Data Structures and Algorithms

Spring 2009-2010

Outline

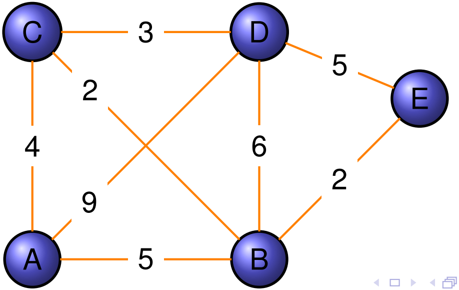
1. Graph Algorithms
   - Shortest-Path Algorithms

Outline

1. Graph Algorithms
   - Shortest-Path Algorithms

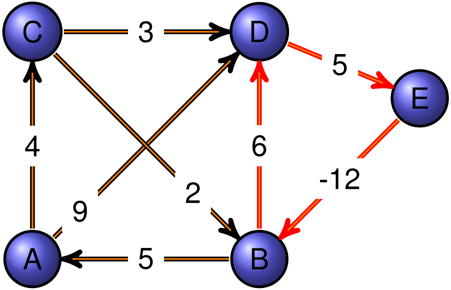
Problems for Google Maps

- Given a weighted graph, find the shortest path (SP) between any one of
  1. two given nodes (locations)
  2. a node and every other node (*Single-Source SP*)
  3. every pair of nodes (*All-Pairs SP*)
- Edge weight: to every edge (v_i, v_j), we associate a weight or, *cost*, $c_{i,j}$
- Then cost of a path $v_1, v_2, \ldots, v_n$ is $\sum_{i=1}^{n-1} c_{i,i+1}$



Related Problems

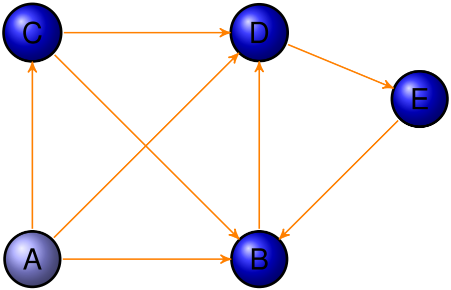
- If the graph was unweighted then
  - the cost of a path would be $\sum_{i=1}^{n-1} 1 = n - 1$
  - a *breadth-first search* (BFS) starting at the given node will solve problems 1 and 2 above
- Negative weights can cause problems and require special care in algorithms



- When a graph has a negative weight, we say that shortest paths are not defined

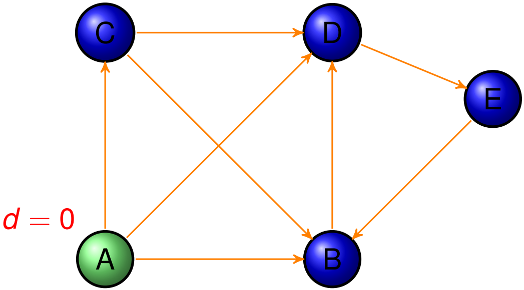
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



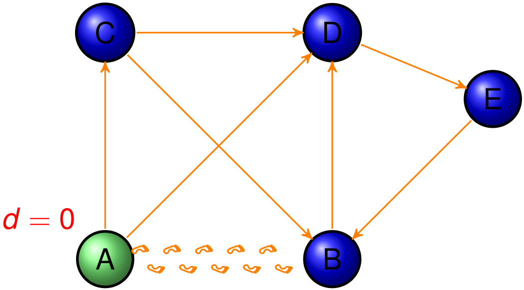
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



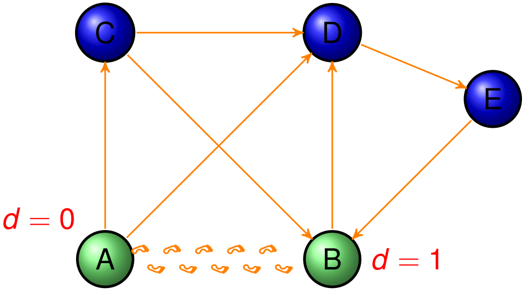
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



