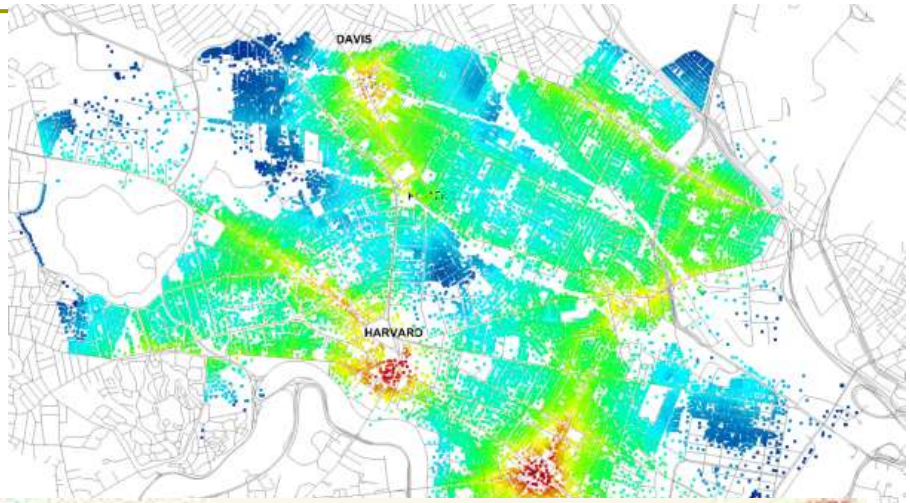
# COMP7106B
# Big Data Management
## Lecture 3 Spatial Networks

Prof. Reynold Cheng

24th February, 2024

*Acknowledgement: This presentation is based on Prof. Nikos Mamoulis' COMP3323 slides.*

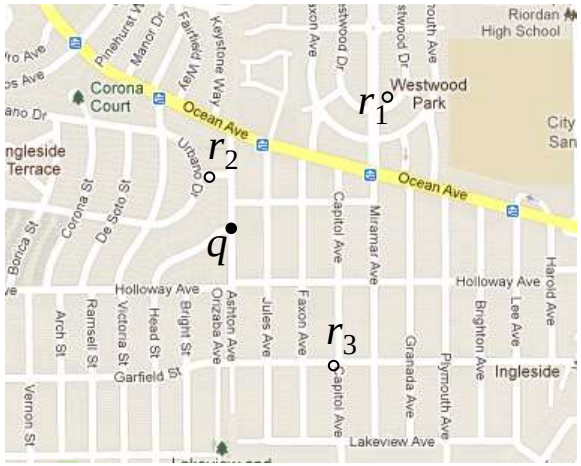
# Spatial Networks



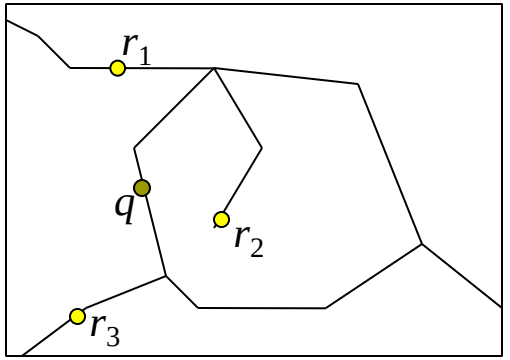
# Spatial Networks

- Modeling and storing spatial networks
- Shortest path search
- Spatial queries over spatial networks
- Advanced indexing techniques for spatial networks



# Network Distance

- In many real applications accessibility of objects is restricted by a spatial network
  - Examples
    - Driver looking for nearest gas station
    - Mobile user looking for nearest restaurant



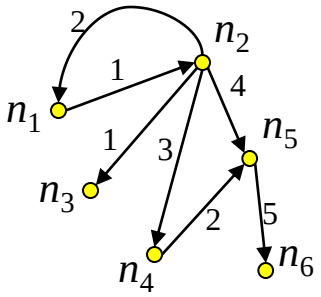

- Shortest path distance used instead of Euclidean distance
- SP(a,b) = path between a and b with the minimum accumulated length

# Challenges

- Euclidean distance is no longer relevant
  - R-tree may not be useful, when search is based on shortest path distance
- Graph cannot be flattened to a one-dimensional space
  - Special storage and indexing techniques for graphs are required
- Graph properties may vary
  - directed vs. undirected
  - length, time, etc. as edge weights

# Modeling Spatial Networks

- Adjacency matrix only appropriate for dense graphs
- Spatial networks are sparse: use adjacency lists instead




graph

| | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ |
|---|---|---|---|---|---|---|
| $n_1$ | 0 | 1 | ∞ | ∞ | ∞ | ∞ |
| $n_2$ | 2 | 0 | 1 | 3 | 4 | ∞ |
| $n_3$ | ∞ | ∞ | 0 | ∞ | ∞ | ∞ |
| $n_4$ | ∞ | ∞ | ∞ | 0 | 2 | ∞ |
| $n_5$ | ∞ | ∞ | ∞ | ∞ | 0 | 5 |
| $n_6$ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 |

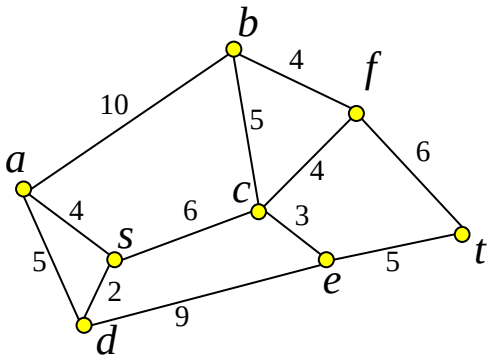
adjacency matrix

| | |
|---|---|
| $n_1$ | ($n_2$, 1) |
| $n_2$ | ($n_1$, 2), ($n_3$, 1), ($n_4$, 3), ($n_5$, 4) |
| $n_4$ | ($n_5$, 2) |
| $n_5$ | ($n_6$, 5) |

