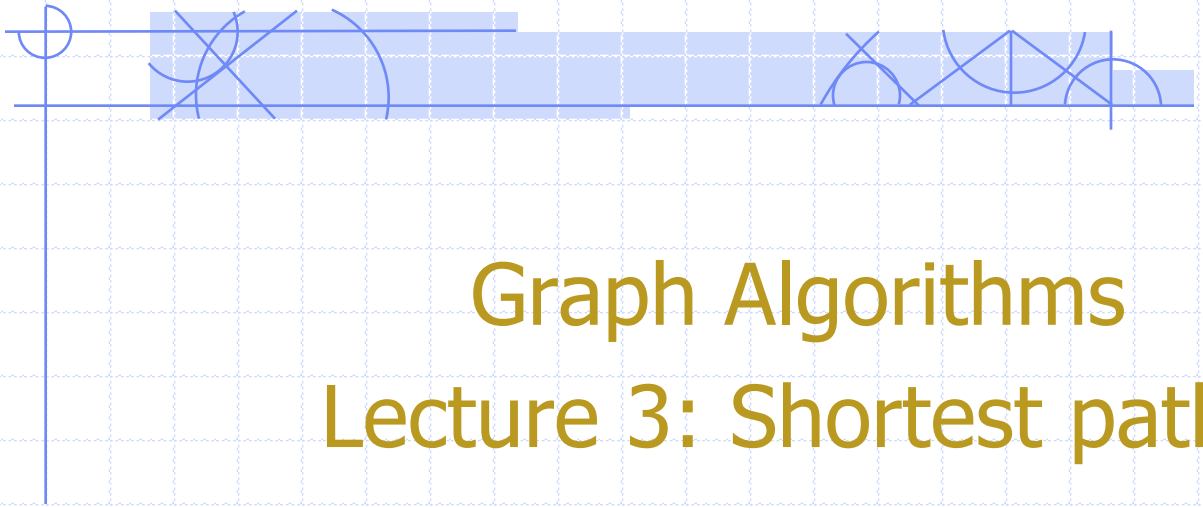
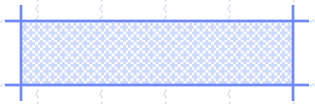
COMP26120: Algorithms and Imperative Programming



Graph Algorithms

Lecture 3: Shortest paths



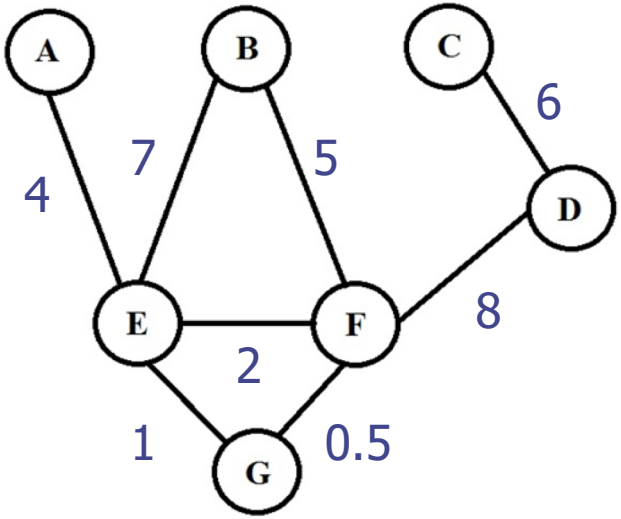
Lecture outline

- Single source shortest paths in a weighted graph;
- Dijkstra's algorithm;
- The Bellman-Ford algorithm;
- Shortest path in directed acyclic graphs;
- All-pairs shortest paths;

Single source shortest paths

- Many systems in real world can be represented as weighted graphs (road/railway network, computer network, social networks, complex systems).
- In general, a weighted graph is a graph that has a numeric label $w(e)$ associated with each edge e.
- Edge weights represent a concept such as distance, connection costs, or affinity.

Single source shortest paths



Single source shortest paths

- Let G be a weighted graph. The length $w(p)$ of a path $p = \{e_0, e_1, \dots, e_n\}$ is defined as

$$w(p) = \sum_{k=0}^{n} w(e_i)$$

- We need to ensure that all the weights in the graph are non-negative, so that no negative cycles in the graph exist.
- The problem: For a given weighted graph G find the shortest paths from a selected node v to all other nodes.
- For the case with no negative edge weights these exists a greedy strategy referred to as Dijkstra's algorithm.
- Dijkstra's algorithm is performing a weighted BFS starting at the node v.