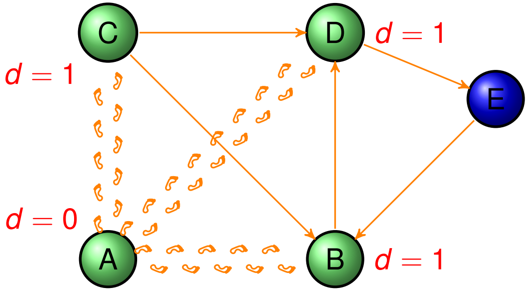
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



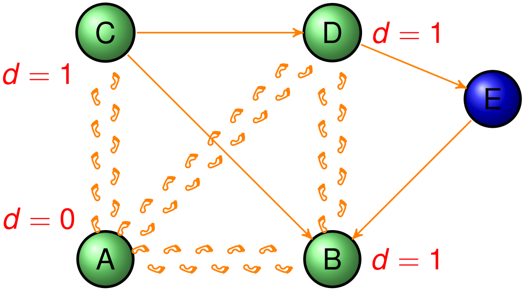
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



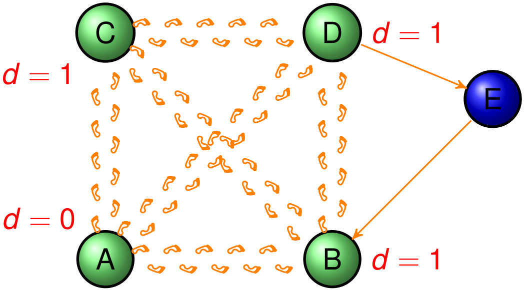
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



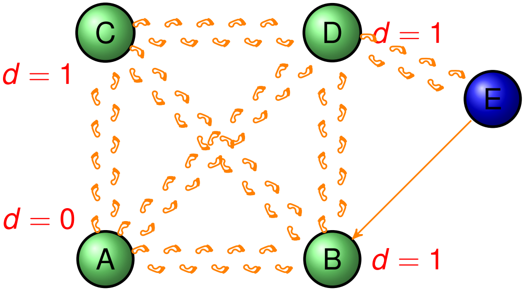
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, *s*, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from *s*, one level at a time
- Example: find SP, d, from A to every other node



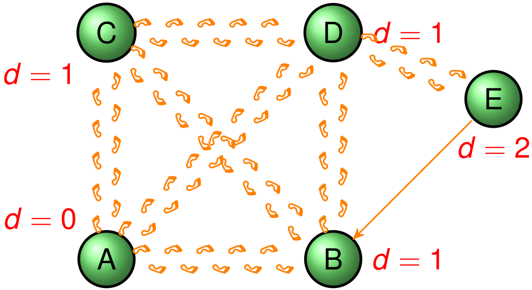
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



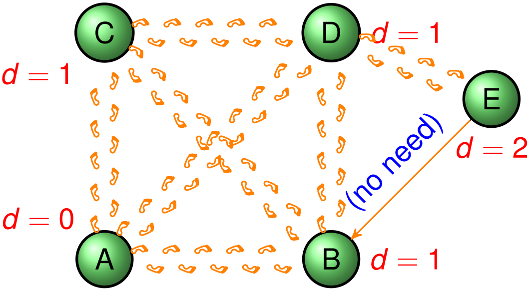
Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



Unweighted Shortest Path

- Given a graph, we want to find SP from a vertex, s, to every other vertex; all edge weights $= 1$
- Use **breadth-first search** to spread out from s, one level at a time
- Example: find SP, d, from A to every other node



Unweighted Shortest Path (contd.)

- We “spread out” from a vertex to all those vertices adjacent to it, ignoring them if we have seen them before. (Why?)
- We call this a breadth-first search (BFS) strategy
- For each node v we keep
  - Its distance, d_v, from s (`dist[v]` on next slide)
  - Its predecessor, p_v, in SP
- Running time (of BFS and SP) is $O(|V| + |E|)$

Unweighted Shortest Path (contd.)

- Code for BFS here
- From a vertex, v, record all of v's adjacancies by *queue*ing them on `Q`
- Since v is at distance d_v from s, then v's *new* (not seen before) adjacancies must be at $d_v + 1$
- Record these distances also
- LEDA algorithm BFS over returns a (non-unique) BFS ordering of vertices
- Would need to include an array of nodes indexable by node, say

  ```
  node_array<node> pred;
  ```

  to remember the predecessor of node on optimal path
- With above declaration, can ask what node is the predecessor of nove `v` with

  ```
  node v;
     :
  ```