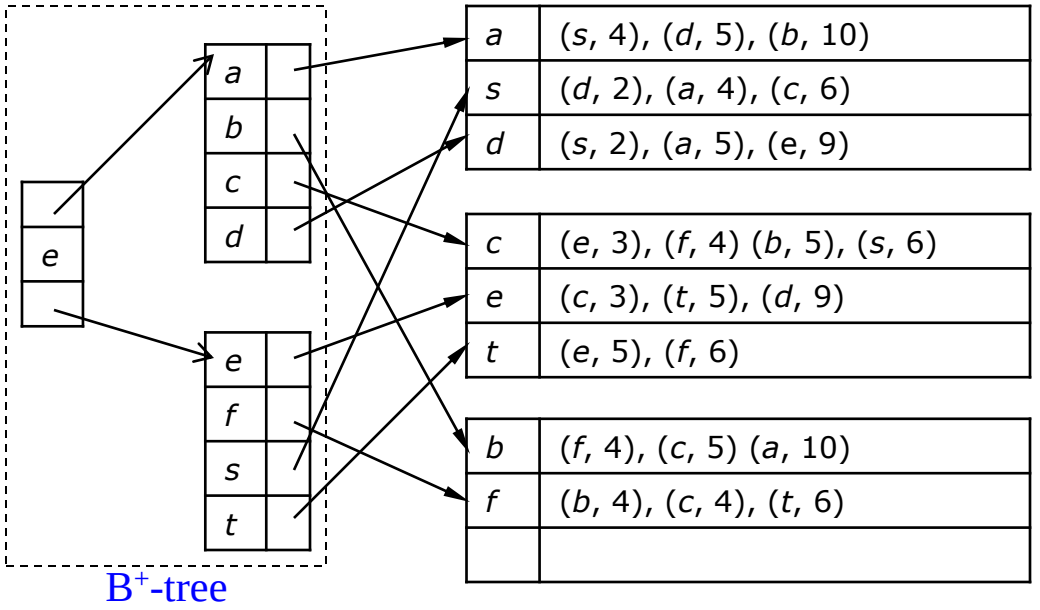
adjacency lists

# Storing Large Spatial Networks

- Problem: adjacency lists representation may not fit in memory if graph is large
- Solution:
  - partition adjacency lists to disk blocks [based on proximity]
  - create B$^+$-tree index on top of partitions [based on node-id]




graph



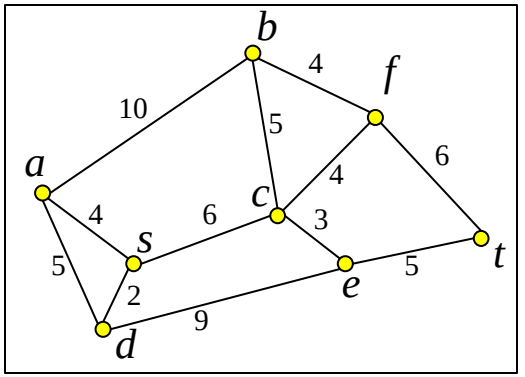

B$^+$-tree

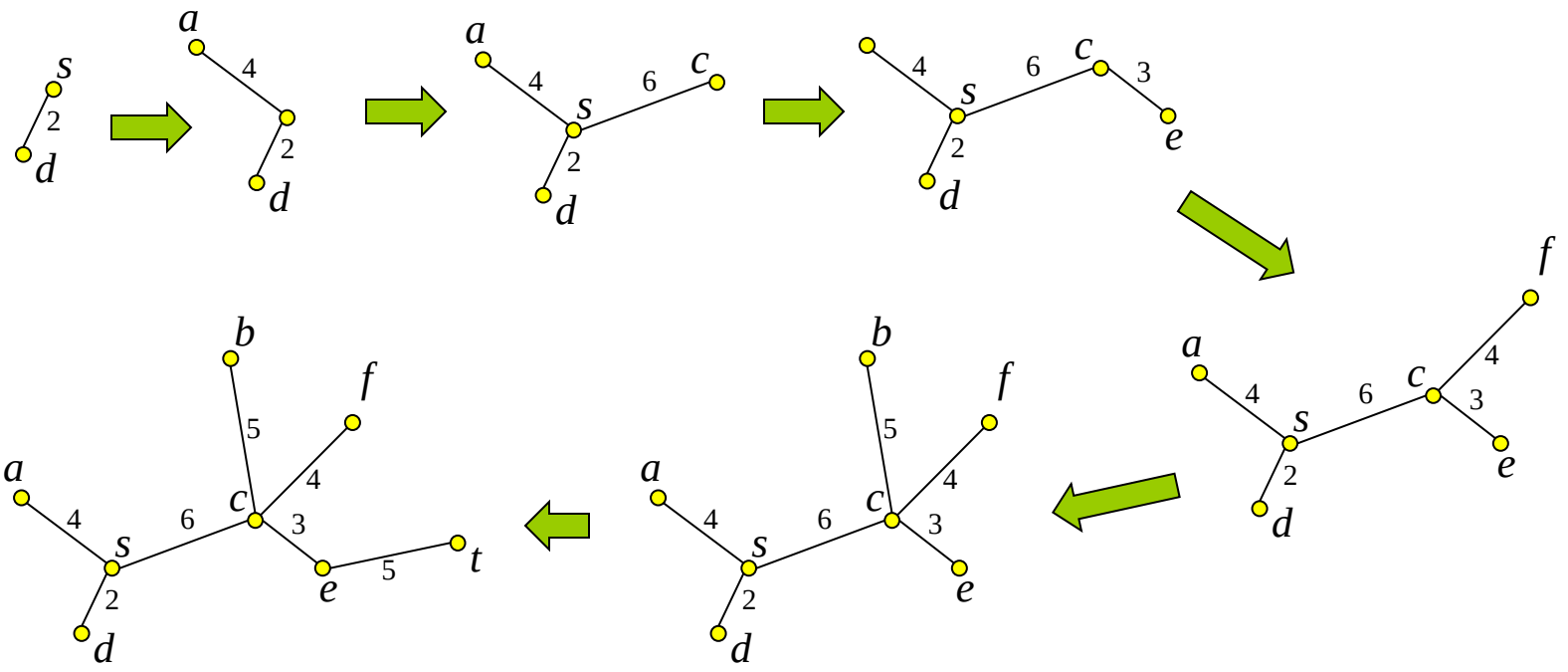
# Shortest Path Computation

- Given a graph G(V,E), and two nodes s,t in V, find the shortest path from s to t
- A classic algorithmic problem
- Studied extensively since the 1950's
- Several methods
  - Dijkstra's algorithm
  - A*-search
  - Bi-directional search

# Dijkstra's Shortest Path Search

- Idea: incrementally explore the graph around s, visiting nodes in distance order to s until t is found (like NN)



