CS430 Lecture 18 Activities - KEY

<u>Opening Questions</u>

1. How do you think the allocated size growth of a dynamic array like Java's ArrayList is implemented? How much bigger does it grow when needed? What is the runtime for a sequence of n insertions starting from a default size of 10 considering the worst individual insert?

Whenever the array overflows, "grow" it by allocating a new, larger table twice the size. Move all items from the old table into the new one, and free the storage for the old table.
Consider a sequence of n insertions. The worst-case time to execute one insertion is Θ(n). Therefore, the worst-case time for n insertions is n · Θ(n) = Θ(n^2).

<u>Amortized (to pay off gradually) Analysis</u>

So far, we have analyzed best and worst case running times for an operation without considering its context. With amortized analysis, we study a sequence of operations rather than individual operations. An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

<u>Aggregate Method of Amortized Analysis</u>

1. Can we do a better analysis by amortizing the cost over all inserts? Starting with a table size one and doubling the size when necessary make a table showing the first 10 inserts and determine a formula for cost(i) for the cost of the ith insert. Then aggregate *add up" all the costs and divide by n (aggregate analysis).

cost(i) = i   if (i-1) is an exact power of 2
1 otherwise

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\le 3n \\ &= \Theta(n).\end{aligned}$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

<u>Accounting Method of Amortized Analysis</u>

Figure out a specific amortized cost to be allocated to each operation to ensure you have enough "balance" to handle the bad operations.

Charge i th operation a fictitious amortized cost ĉi, where $1 pays for 1 unit of work (i.e., time).

- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the bank for use by subsequent operations.
- The bank balance must not go negative! We must ensure that for all n

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

Thus, the total amortized costs provide an upper bound on the total true costs.

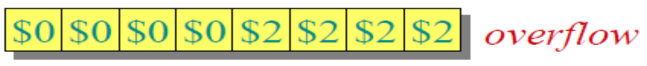
2. For the previous ArrayList example determine the amortized cost ĉi necessary.

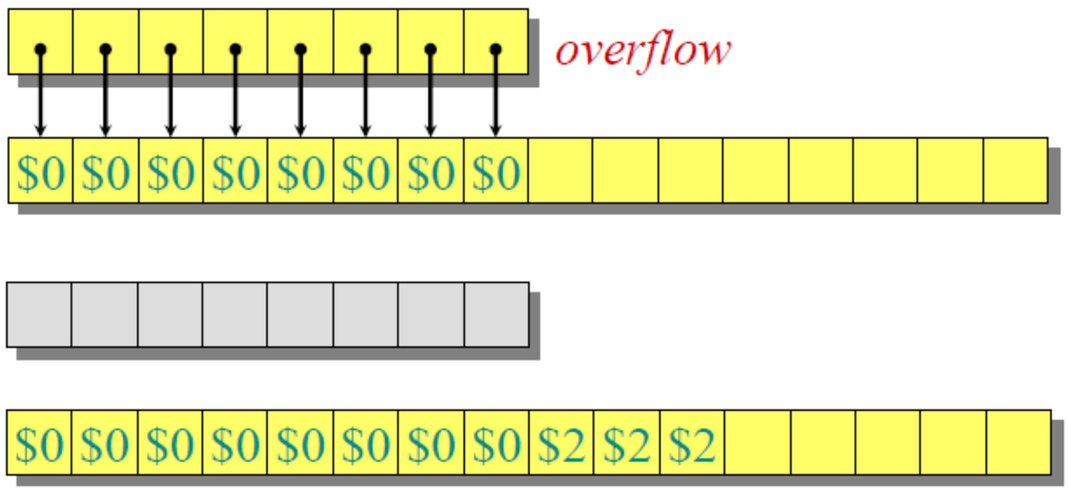
Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:





i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

*Okay, so I lied. The first operation costs only $2, not $3.

3. Consider, as a second example, a binary counter that is being implemented in hardware. Assume that the machine on which it is being run can flip a bit as its basic operation. We now want to analyze the cost of counting up from 0 to n (using k bits).

