COMP 761: Lecture 24 – Graph Algorithms I

David Rolnick

October 30, 2020

Problem

Find a way to remove the maximum from a heap of n elements, while preserving the heap property, in $O(\log n)$ time.

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- For Problem 1 on the Problem Set, please do not just cite a result about n^k and $O(\cdot)$ or $\Omega(\cdot)$, would like at least an explanation for why the definitions of $O(\cdot)$ and $\Omega(\cdot)$ hold here.

Review: Heaps

Review: Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.

Review: Heaps

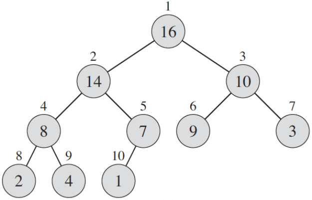
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.

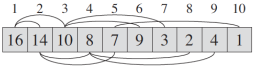
Review: Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.

Review: Heaps

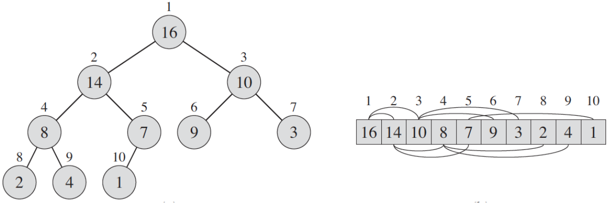
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.





Review: Heaps

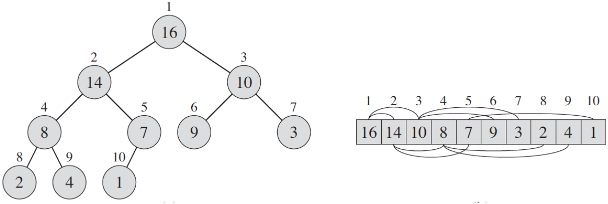
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.



- We also assume in a heap that each *row* (set of nodes of a single depth) is full except possibly the last one, so depth = $\Theta(\log n)$.

Review: Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.



- We also assume in a heap that each *row* (set of nodes of a single depth) is full except possibly the last one, so depth = $\Theta(\log n)$.
- And that the last row has all its nodes as far left as possible (easy by swapping left and right).

Review: Priority queues

Review: Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).

Review: Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.

Review: Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.

Review: Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.
- A max-heap can be used to implement a max-priority queue!

Review: Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.
- A max-heap can be used to implement a max-priority queue!
- (Likewise, a min-heap can be used to implement a min-priority queue.)

Heaps as priority queues

Operations we need: Insert, Maximum, ExtractMax, IncreaseKey.

Heaps as priority queues

Operations we need: Insert, Maximum, ExtractMax, IncreaseKey.

- Do we have any of these already in a max-heap?

Heaps as priority queues

Operations we need: Insert, Maximum, ExtractMax, IncreaseKey.

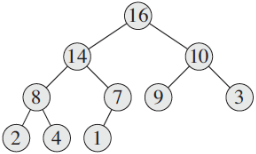
- Do we have any of these already in a max-heap?
- We already have Maximum, since it's the root.

Heaps as priority queues

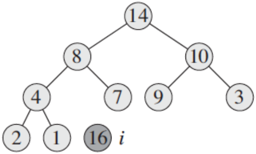
Operations we need: Insert, Maximum, ExtractMax, IncreaseKey.

- Do we have any of these already in a max-heap?
- We already have Maximum, since it's the root.
- Let's now try to do ExtractMax.