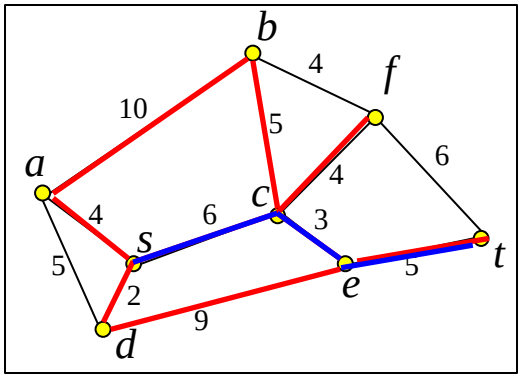

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

# Dijkstra's Shortest Path Search

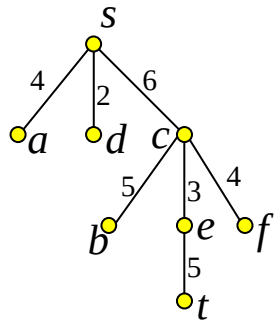
```
function Dijkstra_SP(source node s, target node t)
1.    for each graph node v
2.        SPD(s, v) := ∞; /* initialize shortest path distance */
3.        path(s, v) := null; /* initialize shortest path */
4.        mark v as unvisited;
5.    initialize a priority queue Q;
6.    SPD(s, s) := 0; add s to Q;
7.    while not empty(Q)
8.        v := top(Q); /* node v on Q with smallest SPD(s, v) */
9.        remove v from Q;
10.       mark v as visited;
11.       if v = t then return path(s, t);
12.       for each neighbor u of v
13.           if u is not marked as visited
14.               if SPD(s, u) > SPD(s, v) + weight(v, u)
15.                   SPD(s, u) := SPD(s, v) + weight(v, u);
16.                   path(s, u) := path(s, v) + (v, u);
17.                   add or update u on Q;
```

# Dijkstra's Shortest Path Search

| Current node | Queue |
|---|---|
| *s* (0) | *d* (2), *a* (4), *c* (6) |
| *d* (2) | *a* (4), *c* (6), *e* (11) |
| *a* (4) | *c* (6), *e* (11), *b* (14) |
| *c* (6) | *e* (9), *f* (10), *b* (11) |
| *e* (9) | *f* (10), *b* (11), *t* (14) |
| *f* (10) | *b* (11), *t* (14) |
| *b* (11) | *t* (14) |
| *t* (14) | |



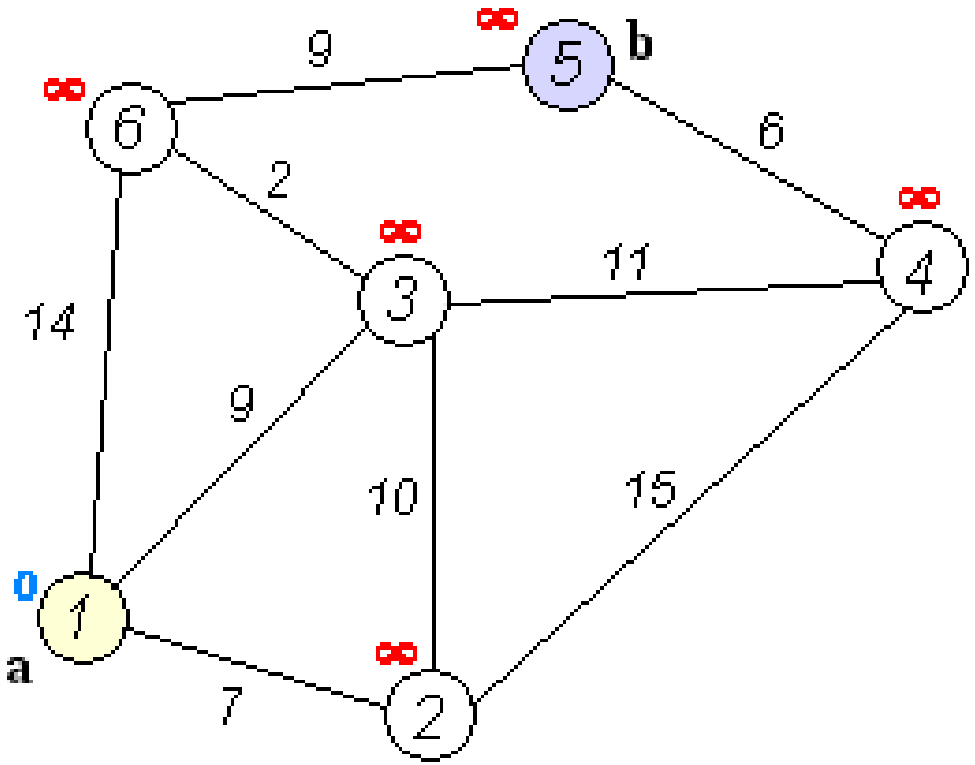

*t* is de-queued: shortest path has been found!




shortest path tree of node *s*

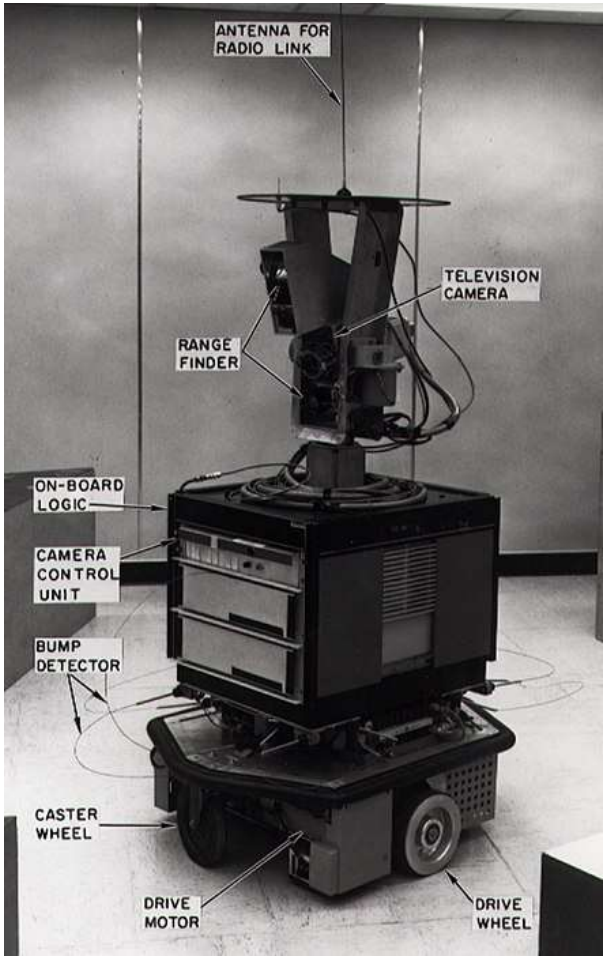
# Illustrating Dijkstra's algorithm

- Find the shortest path between *a* and *b*.
- Worst-case performance $O(|E| + |V| \log |V|)$