What is the naive worst-case analysis for how many bits we need to flip?
A naive worst-case analysis would argue that each time we increment the counter, we can, at worst, flip each of the k bits giving Theta(kn) total bit flips for counting up to n.

Decimal	Binary
1	000001
2	000010
3	000011
4	000100
5	000101
⋮	⋮
n	100110

4. Use the aggregate method to perform a more careful analysis for n increments of a binary counter.
Every time we increment the counter, the ones bit flips = n flips in total
Every second time we increment the counter, the twos bit flips = n/2 flips total
Every fourth time we increment the counter, the fours bit flips = n/4 flips total

Thus, altogether, the number of bit flips needed is:
n + n/2 + n/4 + . . . <= 2n
This gives an amortized time of 2n/n = 2 bit flips per increment.

5. Use the accounting method to perform a more careful analysis for n increments of a binary counter.
We will charge customers $2 for incrementing the counter. Since each increment converts a bit from zero to one, you can think of this as $2 to flip a bit from 0 to 1 and $0 to flip a bit from 1 to 0. Again, when a customer pays $2 for an increment, you would pay your employee $1 for the bit that flips from 0 to 1. The other dollar is stapled to this bit. In order to pay for flipping the bits which change from 1 to 0, the student takes the dollar stapled to those bits.

CS430 Lecture 19 Activities - KEY

Opening Questions

1. Review the operations on a min (or max) binary heap and their runtimes
Insert and extractMin on a binary heap are O(lg n). buildHeap is O(n).

2. What if we needed a function to union two min binary heaps? How would you do it and what is the run time?
The best way is to concatenate the arrays containing the two heaps and re-heapify the resulting mess, requiring O(n) time (like buildHeap).

3. Why did we use an array for a binary heap? Does a heap need to be binary?
Since a binary heap is an almost complete binary tree, and grows/shrinks only from the end of the heap, an array can support it, we do not need the dynamic data structure like a linked list. The parent/child heap rule does not require a binary tree, but that along with the almost complete makes the lg n height possible.

Mergeable (Min)Heaps

Consider the following operations on heaps. Our goal is to support all of these operations in no worse than Theta(log n) and not be constrained to use an array or a binary tree.

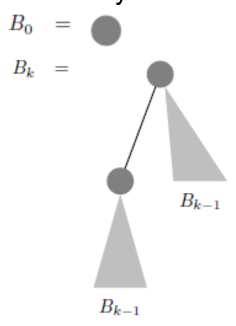
- Make-Heap: creates a new empty heap
- Insert: inserts a new element into a heap
- Minimum: returns the minimum element in a heap
- Extract-Min: returns the minimum element in a heap and removes it from the heap
- Union: creates a new heap consisting of the elements of two existing heaps

Two types of mergeable heaps are Binomial heaps and Fibonacci heaps. They also support these operations.

- Decrease-Key: changes the value of some element in a heap to a smaller value
- Delete: removes an element from a heap

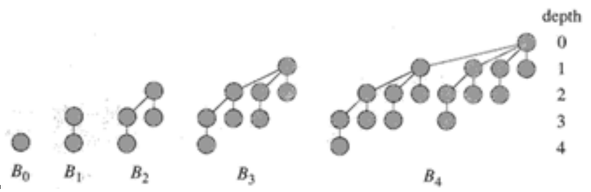
Binomial Heaps (utilizing binomial trees)

A binomial tree is an ordered tree defined recursively:



1. How many nodes does B_k have, what is its height, and how many children does its root have?
It follows directly from this definition that B_k has 2^k nodes, the height of B_k is k, the root of B_k has k children. As such, the height of B_k and the maximum degree of B_k are both logarithmic in the number of nodes, yielding a "bushy" tree.

2. Why are they called binomial trees?
It is because the number of nodes at a given depth in the tree is determined by the binomial coefficient.



A binomial heap is a collection (linked list) of binomial trees in which each binomial tree is heap-ordered: each node is greater than or equal to its parent. Also, in a binomial heap at most one instance of B_i may occur for any i.

