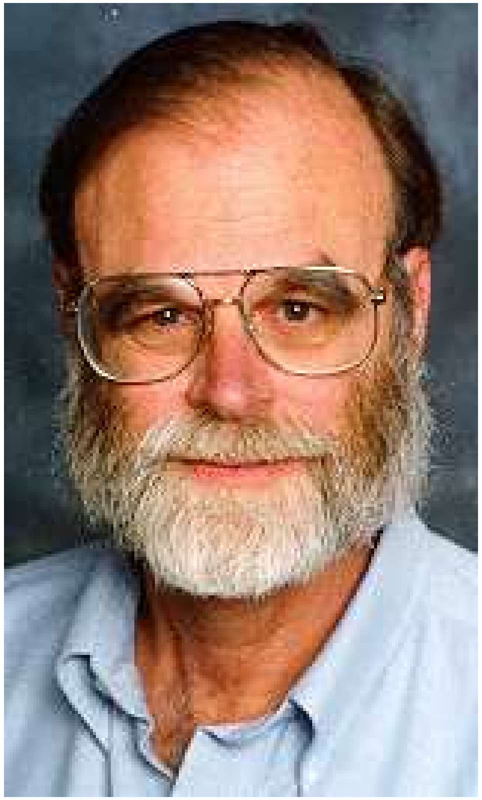
1. [Call95] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, San Francisco, 1995.
2. [Filh02] G. G. Filho and H. Samet. A linear iterative approach for hierarchical shortest path finding. Computer Science Department CS-TR-4417, University of Maryland, College Park, MD, Nov. 2002.
3. [Gold05a] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* search meets graph theory. In *SODA '05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, Vancouver, BC, Canada, Jan. 2005.
4. [Hjal95] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD '95: Proceedings of the 4th International Symposium on Large Spatial Databases*, pages 83–95, Portland, ME, Aug. 1995.
5. [Hjal98] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD '98: Proceedings of the International Conference on Management of Data*, pages 237–248, Seattle, WA, June 1998.

References

6. [Jing98] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: an optimal model and its performance evaluation. ons of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, CA, 2005.
7. [Papa03] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB'03: Proceedings of the 29th International Conference on Very Large Databases*, pages 802–813, Berlin, Germany, Sept. 2003.
8. [Same05] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2005.
9. [Shah03] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, Sept. 2003.
10. [Shin00] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD '00: Proceedings of the International Conference on Management of Data*, pages 343–354, Dallas, TX, May 2000.
11. [Wagn03] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA '03: Proceedings of the 11th Annual European Symposium on Algorithms*, pages 776–787, Budapest, Hungary, Sept. 2003.

Acknowledgments

1. National Science Foundation
2. Microsoft Research under Jim Gray
3. Microsoft Research Virtual Earth Project
4. NVIDIA Corporation
5. University of Maryland General Research Board



Questions and Comments?

Thank you!