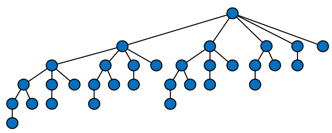
Lecture 22. Shortest Paths

CpSc 8400: Algorithms and Data Structures
Brian C. Dean



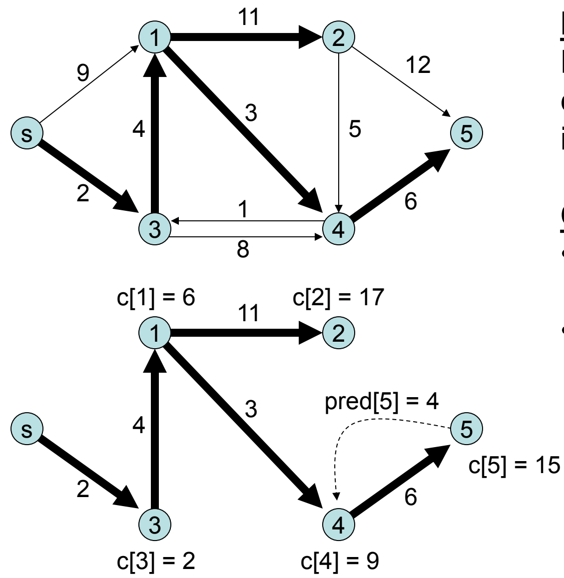
School of Computing
Clemson University

Spring, 2016

Types of Shortest Path Problems

- **Single-source single-destination:** find the shortest path from s to t in a (directed) graph.
- **Single-source:** find the shortest path from s to every other node in a (directed) graph.
- **All-pairs:** find the shortest path from every node to every other node.

Single-Source Shortest Paths



Input:
Directed graph with costs/weights/lengths on its edges.

Output:
- c[j]: cost of shortest s $\rightarrow$ j path
- pred[j]: previous node (before j) on a shortest s $\rightarrow$ j path.

Single-Source Shortest Paths: Algorithms

- Easy cases that we can solve in linear time:
  - Unweighted graphs: use **breadth-first search**.
  - Directed acyclic graphs (DAGs): use **dynamic programming**.
- If edge costs are <u>nonnegative</u>, we'll use **Dijkstra's algorithm**.
- If some edge costs are negative, we use the **Bellman-Ford algorithm**.

Breadth-First Search

- Depth-first search dives as deeply as possible into a graph until it can go no further, then it backs up and tries to branch.
- In contrast, a **breadth-first search** (BFS) starting at some source node s will visit s, then all nodes 1 hop away from s, then all nodes 2 hops away from s, etc.
- Like DFS, we can also use BFS to find connected components or answer “find a path from i to j” queries.
  – BFS finds a path from i to j having fewest edges.

Breadth-First Search

```
BFS(s):
   For all nodes i:
      pred[i] = null
      dist[i] = Infinity
   Q = {s}
   dist[s] = 0
   While Q is nonempty:
        i = next node in Q
        For all nodes j such that (i,j) is an edge:
           If dist[j] = Infinity,
              pred[j] = i
              dist[j] = dist[i] + 1
              Append j to the end of Q
```

- If Q is implemented as a stack rather than as a FIFO queue, we get DFS rather than BFS!

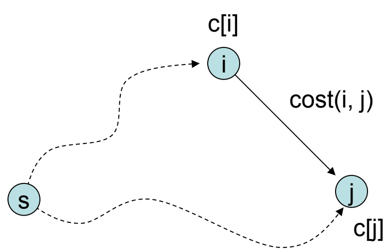
Negative-Cost Cycles

- If negative-cost edges exist:
  - We need to use the O(mn) Bellman-Ford algorithm.
  - We can only find shortest paths if there are no negative-cost directed cycles (Bellman-Ford can at least detect whether negative cycles exist).
  - If our graph is undirected, we typically assume it has no negative-cost edges to begin with.
- Longest path problems convert to shortest path problems by negating edge lengths, so positive-cost cycles are bad for these problems
  - So longest path problems are rarely tractable, since most graphs have positive-cost cycles.
  - Except DAGs, that is…

The Triangle Inequality

- Shortest path costs satisfy the triangle inequality:

  c[i] + cost(i, j) ≥ c[j] for every edge (i, j).



- **Shortest path optimality conditions**: If we have a set of valid shortest path cost labels c[j] that satisfy the triangle inequality for every edge, then these costs must be optimal.