Dijkstra's algorithm

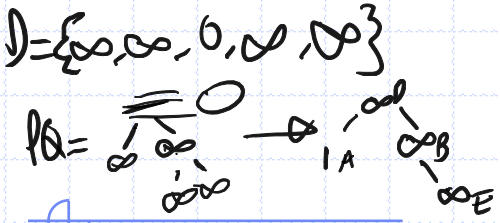
- The greedy method iteratively grows a "cloud" of vertices out of v , with the vertices entering the cloud in based on increasing distances from v.
- At each iteration, the next chosen vertex the one outside the cloud that is closest to v.
- The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from v to every other vertex of G.
- Our presentation will be for a graph that is undirected and has no self loops or parallel edges.
- We use the approximation to the true distance between the two nodes which is iteratively improved until its true value is obtained.

Dijkstra's algorithm

- Greedy procedure deploys edge relaxation, where for each node $u \neq v$ a label $\mathcal{D}[u]$ is assigned with an initial value $+\infty$. During the course of the algorithm, this label holds current approximation of the shortest path length between u and v.
- We define a cloud C of visited nodes to be initially an empty set.
- At each iteration we pull out a node with the smallest $\mathcal{D}[u]$ and update those $\mathcal{D}[z]$ for the nodes z that are adjacent to u. This update should reflect the fact that there is a better way to get to z via the node u than previously determined.

if $\mathcal{D}[u] + w(u,z) < \mathcal{D}[z]$ **then**

$\mathcal{D}[u] + w(u,z) \rightarrow \mathcal{D}[z]$



Dijkstra's algorithm

Algorithm DijkstraShortestPaths(G, v);
Input: A simple undirected weighted graph G with nonnegative edge weights, and a distinguished vertex v of G;
Output: Labels $\mathcal{D}[u]$ for all $u \in G$, representing the distances from v to u;
$\mathcal{D}[v] = 0$; $\mathcal{D}[u] = +\infty$ for all $u \neq v$ in G;
Set a priority queue Q to contain all vertices $u \in G$ using $\mathcal{D}[u]$ as keys;
while (Q is not empty) **do**
  $u \leftarrow Q$.removeMin(); // *pull a new vertex into the cloud*
  for (each vertex z adjacent to u && $z \in Q$) **do**
    if $\mathcal{D}[u] + w(u,z) < \mathcal{D}[z]$ **then**
      $\mathcal{D}[u] + w(u,z) \rightarrow \mathcal{D}[z]$; // *perform the relaxation on edge* (u,z)
    end if
  end for
end while
return the labels $\mathcal{D}[u]$ of each vertex u

Dijkstra's algorithm

- Complexity analysis: Denote by n and e the number of nodes and the number of edges in a graph G, respectively.
- The assumption is that edge weight additions and comparisons are the basic operations that take $O(1)$ time.
- We also assume that the graph G is represented by an adjacency list. The cost of sweeping the vertices This data structure allows us to step through the vertices adjacent to u during the relaxation step is proportional to their number.
- How is the priority queue Q implemented? We can implement a PQ either as a sorted list or as a heap. In the latter case the cost of performing the method Q.removeMin() is $O(\log(n))$ -- recall up/down heap bubbling.

Dijkstra's algorithm

- The cost of inserting n vertices in Q with their initial key value is $O(n \cdot \log(n))$.
- At each of the n iterations of the while loop we spend $O(\log(n)$ time performing Q.removeMin() and $O(\deg(u) \cdot \log(n))$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the while loop is

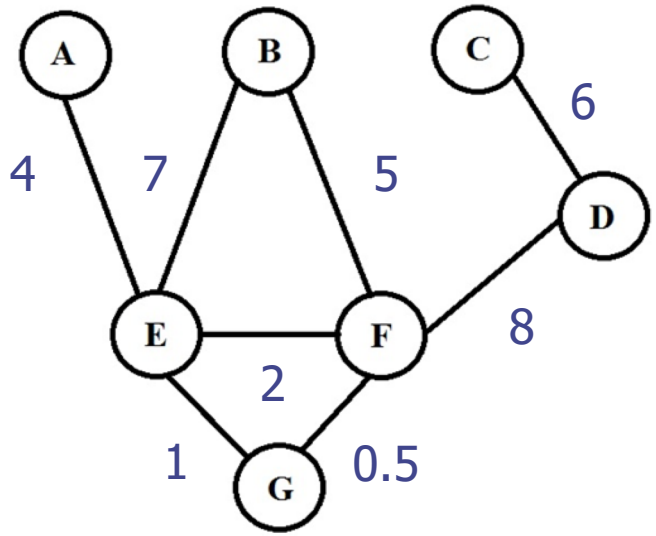
$$\sum_{u \in G} (1 + \deg(u)) \cdot \log(n) = O((n + e) \cdot \log(n))$$

since $\sum_{u \in G} \deg(u) = 2e$.