# A*-search

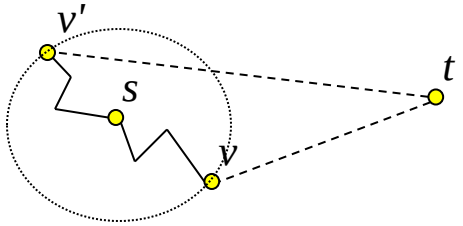
A* was invented by AI researchers working on Shakey the Robot's path planning in 1968.




https://en.wikipedia.org/wiki/A*_search_algorithm

# A*-search

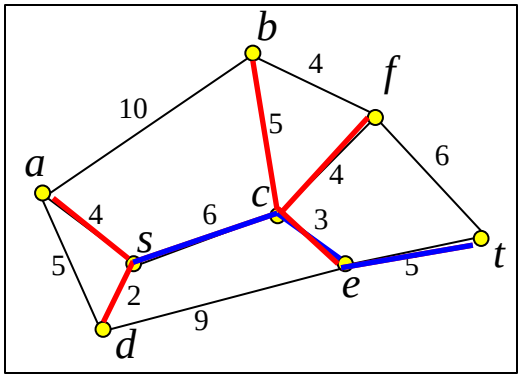
- Dijkstra's search explores nodes around *s* without a specific search direction until *t* is found
- Idea: improve Dijkstra's algorithm by directing search towards *t*
- Due to triangular inequality, Euclidean distance is a lower bound of network distance
- Use Euclidean distance to lower bound network distance based on known information:
  - Nodes are visited in increasing SPD(*s*,*v*)+dist(*v*,*t*) order
    - SPD(*s*,*v*): shortest path distance from *s* to *v* (computed by Dijkstra)
    - dist(v,t): Euclidean distance between *v* and *t*
  - Original Dijkstra visits nodes in increasing SPD(*s*,*v*) order



# A*-search: Example

SPD($s$,$c$)+dist($c$,$t$)

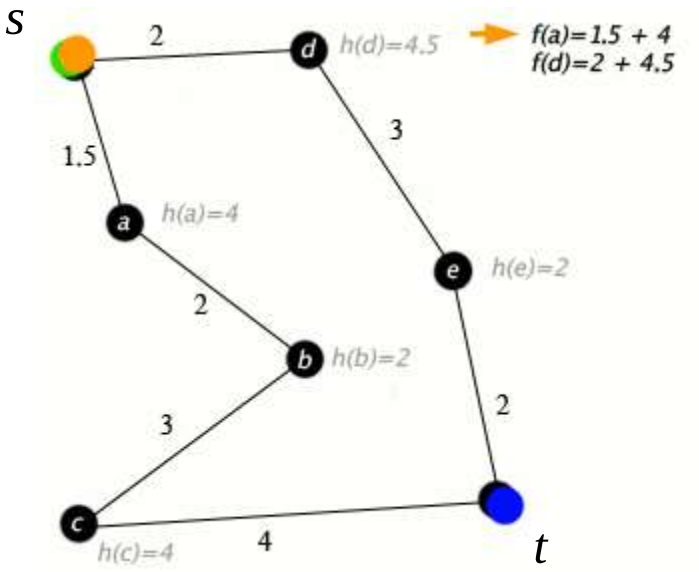
| Current node | Queue |
|---|---|
| $s$ (0) | $d$ (2+14), $a$ (4+15), $c$ (6+7) |
| $c$ (6) | $a$ (4+15), $d$ (2+14), $b$ (11+9), $f$ (10+6), $e$ (9+5) |
| $e$ (9) | $a$ (4+15), $d$ (2+14), $b$ (11+9), $f$ (10+6), $t$ (14) |
| $t$ (14) | |




$t$ is de-queued: shortest path has been found!

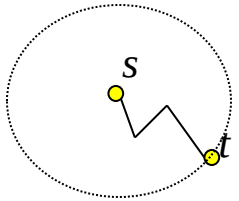
# Illustrating A* search algorithm

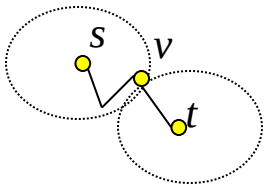
- Find the shortest path between *s* and *t*.
  *f(p) = Dijkstra_dist(s, p) + Euclidean_dist(p, t)*



# Bi-directional search

- Dijkstra's search explores nodes around *s* without a specific search direction until *t* is found
- Idea: search can be performed concurrently from *s* and from *t* (backwards)
- The shortest path tree of s and the (backward) shortest path tree of t are computed in concurrently
  - One queue $Q_s$ for forward and one queue $Q_t$ for backward search
  - Node visits are prioritized based on min(SPD(s,v), SPD(v,t))
  - If v already visited from s and v is in $Q_t$, then candidate shortest path: p(s,v)+p(v,t) [if v already visited from t and v in $Q_s$ symmetric]
  - If v is visited by both s and t terminate search; report best candidate shortest path

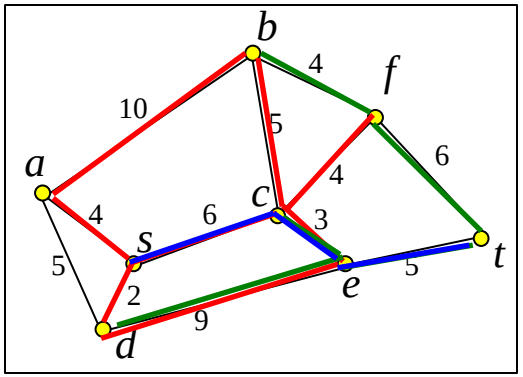


# Bi-directional search: Example

| Current node | $Q_s$ | $Q_t$ |
|---|---|---|
| *s* (0), t (0) | *d* (2), *a* (4), *c* (6) | *e* (5), *f* (6) |
| *d* (2) [s] | *a* (4), *c* (6), *e* (11) | *e* (5), *f* (6) |
| *a* (4) [s] | *c* (6), *e* (11), *b* (14) | *e* (5), *f* (6) |
| *e* (5) [t] | *c* (6), *e* (11), *b* (14) | *f* (6), *c* (8), *d* (14) |
| *c* (6) [s] | *e* (9), *f* (10), *b* (11) | *f* (6), *c* (8), *d* (14) |
| *f* (6) [t] | *e* (9), *f* (10), *b* (11) | *c* (8), *b* (10), *d* (14) |
| *c* (8) [t] | | |