ExtractMax

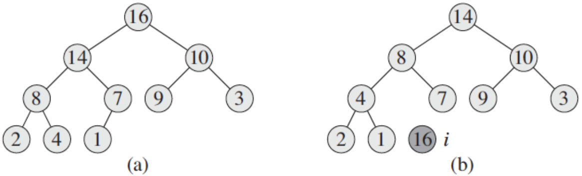


(a)



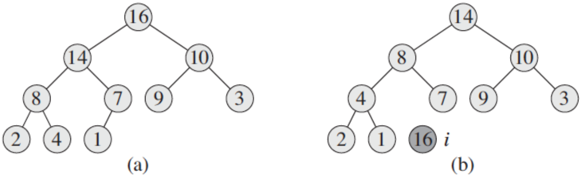
(b)

ExtractMax



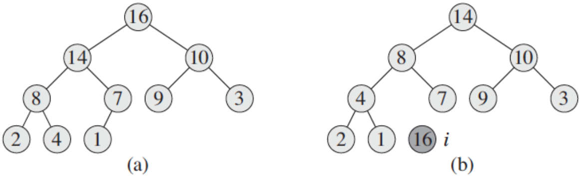
- Let's take out the root and replace it with the last element of the last row.

ExtractMax



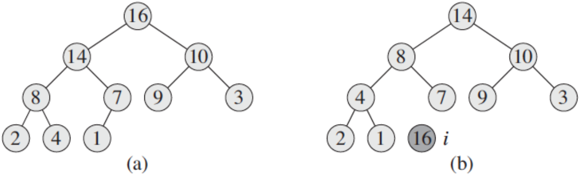
- Let's take out the root and replace it with the last element of the last row.
- Of course, this is no longer a heap.

ExtractMax



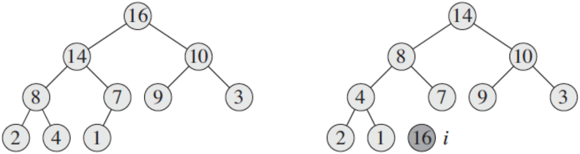
- Let's take out the root and replace it with the last element of the last row.
- Of course, this is no longer a heap.
- But the two sub-trees below the root must both be heaps.

ExtractMax



- Let's take out the root and replace it with the last element of the last row.
- Of course, this is no longer a heap.
- But the two sub-trees below the root must both be heaps.
- So we can do the iterative swapping thing we did before to make a heap.

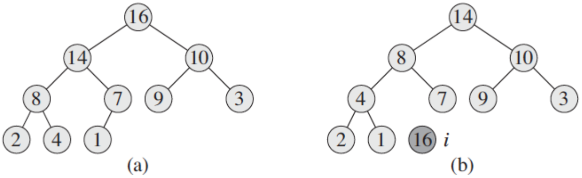
ExtractMax



(a) (b)

- Let's take out the root and replace it with the last element of the last row.
- Of course, this is no longer a heap.
- But the two sub-trees below the root must both be heaps.
- So we can do the iterative swapping thing we did before to make a heap.
- # swaps = depth of whole tree = $O(\log n)$.

ExtractMax



- Let's take out the root and replace it with the last element of the last row.
- Of course, this is no longer a heap.
- But the two sub-trees below the root must both be heaps.
- So we can do the iterative swapping thing we did before to make a heap.
- $\#$ swaps = depth of whole tree = $O(\log n)$.
- So ExtractMax runs in $O(\log n)$.

IncreaseKey

IncreaseKey

- Now we've done Maximum and ExtractMax.

IncreaseKey

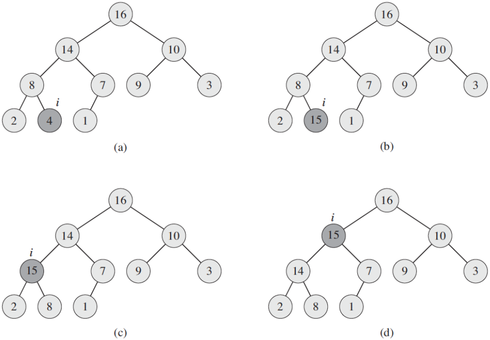
- Now we've done Maximum and ExtractMax.
- Let's do IncreaseKey.

IncreaseKey

- Now we've done Maximum and ExtractMax.
- Let's do IncreaseKey.
- The process is like the reverse of what we just did.