- If the PQ is implemented as an unsorted doubly linked list. Such implementation would result in $O(n^2)$ complexity (Homework: work out the details in Goodrich & Tamasia, p.406).

Dijkstra's algorithm

A	B	C	D	E	F	G
0	+∞	+∞	+∞	+∞	+∞	+∞



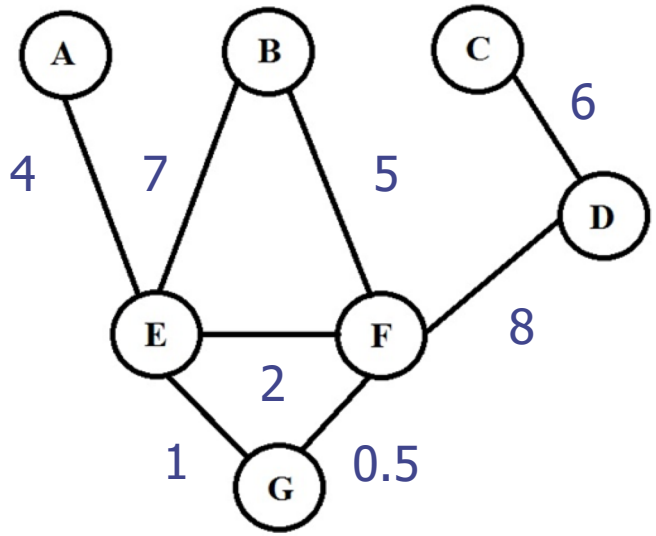
Step 1: Remove A from PQ

B	C	D	E	F	G
+∞	+∞	+∞	**4**	+∞	+∞

Step 2: Remove E from PQ

B	C	D	F	G
11	+∞	+∞	**6**	**5**

Dijkstra's algorithm



$\mathcal{D} = \{0, 10\frac{1}{2}, 19\frac{1}{2}, 13\frac{1}{2}, 4, 5\frac{1}{2}, 5\}$

Step 3: Remove G from PQ

B	C	D	F
11	$+\infty$	$+\infty$	**5.5**

Step 4: Remove F from PQ

B	C	D
10.5	$+\infty$	**13.5**

Step 5: Remove B from PQ

C	D
$+\infty$	**13.5**

Step 6: Remove D from PQ

C
19.5

The Bellman-Ford algorithm

- This algorithm finds the shortest paths in directed graphs which have negative edge weights.
- The Bellman-Ford algorithm shares the edge relaxation with Dijkstra's algorithm, but does not use it in conjunction with the greedy method. Instead, it performs the relaxation of each edge in a digraph $n-1$ times, where n is the number of nodes.
- The shortest paths between the nodes are calculated in a bottom-up manner, where after each iteration $k = 1, \ldots, n-1)$ we have the shortest paths of the length at most k between all pairs of nodes.

The Bellman-Ford algorithm

Algorithm BellmanFordShortestPaths (G, v);
Input: A weighted directed graph G with n vertices, and a distinguished vertex $v \in G$;
Output: Labels $\mathcal{D}[u]$ for all $u \in G$, representing the distances from v to u or an indication that G has a negative-weight cycle;