*c* is visited from both *s* and *t*!
terminate and report shortest path




*candidate shortest path:*
*s ➔ d ➔ e ➔ t (16)*

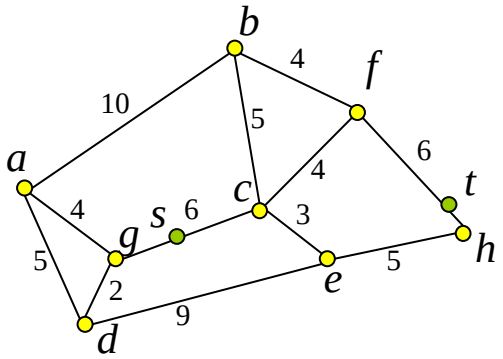
*candidate shortest path:*
*s ➔ c ➔ e ➔ t (14)*

# Discussions

- A* and bi-directional search can be combined to powerful search techniques
- A* can only be applied if lower distance bounds are available
- All versions of Dijkstra's search require non-negative edge weights
  - Bellman-Ford is an algorithm for arbitrary negative edges

# Source/Destination on Edges

- We have assumed that points s and t are nodes of the network
- In practice s and t could be arbitrary points on edges
  - Mobile user locations
- Solve problem by introducing 2 more nodes

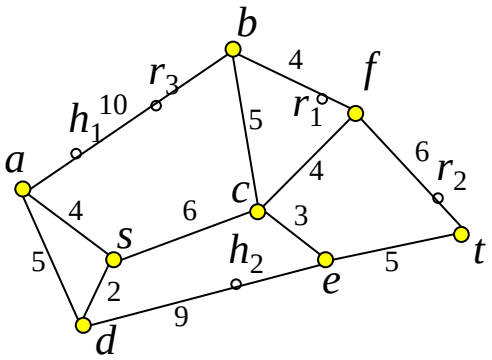


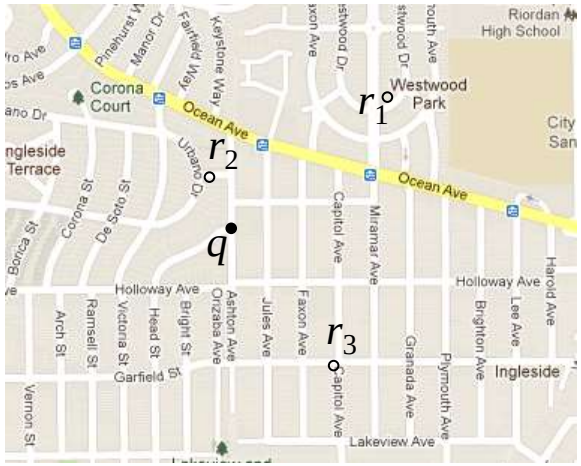

- en-heap g (2) and c (4) first

- t is reached from f or h

# Spatial Queries over Spatial Networks

- Data:
  - A (static) spatial network (e.g., city map)
  - A (dynamic) set of spatial objects
- Spatial queries based on network distance:
  - Selections. Ex: find gas stations within 10km driving distance from here
  - Nearest neighbor search. Ex: find k nearest restaurants from present position
  - Joins. Ex: find pairs of restaurants and hotels at most 100m from each other



# Methodology

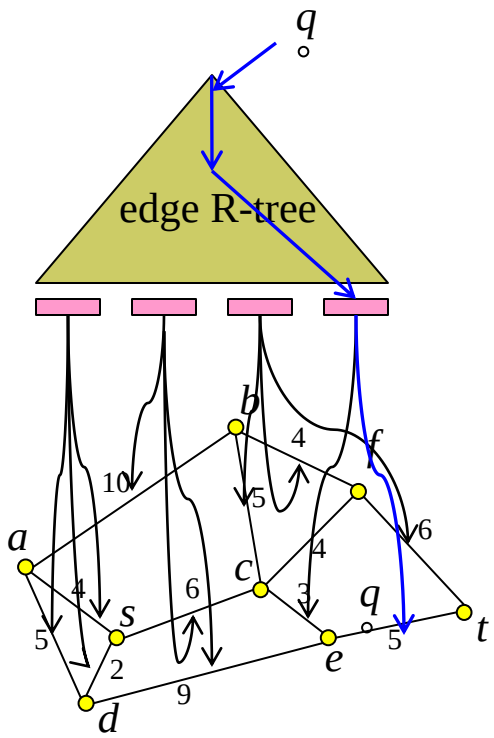
- Store (and index) the spatial network
  - Graph component (indexes connectivity information)
  - Spatial component (indexes coordinates of nodes, edges, etc.)
- Store (and index) the sets of spatial objects
  - Ex., one spatial relation for restaurants, one spatial relation for hotels, one relation for mobile users, etc.
- Given a spatial location p, use spatial component of network to find the network edge containing p
- Given a network edge, use network component to traverse neighboring edges
- Given a neighboring edge, use spatial indexes to find objects on them

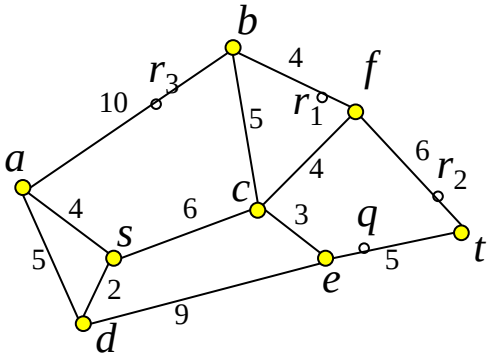
# Evaluation of Spatial Selections (1)

- Query: find all objects in spatial relation R, within network distance ε from location q
- Method:
  - Use spatial index of network (R-tree indexing network edges) to find edge $n_1n_2$, which includes q
  - Use adjacency index of network (graph component) and apply Dijkstra's algorithm to progressively retrieve edges that are within network distance ε from location q
  - For all these edges apply a spatial selection on the R-tree that indexes R to find the results

