Data Structures and Algorithms

Spring 2009-2010

Outline

1. Graph Algorithms
   - Weighted Shortest-Path Algorithms
   - Depth First Search

Outline

Negative Costs on Edges

- Dijkstra's Algorithm works because on every iteration of the `while(! PQ.empty())` loop, the cheapest path to the node taken from the PQ is known
- This doesn't doesn't work for negative weights
- Paths to vertices are optimal with respect to *positive* weights
- A (very) negative weight arc from some vertex could come much later and give a cheaper path
- Seems like we cannot rely on knowing when we have an optimal path to a vertex...
- A brute force algorithm is needed apparently that recomputes new costs to all of the places we could get from a newly explored vertex

Negative Costs on Edges (contd.)

- Brute Force Algorithm: keep a queue, *Q*, of vertices that have been encountered to date, with the possibility that a vertex may be *enqueue*d and *dequeue*d many times

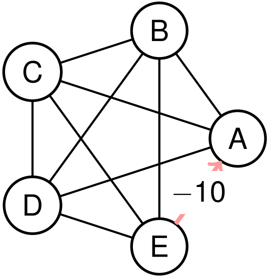
```
Q.enqueue(s);    // start vertex
while (! Q.empty()) {
  v = Q.dequeue();
  forall_adj_edges(e,v) {
    w = target(e);    // v = head(e);
    if (dist[w] > dist[v] + cost[e]) {
      dist[w] = dist[v] + cost[e];
      pred[w] = e; // edge that got us to w
      if (! Q.contains_already(w))
         Q.enqueue(w);
    }
  }
}
```

Negative Costs on Edges (contd.)

- Algorithm works as long as no cycles of negative cost exist (since repeated cycling of these edges keeps lowering the cost)
- The algorithm amounts to verifying that as a result of looking at an edge a cheaper path doesn't now exist to the target of that edge
- If there does, then we must (re)consider all of the target's adjacencies by putting the target (back) on the queue
- The absolute maximum no. of times that a vertex can be dequeued is $|V|$ times – once for every other vertex having an edge pointing into it

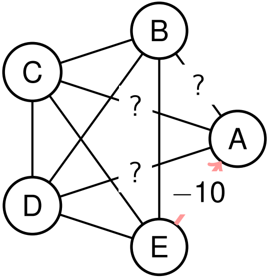
Negative Costs on Edges (contd.)

- Running time is $O(|E||V|)$ since, in the worst case, each new edge may cause us to have to revise the distances to every vertex as shown below on K_5, the complete graph on 5 vertices



- Although A may have been explored already, a cheaper way of getting to it (via E) means we have to reconsider all edges leaving A

Negative Costs on Edges (contd.)



- Can improve significantly on this poor bound as long as the graph doesn't have a directed cycle

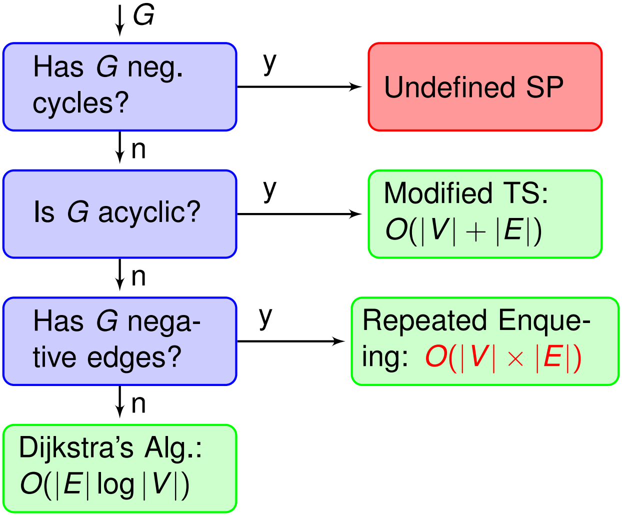
Dijkstra's Algorithm: Special Case

- If graph is *acyclic* then a natural ordering presents itself in which to process the vertices – even if graph has negative weights
- By processing vertices in topological ordering there can be no incoming arcs when we examine the costs of vertices adjacent to this vertex, v
- Algorithm can be implemented by making small modifications to topological sort (TS) algorithm to compare costs of using $e = (v, w)$ to get from v to w versus some prior path
- Therefore, running time of shortest path algorithm is $O(|V| + |E|)$ when graph is acyclic: no need for priority queue
- Need to know if graph is DAG for this to work

Dijkstra's Algorithm: Special Case

- Given an arbitrary graph in which you want to perform a shortest path computation, test graph for *acyclicity* first, then run modified TS (if DAG) or Dijkstra's (if no neg. arcs) or the $O(|E||V|)$ if it has neg. arcs
- But algorithm to test for acyclicity is *also* based on TS so we can combine acyclicity testing with this special case of Dijkstra's algorithm
- If graph turns out to be cyclic then we abandon and revert to full-blown Dijkstra's algorithm

SP Algorithm Summary



Critical Path Analysis

- Critical Path Analysis (CPA) finds the critical jobs for the completion of large projects
- However, this time we want the *longest* path from start, s, node to finish node, f
- In this case we will have problems if we have *positive-cost* cycles

All-Pairs Shortest Path

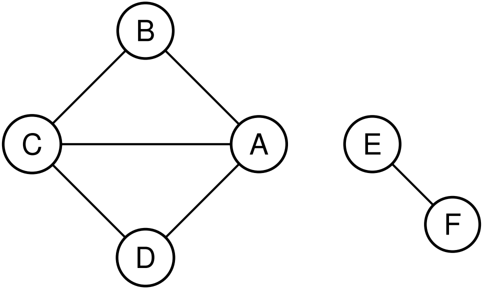
- Have seen algorithms to compute *source-to-all* SP
- What if we want to compute all shortest paths – from every node to every other node?
- Cannot do *much* better than running Dijkstra's algorithm $|V|$ times for each node in turn as source although for *dense* graphs there are some improvements (see APSP links on resources web page)

Outline

An Alternative to Breadth First Search

- Depth First Search (DFS) – like Breadth First Search (BFS) – is a mechanism for *systematically* visiting / searching/ processing all vertices of a graph
- While BFS spreads out from the start node a level at a time, DFS follows a path as deeply into the graph as it can
- DFS generalizes *pre-order* or *post-order* traversals of trees
- We can (sort of) think of a graph in the following recursive way: a graph comprises a node connected to a graph by edges
- Idea of DFS (loosely): to DFS a graph select a vertex, *process* it and, from this, DFS the remaining graph

An Alternative to Breadth First Search (contd.)



- Our choice of vertex selection strategy might be to move to the lexicographically smallest neighbour
- Starting with vertex A we process it and move to vertex B; process that and move to the next unprocessed vertex...
- We need to remember that we have already visited a node so that we do not enter an infinite loop

An Alternative to Breadth First Search (contd.)

```
void DFS(Vertex v)
{
   if (visited[v]) return; // already seen

   visited[v] = true;
   // do whatever ``processing'' on vertex here
   forall_adj_nodes(w, v)
   {
      if ( !visited[w])
         DFS(w);
   }
}
```

An Alternative to Breadth First Search (contd.)

- To DFS an entire graph $G = (V, E)$ we would need to have something like:

```
forall_nodes(v, V)
    DFS(v);   // do a DFS on every node in G
```

- The running time of the *entire* search procedure is $O(|V| + |E|)$
- We can easily tell if a graph is *connected* by picking any vertex, v, and calling `DFS(v)`; if every node is marked visited after one single call then every node was reachable from v