3. How many binomial trees are needed at most in the linked list of roots to make a binomial heap of n nodes?
Since each B_i can occur at most once, we can make an analogy between binomial heaps and binary numbers, where B_0 is represented by the rightmost (least significant) bit, B_1 is represented by the second rightmost bit, and so on. Since the binary representation of an integer n has ⌊lg n⌋+1 bits, a binomial heap of n nodes consists of at most ⌊lg n⌋ + 1 binomial trees.

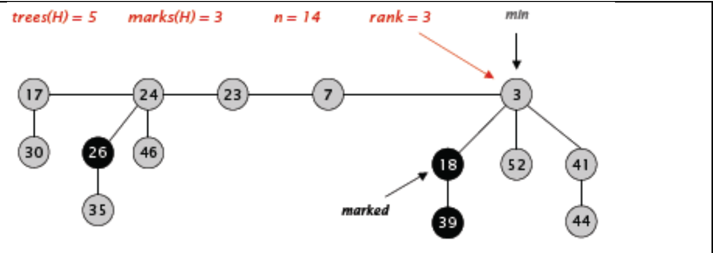
4. See https://www.cs.usfca.edu/~galles/visualization/BinomialQueue.html to help describe how each operation is done, and its run time:
ALSO SEE CS430Lecture19Activities(binomial heap details).ppt
Make-Heap: **To create a new heap, we simply allocate the necessary space and return an empty heap in O(1) time.**
Minimum: **The minimum element must be the root of one of the trees in the heap. Since there are no more than floor(lg n)+1 trees, this consists simply of finding the minimum of each root, which can be done in O(log n) time.**
Union: **To join two binomial heaps, we begin by merging their collections of trees. Since there may now be two trees of the same degree, we perform an operation analogous to binary addition to combine trees as necessary.**
Since there are no more than ⌊lg n⌋+1 pairs of trees to combine in this fashion, this can be done in O(log n) time.
Insert: **Insertion is a special case of union: we create a singleton binomial heap containing the element we would like to insert and compute the union of this heap with the original heap. This clearly takes O(log n) time.**

Extract-Min: **Since the minimum element must be at the root of one of the subtrees, we first compare these elements. We remove the root from its subtree, breaking it into a collection of smaller trees. We then union of these pieces with the other trees in the heap. This takes Theta(log n) time.**
Decrease-Key: **We begin by decreasing the key as necessary, and then we "bubble up" the node through the heap so the heap-ordering property holds. Since the height of a tree in the heap is at most $\lfloor \lg n \rfloor$, this takes Theta(log n) time.**
Delete: **We can delete by applying Decrease-Key and Extract-Min: decrease the key to some value smaller than any other in the heap (CLRS3 uses $-\infty$, although any small enough value is sufficient), and then extract this element from the heap as the new minimum. This yields a time bound of Theta/(log n).**

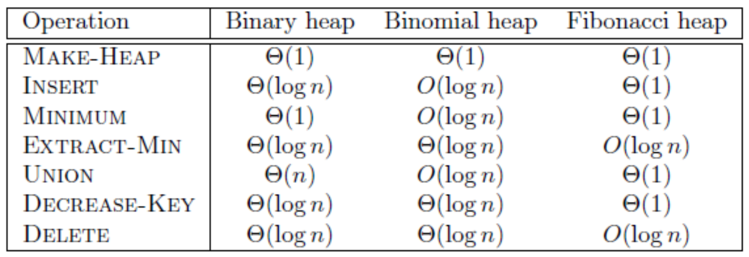
<u>Fibonacci Heaps</u>
Fibonacci heaps which support heap operations that do not delete elements in constant amortized time. From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many graph algorithms.

In essence, a Fibonacci heap is a "lazy" binomial heap in which the necessary housekeeping is delayed until the last possible moment: deletion.