Fixing the Triangle Inequality

- If we ever notice an edge (i, j) that violates the triangle inequality:

  $$c[i] + cost(i, j) < c[j]$$

  then we ought to fix things by reducing c[j]:

```
Tighten(i, j):  (also called Relax(i, j))
   If c[i] + cost(i, j) < c[j],
      c[j] = c[i] + cost(i, j)
      pred[j] = i
```

- Most shortest path algorithms start with c[s] = 0 and c[j ≠ s] = +∞, and repeatedly tighten (relax) edges violating the Δ inequality.
  - This guarantees a set of valid cost labels c[j] that will be optimal when we terminate.

9

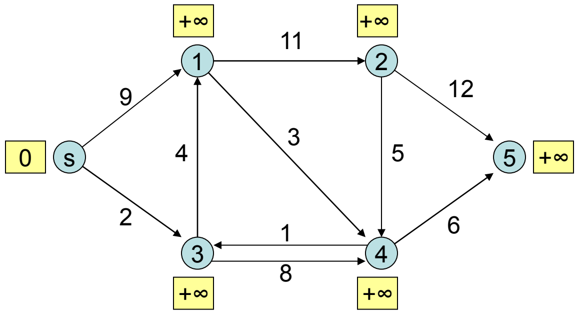
Dijkstra's Algorithm

```
Dijkstra:
   c[s] = 0, c[j ≠ s] = +∞
   Build a priority queue Q on the c[j]'s
   While Q is nonempty:
      Remove from Q the node i having minimum label c[i]
           (at this point we know c[i] is correct)
      Tighten all edges emanating from i
```

- Only works if edge costs are nonnegative.
- Running time depends on priority queue:
  - n inserts (during initialization)
  - n remove-mins
  - at most m decrease-keys
- Using a binary heap, all these opeartions run in O(log n) time, so total is O(m log n).

10

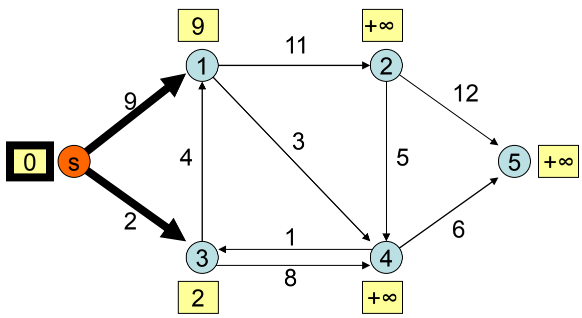
Dijkstra's Algorithm – Example



Initialization.

11

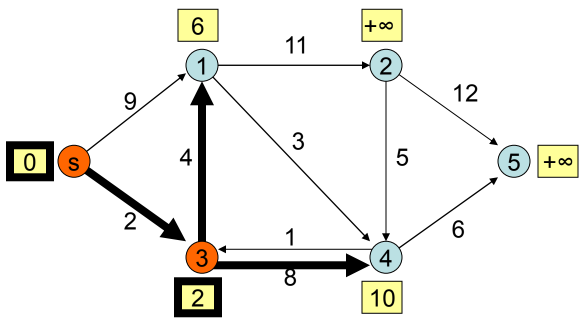
Dijkstra's Algorithm – Example



Examine the source node s, so s becomes "permanently" labeled.
Tighten all edges leaving s.

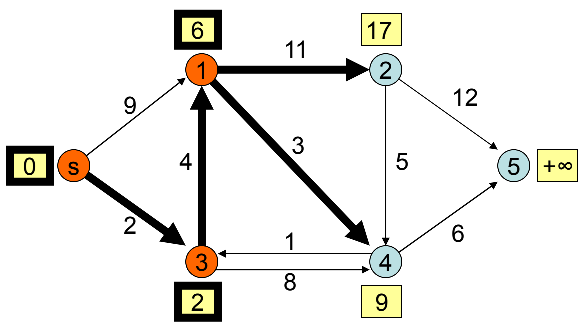
12

Dijkstra's Algorithm – Example



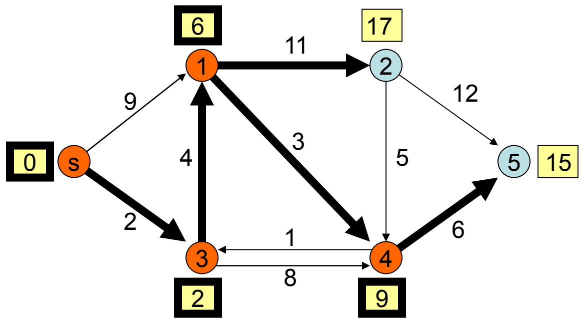
Examine node 3, which becomes permanently labeled.
Tighten all edges leaving node 3.

Dijkstra's Algorithm – Example



Examine node 1, which becomes permanently labeled.
Tighten all edges leaving node 1.