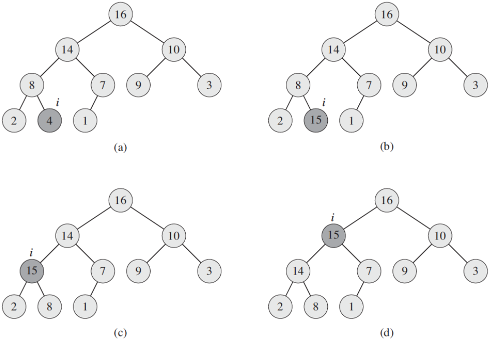
IncreaseKey

- Now we've done Maximum and ExtractMax.
- Let's do IncreaseKey.
- The process is like the reverse of what we just did.
- Swap the key up until it is less than its parent:



IncreaseKey

- Now we've done Maximum and ExtractMax.
- Let's do IncreaseKey.
- The process is like the reverse of what we just did.
- Swap the key up until it is less than its parent:



- This takes $O(\log n)$ moves.

Insert

Insert

- We can use our previous operations to do Insert.

Insert

- We can use our previous operations to do Insert.
- Let's add something really small, $-\infty$, onto the end of the heap.

Insert

- We can use our previous operations to do Insert.
- Let's add something really small, $-\infty$, onto the end of the heap.
- It clearly is still a heap because the new key is smaller than its parent.

Insert

- We can use our previous operations to do Insert.
- Let's add something really small, $-\infty$, onto the end of the heap.
- It clearly is still a heap because the new key is smaller than its parent.
- Now we can run IncreaseKey to make it whatever value we want.

Insert

- We can use our previous operations to do Insert.
- Let's add something really small, $-\infty$, onto the end of the heap.
- It clearly is still a heap because the new key is smaller than its parent.
- Now we can run IncreaseKey to make it whatever value we want.
- The time is just the time for IncreaseKey, $O(\log n)$.

Putting it all together

Putting it all together

- We have a way to use a max-heap as a max-priority queue.

Putting it all together

- We have a way to use a max-heap as a max-priority queue.
- Each of the operations Insert, Maximum, ExtractMax, and IncreaseKey runs in $O(\log n)$ time (and Maximum is just $O(1)$ time).

Putting it all together

- We have a way to use a max-heap as a max-priority queue.
- Each of the operations Insert, Maximum, ExtractMax, and IncreaseKey runs in $O(\log n)$ time (and Maximum is just $O(1)$ time).
- We'll see soon why this is useful.

Heapsort

Heapsort

- We get a new sorting algorithm for free!

Heapsort

- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.

Heapsort

- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.
- Extract the maximum.

Heapsort

- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.
- Extract the maximum.
- Do it again and again for the whole heap.

Heapsort

- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.
- Extract the maximum.
- Do it again and again for the whole heap.
- How long does it take (worst-case)?

Heapsort

- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.
- Extract the maximum.
- Do it again and again for the whole heap.
- How long does it take (worst-case)?
- Each ExtractMax call takes time $O(\log n)$.

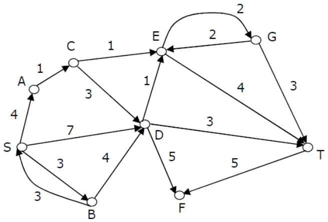
Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Heapsort

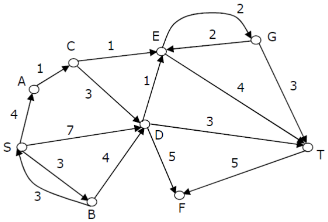
- We get a new sorting algorithm for free!
- Make the list into a heap - time $O(n)$.
- Extract the maximum.
- Do it again and again for the whole heap.
- How long does it take (worst-case)?
- Each ExtractMax call takes time $O(\log n)$.
- Total (worst-case) time $n \cdot O(\log n) = O(n \log n)$.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Review: Weighted, directed graphs

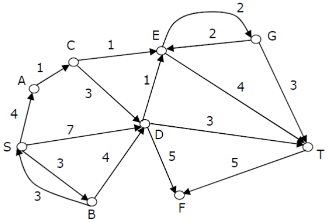


Review: Weighted, directed graphs



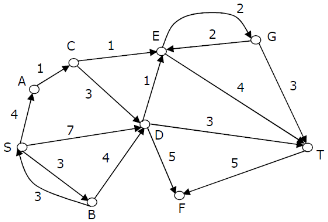
- In a directed graph, (i, j) is not the same as (j, i).