- Set of Heap ordered trees (each parent smaller than children)
- Maintain pointer to minimum element (find-min takes O(1) time)
- Set of marked nodes (if one of its children has been removed)
- n – number of nodes in the heap
- rank(x) – number of children of node x
- rank(H) – max rank of any node in heap H
- trees(H) – number of trees in heap H
- marks(H) – number of marked nodes in H



5. See https://www.cs.usfca.edu/~galles/JavascriptVisual/FibonacciHeap.html
and https://www.cs.usfca.edu/~galles/JavascriptVisual/FibonacciHeap.html to help describe how each operation is done, and a rough estimate on its run time:
ALSO SEE CS430Lecture19Activities(fibonacci heap details).ppt
Make-Heap: **As in binomial heaps, making a new heap consists of simply allocating the necessary space. As the potential of the empty heap is 0, the amortized cost of this operation is simply its actual cost, O(1).**
Insert: **The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root (with no attempt to consolidate it with the existing trees). If the new node is smaller than the current minimum, we also update the min pointer. Update trees(H), n, and possibly min**
Minimum: **As we maintain a pointer to the minimum element in the heap, we can simply follow this pointer. The potential does not change, so the amortized time of this operation is simply its actual time, O(1).**
Union: **Two heaps are united by simply combining their tree lists and updating all the heap variables, as before with no attempt to consolidate trees. The minimum of the resulting Fibonacci heap is the smaller of the minimums for the uncombined trees.**
Extract-Min: **When we extract the minimum node, we must perform the housekeeping we postponed until this point. Note that simply returning a pointer to the minimum node is trivial, as is physically removing it from the structure. First making a root (insert in the root list to the left of the min) out of each of the minimum node's children and removing the minimum node from the root list (new possible min is to the right of node removed, or one of the old mins children). It then consolidates the root list by linking roots of equal degree (as we walk the root list to the right of the new min and wrapping around, use an array of pointers for each root degree and when find a second of the same size merge the trees as in binomial heap) until at most one root remains of each degree**
Decrease-Key: **As in binomial heaps, when we decrease the key of a node, the modified node may violate the heap property. In spirit with the lazy approach, rather than repairing the situation on the spot, we can simply remove the subtree rooted at the decreased node from the tree it is in and place it at the top level. However, we need to restrain this activity. The way we do this is by indicating in a node, by marking it, if a child has been removed. If a node is already marked and we must remove a (second) child, the node itself is removed from its parent, and the process continues upward until an unmarked node or the root is encountered.**
Delete: **As in binomial heaps, this can be implemented simply by decreasing the key of the node to a value smaller than any other in the heap and then extracting the minimum.**



Operation	Binary heap	Binomial heap	Fibonacci heap
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

Note that the times indicated for the Fibonacci heap are amortized times while the times for binary and binomial heaps are worst-case per-operation times.

Also note that the nodes of the heap can also be reference in a dictionary data structure (balanced binary tree) to enable quick lookup of individual nodes.

CS430 Lecture 20 Activities - KEY

Disjoint-set Data Structure

It is useful in many applications to have a structure that handles groups of disjoint sets. In particular, we support the following three operations:

- Make-Set(x): Creates a new set consisting of a single element x. Since the sets are disjoint, we assume that this element is not already contained in any set.
- Union(x,y): Given elements in two different sets, forms the union of two existing sets into one new set.
- Find-Set(x): Returns a pointer to the representative of the set containing element x.

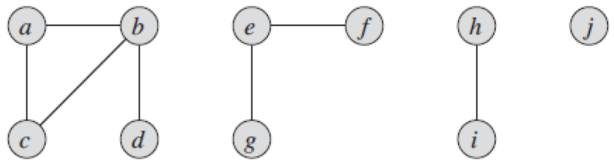
To identify a set, we return a pointer to any element in the set. The only constraint we make on the element chosen is that if the set does not change between calls to Find-Set, we must return the same representative.

