Data Structures and Algorithms

Spring 2009-2010

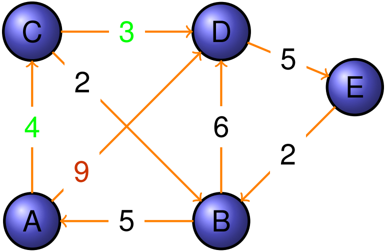
Outline

Outline

1. Graph Algorithms
   - Weighted Shortest-Path Algorithms

Shortest Path on Weighted Graphs

- Problem is similar to unweighted case but now we may discover a cheaper path to a node than we had already
- When a cheaper path to a node is discovered need to update cost of getting to the node and path that gets you there
- This happens when distances do not obey the *triangle inequality*: compare $A - C - D$ with $A - D$

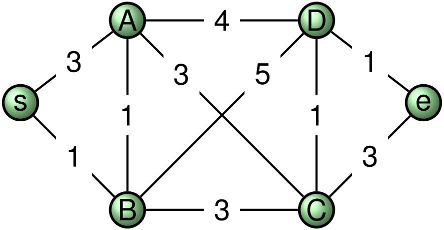


Dijkstra's Algorithm

- The problem with an identical approach to BFS is that we may encounter a **cheaper** path to a vertex later in the algorithm
- *Dijkstra's Algorithm* replaces the queue data structure with a priority queue
- At each step, instead of picking out the first vertex in the queue to explore next we pick out the **nearest** one (`del_min` operation)
- The key argument of the algorithm is that the vertex removed from the priority queue cannot be reached by a shorter path

Dijkstra's Algorithm (contd.)

What is the shortest distance from *s* to *e*?



Dijkstra's Algorithm (contd.)

s	A	B	C	D	e	Explore
0	∞	∞	∞	∞	∞	s
	$3(s)$	$1(s)$	∞	∞	∞	B
	$2(B)$		$4(B)$	$6(B)$	∞	A
			$4(B)$	$6(B)$	∞	C
				$5(C)$	$7(C)$	D
					$6(D)$	e

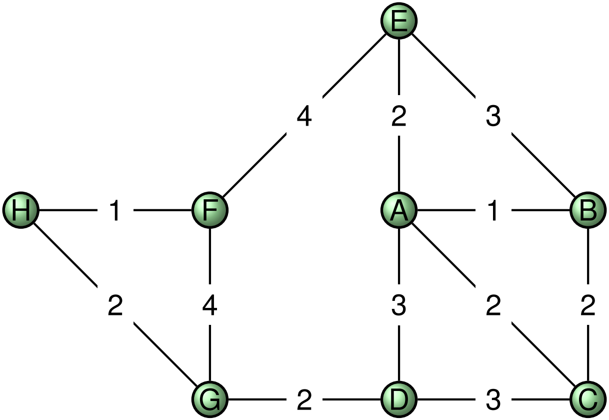
Notation: an entry of "5(C)" in column D means there is a path of cost 5 to D with C as immediate predecessor; read downwards, a node's column changes from a) ∞ to b) better and better to, finally, c) best cost to it.

Row entries indicate the shortest known distance **to date**; when it is smallest in row it is **explored**.

So, working backwards, best path is *sBCDe*

Dijkstra's Algorithm (contd.)

What is the shortest distance from *H* to *B*?



Dijkstra's Algorithm (contd.)

H	F	G	E	D	A	C	B	Explor
0	∞	∞	∞	∞	∞	∞	∞	H
	$1(H)$	$2(H)$	∞	∞	∞	∞	∞	F
		$2(H)$	$5(F)$	∞	∞	∞	∞	G
			$5(F)$	$4(G)$	∞	∞	∞	D
			$5(F)$		$7(D)$	$7(D)$	∞	E
					$7(D)$	$7(D)$	$8(E)$	A
						$7(D)$	$8(E)$	C
							$8(E)$	B

At each iteration choose as node to explore next the smallest-value node in the row.
A node is **active** (in priority queue) when it is not ∞ (it has been discovered) and it has not yet turned black (not yet been removed from priority queue)

Dijkstra's Algorithm (contd.)

- Code for Dijkstra's algorithm is here
- Dijkstra's algorithm is an example of *greedy* algorithm
- A greedy algorithm is one which the item with the most "stuff" is taken on each iteration
- We have not discussed the operation `decrease_p()` on a priority queue; nonetheless, it can be shown that a node's cost can be adjusted in time $O(\log|V|)$
- The `while(! PQ.empty())` loop executes exactly $O(|V|)$ times, taking a new node from the PQ each time
- These $|V|$ `delete_min`s take time $O(|V|\log|V|)$
- Either a PQ `insert()` or `decrease_p` (each taking $O(\log|V|)$) is done for each node that is connected to the node extracted from `PQ` iteration: $O(|E|\log|V|)$

Dijkstra's Algorithm (contd.)

- Using a priority queue, the running time of algorithm is $O(|E|\log|V| + |V|\log|V|) = O(|E|\log|V|)$ since $|E| \geq |V|$ almost always
- Within the `forall_adj_edges` loop it appears that a node that has been selected by an earlier `del_min` could be reinserted in `PQ`
- The proof of correctness of the algorithm prevents this happening
- Correctness: exercise (either look it up in book or think about what it means for a node to be selected by the `del_min()` operation