Review: Weighted, directed graphs



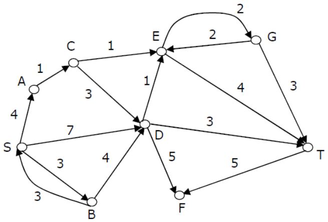
- In a directed graph, (i, j) is not the same as (j, i).
- Can have either one, or both edges, or neither.

Review: Weighted, directed graphs



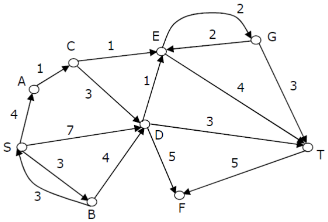
- In a directed graph, (i, j) is not the same as (j, i).
- Can have either one, or both edges, or neither.
- Weighted graphs have a weight on every edge.

Review: Weighted, directed graphs

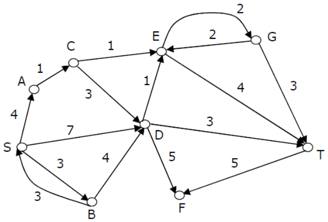


- In a directed graph, (i, j) is not the same as (j, i).
- Can have either one, or both edges, or neither.
- Weighted graphs have a weight on every edge.
- If (i, j) and (j, i) both exist, then can have different weights on them.

Shortest paths

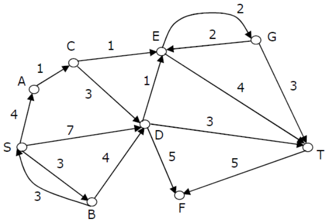


Shortest paths



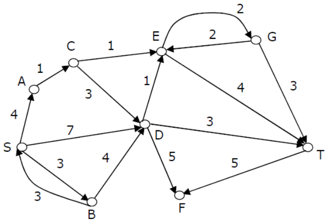
- A *path* between two vertices s and t is a sequence of directed edges from s to t.

Shortest paths



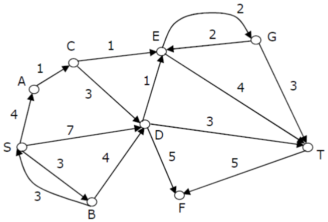
- A *path* between two vertices s and t is a sequence of directed edges from s to t.
- The *length* of a path is the sum of weights along the edges.

Shortest paths



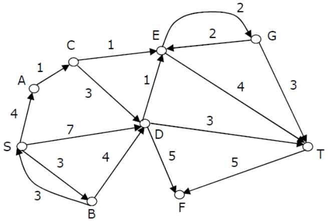
- A *path* between two vertices *s* and *t* is a sequence of directed edges from *s* to *t*.
- The *length* of a path is the sum of weights along the edges.
- A *shortest path* between *s* and *t* is a path with minimal length between them. (There might be several such paths).

Shortest paths

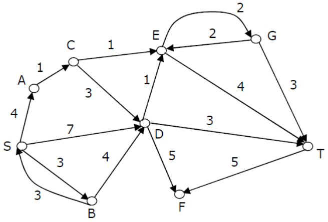


- A *path* between two vertices *s* and *t* is a sequence of directed edges from *s* to *t*.
- The *length* of a path is the sum of weights along the edges.
- A *shortest path* between *s* and *t* is a path with minimal length between them. (There might be several such paths).
- We can also do this with an unweighted graph (all weights = 1) or an undirected graph (edges go both ways).