We analyze the algorithms implementing these operations in terms of n, the number of Make-Set operations (all Make-Sets are usually assumed to run first), and m, the total number of Make-Set, Union, and Find-Set operations (n <=m)

1. Give then above, how many possible Union operations might there be? **At most n-1**

Disjoint-set Application

A graph data structure is a set of vertices and edges between those vertices, and supports problems where there can be relationships between any two items (vertices)

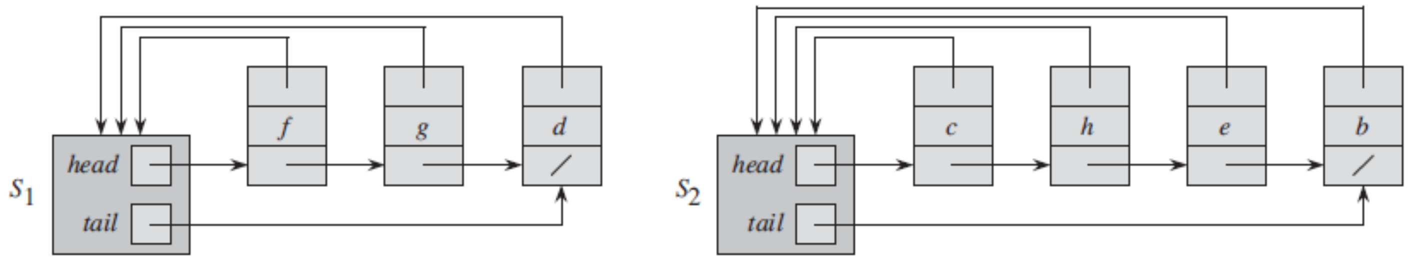


1. It is easy to see the disjoint sets (connected components) of the graph. Given a graph G=(V, E) write an algorithm using Make-Set, Union, Find-Set to find the connected components of any undirected graph

```
CONNECTED-COMPONENTS(G)
  for each vertex v in G
    MAKE-SET (v)
  for each edge (u,v) in G
    if FIND-SET(u) != FIND-SET(v)
      UNION (u,v)
```

// If actually implementing connected components, each vertex needs a handle to its object in the disjoint-set data structure, each object in the disjoint-set data structure needs a handle to its vertex.

Linked-list representation



One simple approach is to represent a set as an unordered linked list. Each element contains two pointers, one to the next element (as in a simple linked list) and one to the head of the list. The head serves as the set representative.

2. Describe the algorithms for Make-Set(x) and Find-Set(x) including runtime. For Find-Set(x) assume you already have a reference to x.

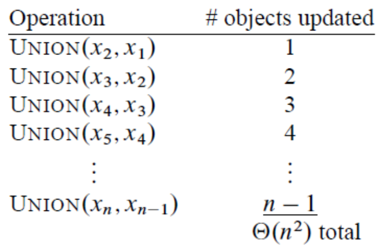
To carry out MAKE-SET(x), we create a new linked list whose only object is x. For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that head points to. Both require constant time (assuming you already have a reference to the element you are calling Find-Set on).

3. How do you think Union(x,y) is implemented?

We first attach one linked list, y, to the end of the other, x, but we then must update the pointers to the head for every element of y to point to the head of x and delete the S2 set object

4. For n total elements, what is the maximum number of Make-Set and Union operations that would need to be called in the worst case to get all the elements in one set? What is the maximum amortized runtime of each union?

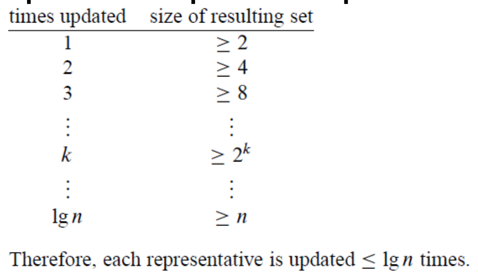
If appending a large list onto a small list, it can take a while. Amortized time per union operation = THETA(n)



Operation	# objects updated
$\text{UNION}(x_2, x_1)$	1
$\text{UNION}(x_3, x_2)$	2
$\text{UNION}(x_4, x_3)$	3
$\text{UNION}(x_5, x_4)$	4
$\vdots$	$\vdots$
$\text{UNION}(x_n, x_{n-1})$	$n-1$
	$\Theta(n^2)$ total

5. For n total elements, what is the minimum number of Make-Set and Union operations that would need to be called in the best case to get all the elements in one set? What is the lower bound amortized runtime of each union?