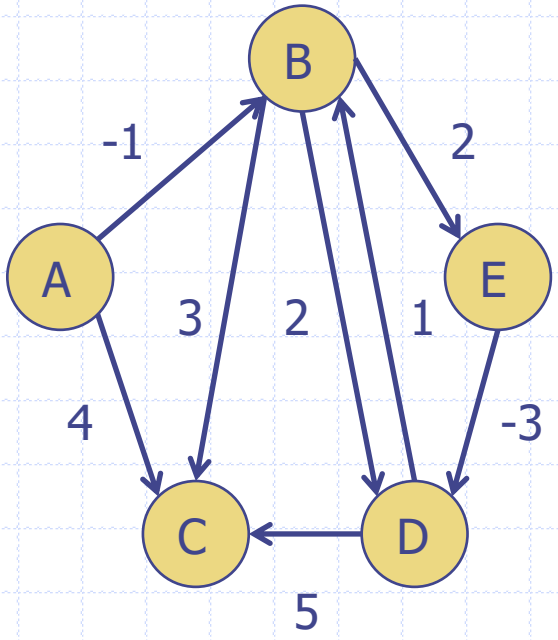
$\mathcal{D}[v] = 0$; $\mathcal{D}[u] = +\infty$ for all $u \neq v$ in G;
for $i = 1 : n - 1$ **do**
  for each directed edge (u, z) outgoing from u **do**
    if $\mathcal{D}[u] + w(u, z) < \mathcal{D}[z]$ **then**
      $\mathcal{D}[u] + w(u, z) \rightarrow \mathcal{D}[z]$; // *perform the relaxation on edge* (u, z)
    end if
  end for
end for
if for any pair of nodes (u, v) we have $\mathcal{D}[v] > \mathcal{D}[u] + w(u, v)$ **then**
  return G contains a negative-weight cycle;
else
  return $\mathcal{D}[u]$ for all $u \in G$;
end if

$\{0, \infty, \infty, \infty\}$ edges: $\{(D,C), (B,C), (D,B), (A,B), (A,D)\}$

$0+(2+1)<\infty \rightarrow \checkmark \ldots$ (D,C)
$0+(1+2)<\infty \rightarrow \checkmark \ldots$ (B,C)
$0+(2-2)<\infty \checkmark$ (D,B)
$0+1<0$ (A,B)
$0+2<\infty$ (A,D)

A	B	C	D
0	∞	∞	∞
		3	
		3	
	0		
			2

The Bellman-Ford algorithm



$n = 5$, the edges need to be processed 4 times

A	B	C	D	E
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$

We process the edges in the following order:
(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)

Iteration 1:

After processing (B,E), (D,B), (B,D), (A,B)

A	B	C	D	E
0	-1	$+\infty$	$+\infty$	$+\infty$

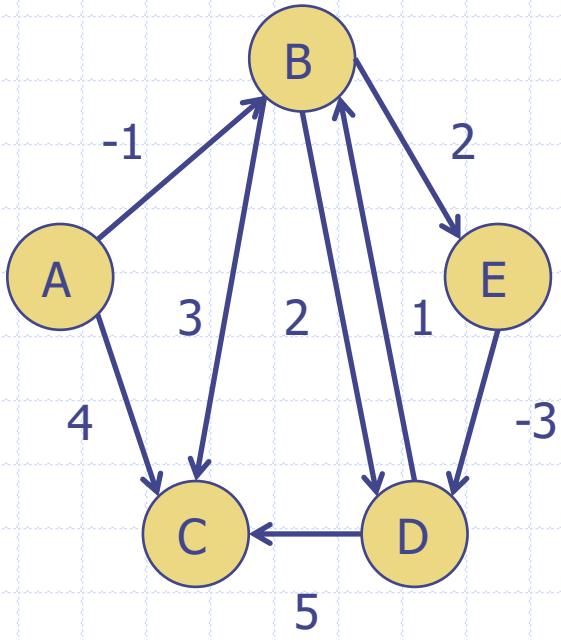
After processing (A,C)

A	B	C	D	E
0	-1	4	$+\infty$	$+\infty$

After processing (D,C), (B,C), (E,D)

A	B	C	D	E
0	-1	2	$+\infty$	$+\infty$

The Bellman-Ford algorithm



$n = 5$, the edges need to be processed 4 times

Iteration 2:

After processing (B,E), (D,B), (B,D), (A,B)

A	B	C	D	E
0	−1	2	1	1

After processing (A,C), (D,C), (B,C), (E,D)

A	B	C	D	E
0	−1	2	−2	1

Iterations 3 and 4 do not produce any changes (check)

Homework: Change the weight -3 in the graph into -4 and rerun the algorithm.

The Bellman-Ford algorithm

- The complexity of the Bellman-Ford algorithm:
  - The main loop is performed $n-1$ times.
  - During the loop all e edges are visited.
  - Each edge is processed in $O(1)$ time.
- The complexity time for this algorithm is $O(n \cdot e)$.