Dijkstra's algorithm

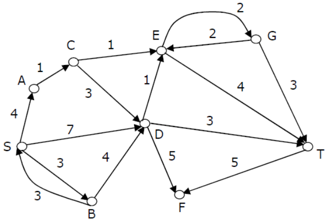


Dijkstra's algorithm



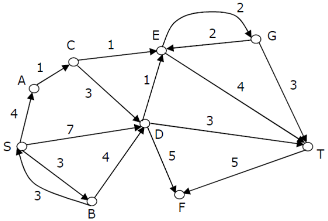
- Problem: Connected graph G (weighted and directed) with weights ≥ 0. Given a vertex s, find the lengths of shortest paths from s to all other vertices.

Dijkstra's algorithm



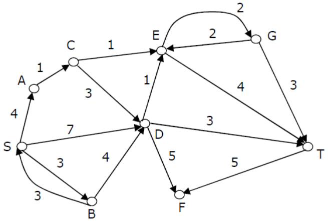
- Problem: Connected graph G (weighted and directed) with weights ≥ 0. Given a vertex s, find the lengths of shortest paths from s to all other vertices.
- Is there any vertex that's easy?

Dijkstra's algorithm



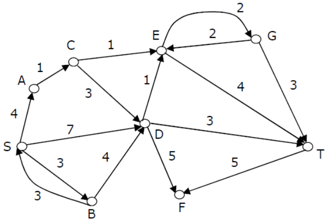
- Problem: Connected graph G (weighted and directed) with weights ≥ 0. Given a vertex s, find the lengths of shortest paths from s to all other vertices.
- Is there any vertex that's easy?
- The vertex b that is closest to s must have distance = w_{sb}.

Dijkstra's algorithm



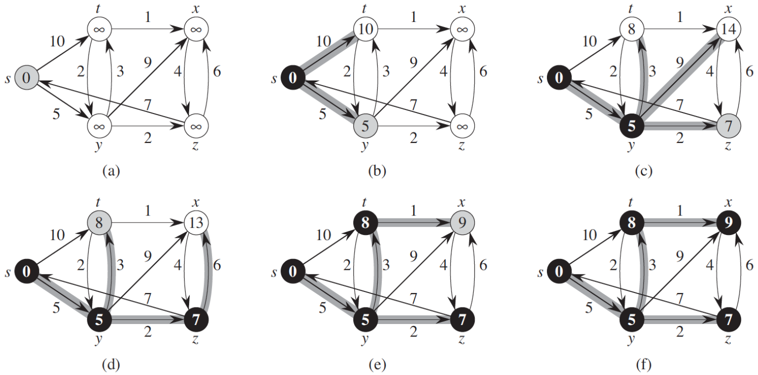
- Problem: Connected graph G (weighted and directed) with weights ≥ 0. Given a vertex s, find the lengths of shortest paths from s to all other vertices.
- Is there any vertex that's easy?
- The vertex b that is closest to s must have distance = w_{sb}.
- Is there another vertex that is easy?

Dijkstra's algorithm

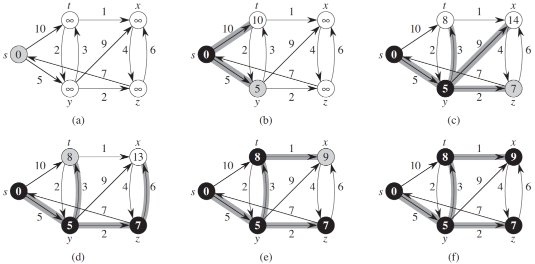


- Problem: Connected graph G (weighted and directed) with weights ≥ 0. Given a vertex s, find the lengths of shortest paths from s to all other vertices.
- Is there any vertex that's easy?
- The vertex b that is closest to s must have distance = w_{sb}.
- Is there another vertex that is easy?
- Similar logic, vertex $a = \text{argmin}_q(\min(w_{sq}, w_{sb} + w_{bq}))$.