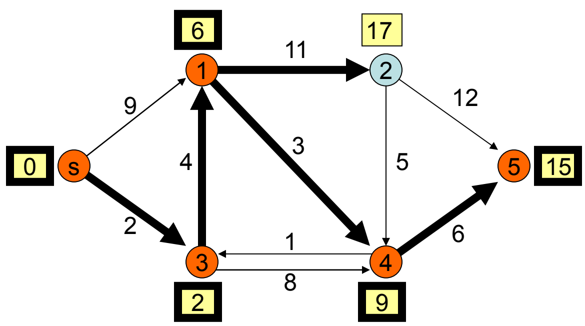
Dijkstra's Algorithm – Example



Examine node 4, which becomes permanently labeled.
Tighten all edges leaving node 4.

15

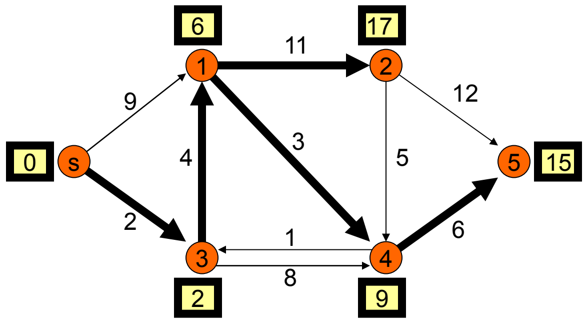
Dijkstra's Algorithm – Example



Examine node 5, which becomes permanently labeled.
Tighten all edges leaving node 5 (there aren't any).

16

Dijkstra's Algorithm – Example



Examine node 2, which becomes permanently labeled.
Algorithm terminates!

17

Dijkstra's Algorithm – Running Time

	Remove-Min O(n) times	**Decrease Key O(m) times**	**Total Runtime**
Unsorted Array	O(n)	O(1)	$O(n^2)$
Binary Heap	O(log n)	O(log n)	O(m log n)
Fibonacci Heap	O(log n)	O(1) amortized	O(m + n log n)

- The Fibonacci heap bound is always at least as good as both other bounds in any graph!
- In the RAM model (small integer edge costs), we can improve performance by using special RAM priority queues.

18

Dijkstra's Algorithm – Correctness

- Just like BFS, Dijkstra visits all nodes in increasing order of distance from the source, and visits each node exactly once.
- **Claim**: when we visit node i, c[i] will be the correct shortest path cost from s to i.
- Think of Dijkstra as maintaining two sets:

  S: visited nodes that are "permanently" labeled.

  T: remaining nodes that are temporarily labeled (whose cost labels are still upper bounds)
- Every iteration takes the node from T with minimum label and adds it to S.

The Bellman-Ford Algorithm

```
Bellman-Ford:
   c[s] = 0, c[j ≠ s] = +∞
   Repeat n-1 times:
      Tighten every edge in the graph
   If c[i] + cost(i, j) < c[j] for any edge (i, j)
      Output "negative cycle detected!"
```

- O(mn) running time.
- Detects negative cycles.
- Easy to analyze using induction: after k iterations of the main loop, each label c[j] reflects the cost of the shortest k-hop path from s to j.
- Can also interpret as a dynamic programming algorithm, or equivalently as our algorithm for finding shortest paths in a DAG with n "levels".

All-Pairs Shortest Path Algorithms

- Typically we think of this problem as taking an n x n matrix of edge costs as input and producing an n x n matrix of shortest path costs as output.
- Related problem: find the **diameter** of a graph.
- Can solve the problem using n invocations of a single-source algorithm.
  - Nonnegative edge costs: n x Dijkstra = $O(mn + n^2 \log n)$.
  - Negative edge costs: n x Bellman-Ford = $O(mn^2)$.

    (this can be improved to $O(mn + n^2 \log n)$ by using Johnson's algorithm: run Bellman-Ford once, then "reweight" the edges of the graph with nonnegative costs, and run Dijkstra n times…).

Johnson's Algorithm

- Add a dummy node s to the graph with a zero-cost edge to every other node.
- Use Bellman-Ford to find the shortest path cost c[i] from s to every other node i.
- These costs c[i] satisfy the triangle inequality:

  $c[i] + \text{cost}(i, j) \geq c[j]$ for every edge (i, j).
- Define the **reduced cost** of (i, j) as:

  $\text{cost}'(i, j) = \text{cost}(i, j) + c[i] - c[j] \geq 0$.
- Now find shortest paths from every other source node using Dijkstra's algorithm!
- Total running time: $O(mn + n^2 \log n)$.

Johnson's Algorithm

- Define the **reduced cost** of (i, j) as:
  cost'(i, j) = cost(i, j) + c[i] – c[j] ≥ 0.
- The reduced cost of an i → j path p is:
  cost'(p) = cost(p) + c[i] – c[j].
- So when we convert to reduced costs, this just offsets the cost of every i → j path by the same amount: c[i] – c[j]. The shortest i → j path remains the shortest, however!
- Reduced costs have many nice uses in shortest path calculations.