# Evaluation of Spatial Selections (1)

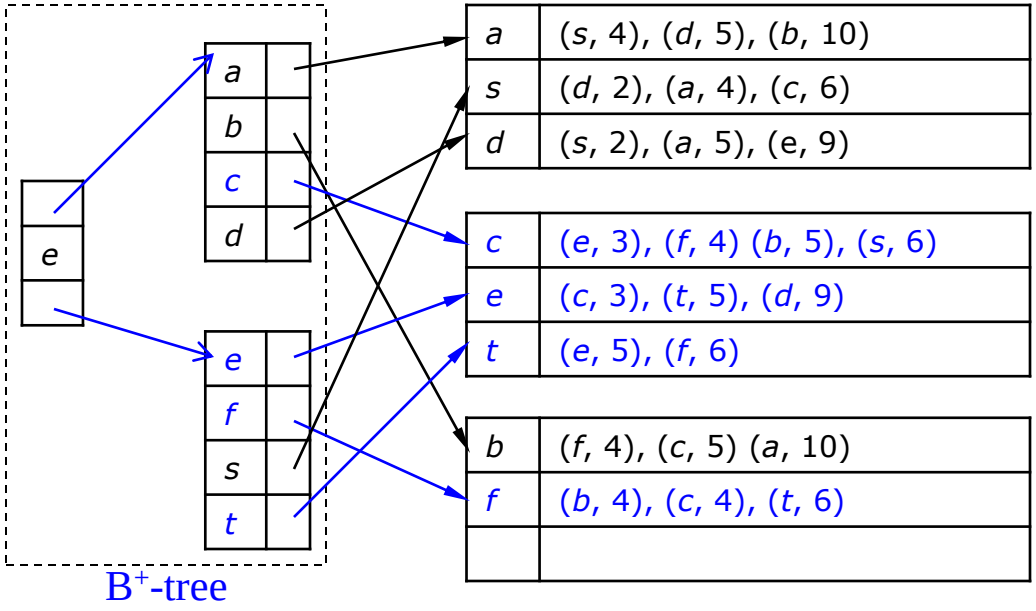
- Example: Find restaurants at most distance 10 from q
- Step 1: find network edge which contains q

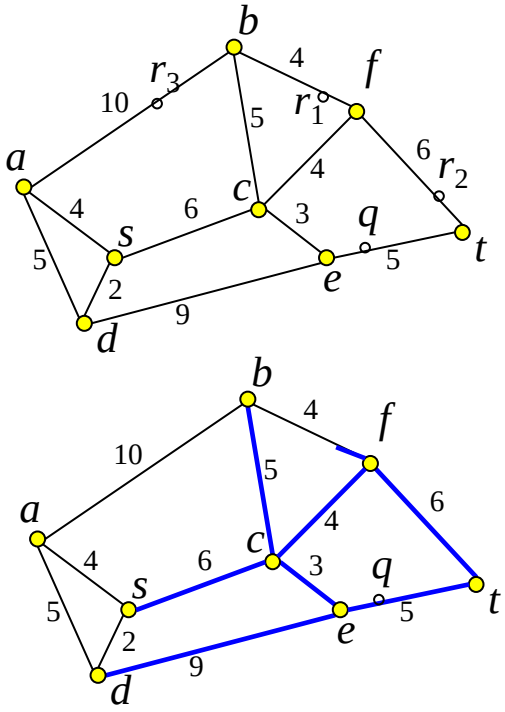


# Evaluation of Spatial Selections (1)

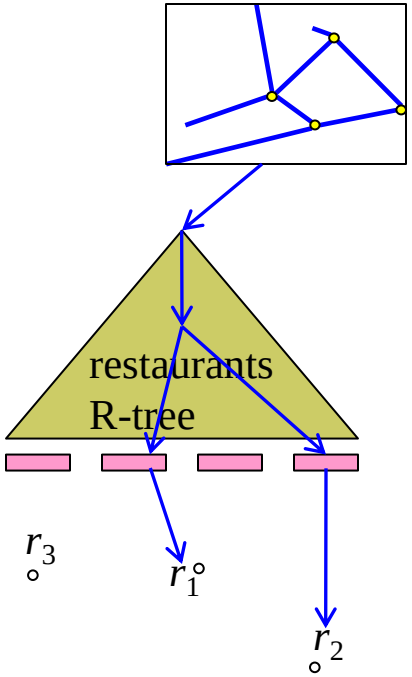
- Example: Find restaurants at most distance 10 from q
- Step 2: traverse network to find all edges (or parts of them within distance 10 from q)

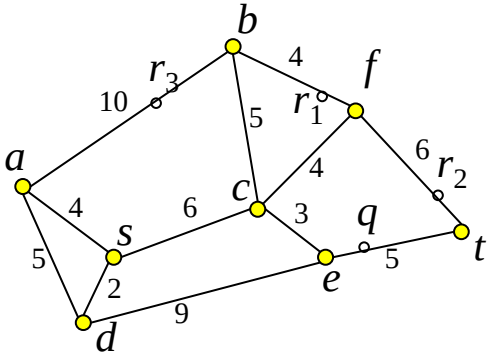


# Evaluation of Spatial Selections (1)

- Example: Find restaurants at most distance 10 from q
- Step 3: find restaurants that intersect the subnetwork computed at step 2



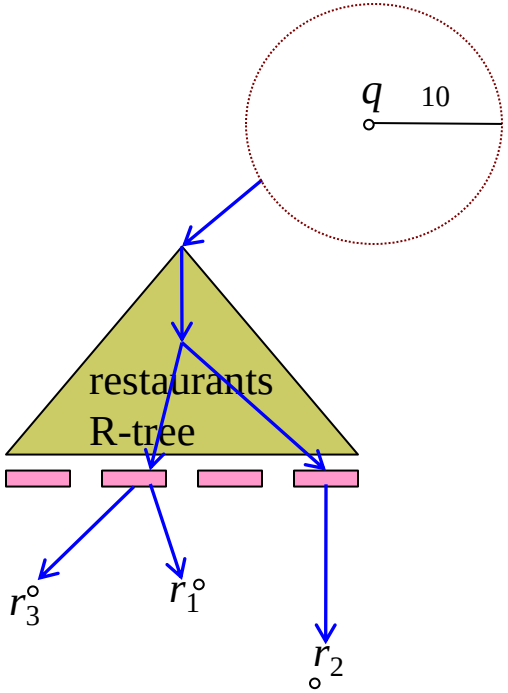



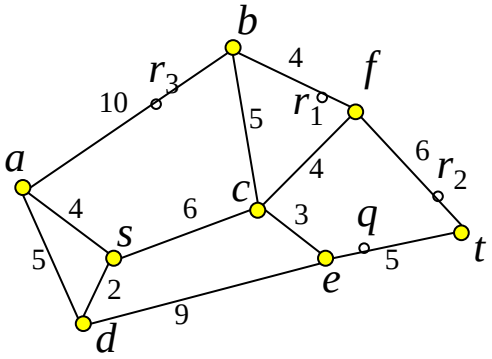
# Evaluation of Spatial Selections (2)