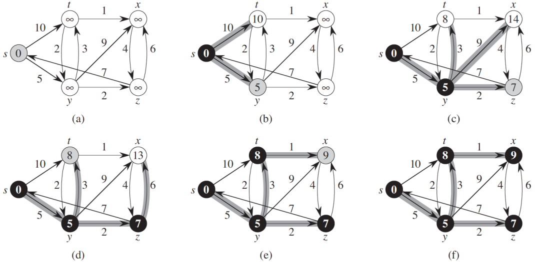
Dijkstra's algorithm



Dijkstra’s algorithm

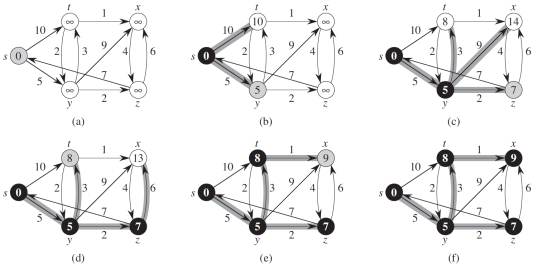


Dijkstra's algorithm



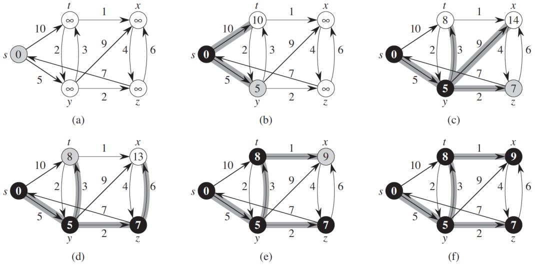
- Maintain guesses for distances of all vertices from s.

Dijkstra's algorithm



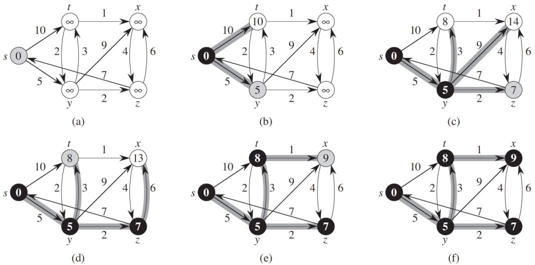
- Maintain guesses for distances of all vertices from *s*.
- Start out with ∞ everywhere, *s* at distance 0 from itself.

Dijkstra's algorithm



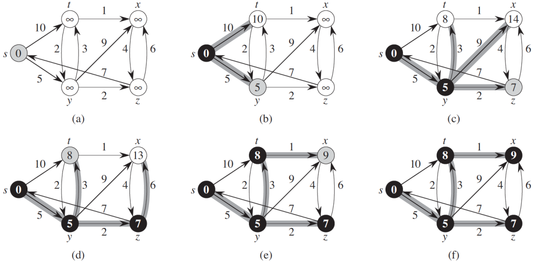
- Maintain guesses for distances of all vertices from *s*.
- Start out with ∞ everywhere, *s* at distance 0 from itself.
- At each step, pick unvisited vertex *i* with smallest distance estimate (starting with *s*) - this estimate has to be correct.

Dijkstra's algorithm



- Maintain guesses for distances of all vertices from s.
- Start out with ∞ everywhere, s at distance 0 from itself.
- At each step, pick unvisited vertex i with smallest distance estimate (starting with s) - this estimate has to be correct.
- Visit all its neighbors j, and update distance estimate $d(j)$ to $\min(d(j), d(i) + w_{ij})$.

Dijkstra's algorithm



- Maintain guesses for distances of all vertices from s.
- Start out with ∞ everywhere, s at distance 0 from itself.
- At each step, pick unvisited vertex i with smallest distance estimate (starting with s) - this estimate has to be correct.
- Visit all its neighbors j, and update distance estimate $d(j)$ to $\min(d(j), d(i) + w_{ij})$.
- Repeat.

Next time!

Graph algorithms II