23

The Floyd-Warshall Algorithm

- Let c(i, j) denote the cost of an edge from node i to node j.
  - If there is no edge, set c(i, j) = ∞.
  - Also, set c(i, i) = 0.
- After running the remarkably simple $O(n^3)$ Floyd-Warshall algorithm…

```
Floyd-Warshall:
For k = 1 to n
  For i = 1 to n
    For j = 1 to n
      If c[i,k] + c[k,j] < c[i,j], c[i,j] = c[i,k] + c[k,j]
```

- … c(i, j) now gives us the cost of a shortest path from i to j!

24

The Floyd-Warshall Algorithm

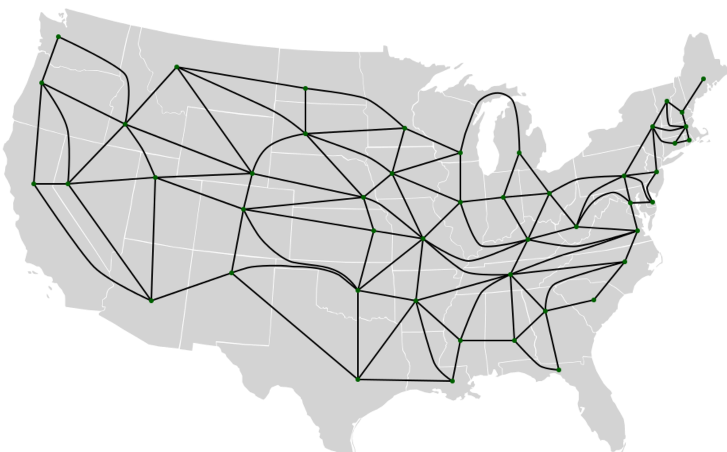
```
Floyd-Warshall:
For k = 1 to n
  For i = 1 to n
    For j = 1 to n
      If c[i,k] + c[k,j] < c[i,j], c[i,j] = c[i,k] + c[k,j]
```

- This works even if there are negative-cost edges. We can detect a negative cycle if c(i, i) < 0 for any node i after termination.
- Correctness follows from analysis as a dynamic programming algorithm:
  - Let $c^k[i, j]$ denote the cost of a shortest i → j path that uses only nodes 1 … k as potential intermediate nodes along the path.
  - Initially, $c^0[\mathbf{I}, j]$ = the cost of edge (i, j).
  - Moreover, $c^n[i, j]$ = the cost of a shortest i → j path.
  - DP recursion: $c^k[i, j] = \min\{ c^{k-1}[i, j],\ c^{k-1}[i, k] + c^{k-1}[k, j] \}$

25

Recall: Centrality

- What nodes are most "central" in a graph?



26

Centrality & Shortest Paths

- Many notions of centrality are based on shortest paths:
  - Eccentricity(x) = max dist(x,y) over other nodes y; a graph "center" is a node with minimum eccentricity.
  - "Betweenness" centrality of an edge or node: fraction of all point-to-point shortest paths passing through that edge/node.
- Other prominent notions of centrality include:
  - Degree centrality: high degree treated as "more central"
  - Eigenvector centrality (centrality proportional to probability a node is visited by a random walk in steady state); used by Google Pagerank.

Formulating Shortest Path Problems: Change the Problem, not the Algorithm…

- In an automobile transportation network, suppose right turns can be made instantly, while left turns take 1 extra unit of time…

Formulating Shortest Path Problems: Change the Problem, not the Algorithm…

- In an automobile transportation network, suppose right turns can be made instantly, while left turns take 1 extra unit of time…
- Suppose we have N jobs to complete. Each job has a duration and possibly a set of prerequisite jobs. Working in parallel, how quickly can we finish all the jobs?

29

Formulating Shortest Path Problems: Change the Problem, not the Algorithm…

- In an automobile transportation network, suppose right turns can be made instantly, while left turns take 1 extra unit of time…
- Suppose we have N jobs to complete. Each job has a duration and possibly a set of prerequisite jobs. Working in parallel, how quickly can we finish all the jobs?
- Suppose some roads charge a $1 toll. How do we find a shortest path if we can pay at most $3 worth of tolls?

30

Formulating Shortest Path Problems: Change the Problem, not the Algorithm…

- In an automobile transportation network, suppose right turns can be made instantly, while left turns take 1 extra unit of time…
- Suppose we have N jobs to complete. Each job has a duration and possibly a set of prerequisite jobs. Working in parallel, how quickly can we finish all the jobs?
- Suppose some roads charge a $1 toll. How do we find a shortest path if we can pay at most $3 worth of tolls?
- What is the second-shortest s-t path?