- Query: find all objects in spatial relation R, within network distance ε from location q
- Alternative method based on Euclidean bounds:
  - Assumption: Euclidean distance is a lower-bound of network distance:
    - dist(v,u) ≤ SPD(v,u), for any v,u
  - Use R-tree on R to find set S of objects such that for each o in S: dist(q,o) ≤ ε
  - For each o in S:
    - find where o is located in the network (use Network R-tree)
    - compute SPD(q,o) (e.g. use A*)
    - If SPD(q,o) ≤ ε then output o

# Evaluation of Spatial Selections (2)

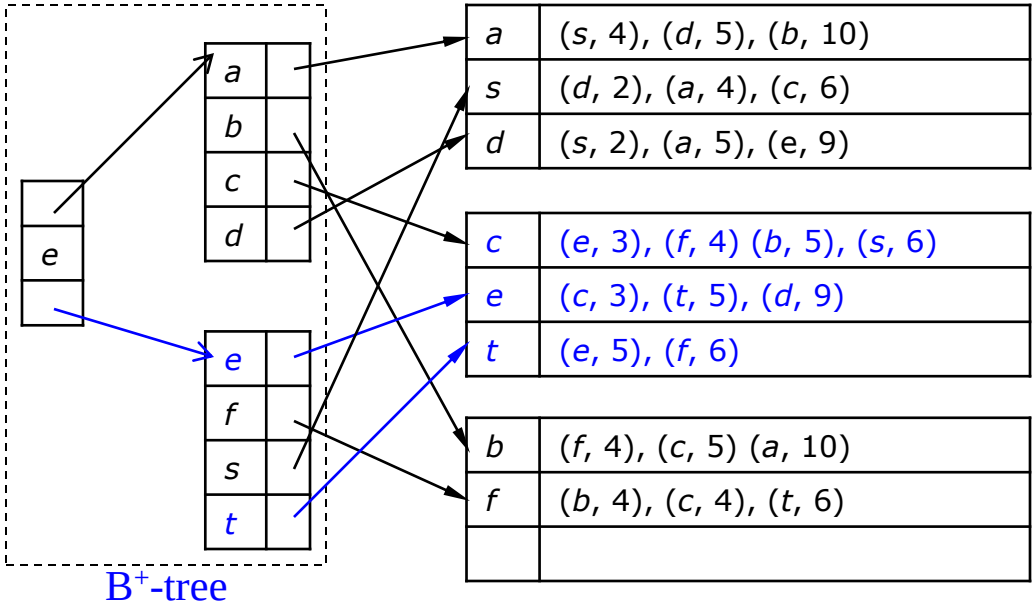
- Example: Find restaurants at most distance 10 from q
- Step 1: find restaurants for which the Euclidean distance to q is at most 10: S={$r_1$,$r_2$,$r_3$}

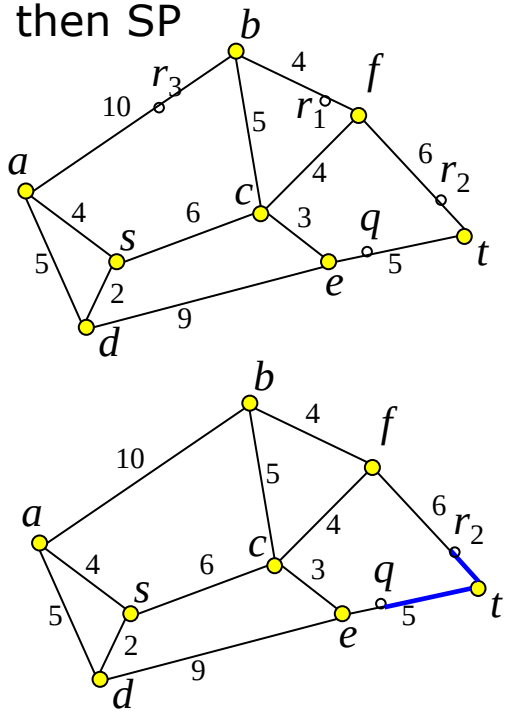


# Evaluation of Spatial Selections (2)

- Example: Find restaurants at most distance 10 from q
- Step 2: for each restaurant in S, compute SPD to q and verify if it is indeed a correct result
  - Ex. for $r_2$, first find where $r_2$ is located (edge f,t), then SP



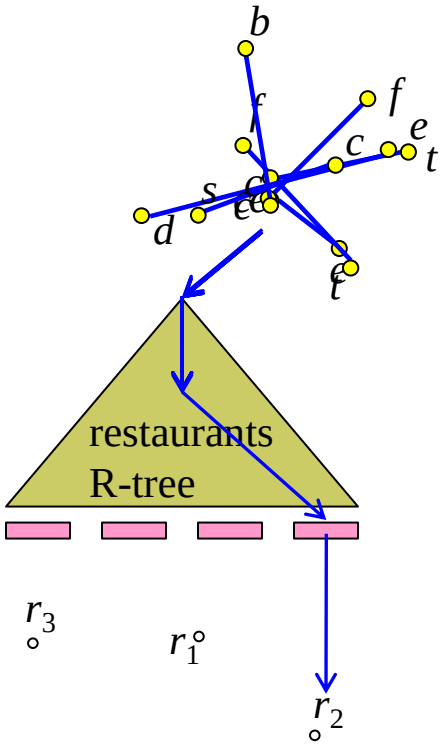



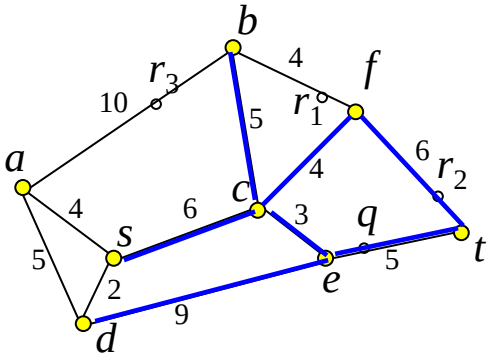
# Evaluation of NN search (1)

- Query: find in spatial relation R the nearest object to a given location q
- Method:
  - Use spatial index of network (R-tree indexing network edges) to find edge $n_1n_2$, which includes q
  - Use adjacency index of network (graph component) and apply Dijkstra's algorithm to progressively retrieve edges in order of their distance to q
  - For each edge apply a spatial selection on the R-tree that indexes R to find any objects
  - Keep track of nearest object found so far; use its shortest path distance to terminate network browsing

# Evaluation of NN search (1)

- Example: Find nearest restaurant to q




*current NN* = $r_2$
*SPD*($q$, $r_2$)=5 √

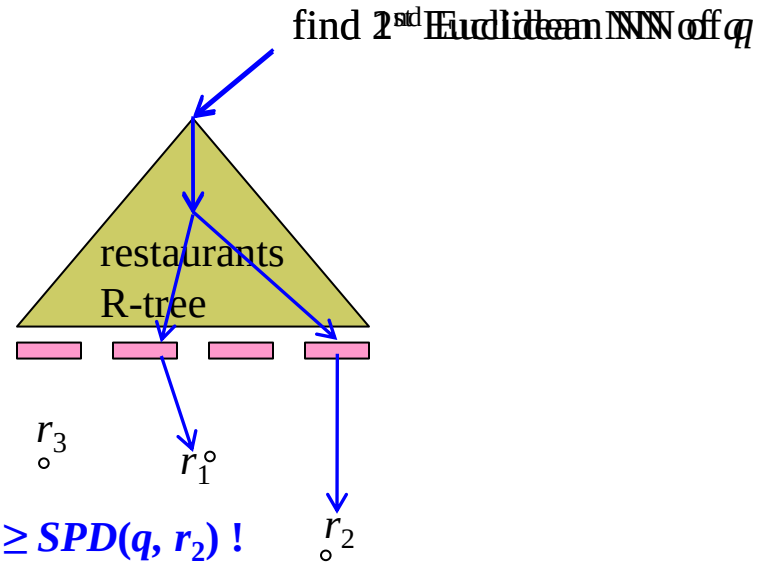
# Evaluation of NN search (2)

- Query: find in spatial relation R the nearest object to a given location q
- Alternative method based on Euclidean bounds:
  - Assumption: Euclidean distance lower-bounds network distance:
    - dist(v,u) ≤ SPD(v,u), for any v,u

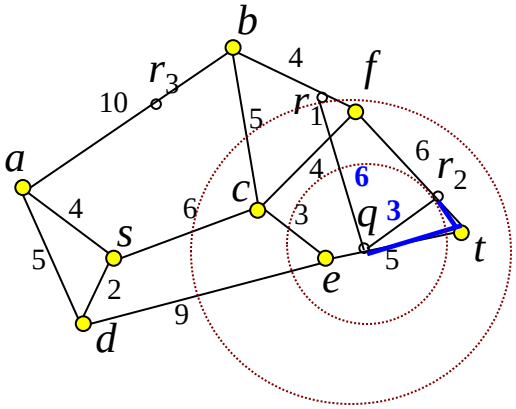
1. Use R-tree on R to find Euclidean NN $p_{E1}$ of q;
2. CurrentNN= $p_{E1}$; bound=SPD(q,$p_{E1}$); //e.g. use A*
3. Find next Euclidean NN $p_{Ei}$ of q
4. If dist(q,Ei)≥bound, then
   report (CurrentNN,bound) as result;
5. Compute SPD(q,$p_{Ei}$); if bound>SPD(q,$p_{Ei}$) then
   CurrentNN= $p_{Ei}$; bound=SPD(q,$p_{Ei}$);
6. Goto step 3

# Evaluation of NN search (2)

- Example: Find nearest restaurant to q




***dist*(q, $r_1$) ≥ *SPD*(q, $r_2$) !**




*current NN* = $r_2$
*SPD*(q, $r_2$)=5 √

# Spatial Join Queries

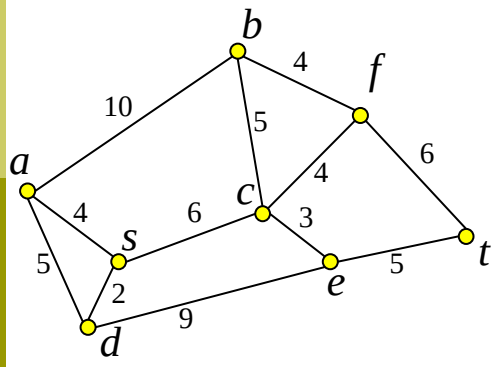
- Query: find pairs (r,s), such that r in relation R, s in relation S, and SPD(r,s)≤ε
- Methods:
  - For each r in R, do an ε-distance selection queries for objects in S (Index Nested Loops)
  - For each pair (r,s), such that Euclidean dist(r,s)≤ε compute SPD(r,s) and verify SPD(r,s)≤ε

# Notes on Query Evaluation based on Network Distance

- For each query type, there are methods based on network browsing and methods based on Euclidean bounds
- Network browsing methods are fast if network edges are densely populated with points of interest
  - A limited network traversal can find the result fast
- Methods based on Euclidean bounds are good if the searched POIs are sparsely distributed in the network
  - Few verifications with exact SP searches are required
  - Directed SP search (e.g. using A*) avoids visiting empty parts of the network

# Shortest Path Materialization and Indexing in Large Graphs

- Dijkstra's algorithm and related methods could be very expensive on very large graphs
- (Partial) materialization of shortest paths in static graphs can accelerate search




graph

| | *a* | *b* | *c* | *s* | … |
|---|---|---|---|---|---|
| *a* | 0, - | 10, *ab* | 10, *asc* | 4, *as* | … |
| *b* | 10, *ba* | 0, - | 5, *bc* | 11, *bcs* | … |
| *c* | 10, *csa* | 5, *cb* | 0, - | 6, *cs* | … |
| *s* | 4, *sa* | 11, *scb* | 6, *sc* | 0, - | … |
| … | … | … | … | … | … |

brute-force materialization
$O(n^3)$ space, $O(1)$ time

| | *a* | *b* | *c* | *s* | … |
|---|---|---|---|---|---|
| *a* | 0, - | 10, *b* | 10, *s* | 4, *s* | … |
| *b* | 10, *a* | 0, - | 5, *c* | 11, *c* | … |
| *c* | 10, *s* | 5, *b* | 0, - | 6, *s* | … |
| *s* | 4, *a* | 11, *c* | 6, *c* | 0, - | … |
| … | … | … | … | … | … |

distance matrix with successors
$O(n^2)$ space, $O(n)$ time

# Summary

- Indexing and search of spatial networks is different than spatial indexing
  - Shortest path distance is used instead of Euclidean distance, to define range queries, nearest neighbor search, and spatial joins
- Spatial networks could be too large to fit in memory
  - Disk-based index for adjacency lists is used
- Several shortest path algorithms
- Spatial queries can be evaluated using Euclidean bounds