Always append the smaller list to the larger list. (Break ties arbitrarily.) How many times can each object's representative pointer be updated? It must be in the smaller set each time.



times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

Therefore, each representative is updated $\leq \lg n$ times.

Amortized time per union operation = THETA(lg n)

CS430 Lecture 21 Activities - KEY

Opening Questions

1. Give an example NOT discussed in the video lecture of a problem that can be represented by a graph.

Many options

2. If there is a path in a graph from a vertex back to itself that is called a _____________

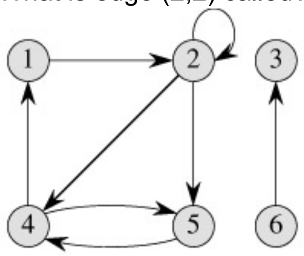
Cycle

3. Which representation of a graph, adjacency-list and adjacency-matrix, usually uses more memory and why?

Adjacency matrix, because both edges and non edges are represented.

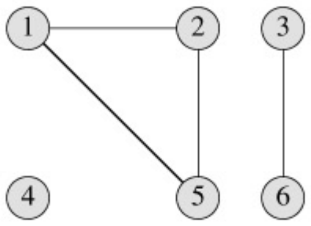
Graphs

1. Draw the graph: A directed graph G=(V,E), where V={1,2,3,4,5,6} and E={(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)}. What is edge (2,2) called?



(2,2) is self loop

2. Draw the graph: An undirected graph G=(V, E), where V={1,2 3,4,5,6} and E={{1,2},{1,5},{2,5},{3,6}}. What is vertex 4 called? What is different about how the edge set E is denoted for an undirected graph? Are self-loops allowed in an undirected graph?



The vertex 4 is isolated. No self-loops allowed. By convention, we use the notation (u, v) for an edge, rather than the set notation {u, v}, and (u, v) and (v, u) are considered to be the same edge.

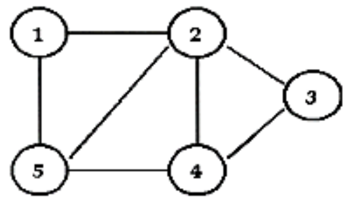
3. Define these terms:

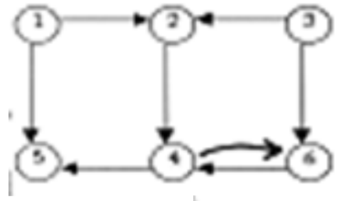
- vertex v is adjacent to vertex u in an undirected graph **{u,v} is in E**
- vertex v is adjacent to vertex u in a directed graph **(u,v) is in E**
- the degree of a vertex in an undirected graph **The degree of a vertex in an undirected graph is the number of edges adjacent to it. A vertex whose degree is 0 is isolated.**
- the degree of a vertex in a directed graph **In a directed graph, the out-degree of a vertex is the number of edges leaving it, and the in-degree of a vertex is the number of edges entering it. The degree of a vertex in a directed graph is its in-degree plus its out-degree.**
- a path in an undirected graph **a sequence <v0, v1, ..., vk> of vertices where each {v_{i-1}, v_i}∈E**
- a path in a directed graph **a sequence <v0, v1, ..., vk> of vertices following the direction of each edge where each (v_{i-1}, v_i)∈E**
- the length of a path **the number of edges in the path**
- v is reachable from u **there is a path from u to v**
- a simple path **all vertices in the path are distinct**
- a cycle in an undirected graph. What about a simple cycle? **In an undirected graph, a path <v0, v1,..., vk> forms a (simple) cycle if k >= 3, v0 = vk, and v1, v2,..., vk are distinct.**
- a cycle in a directed graph. What about a simple cycle? **In a directed graph, a path <v0, v1,..., vk> forms a cycle if v0 = vk and the path contains at least one edge. The cycle is simple if, in addition, v1, v2,..., vk are distinct. A self-loop is a cycle of length 1. A directed graph with no self-loops is simple.**
- Acyclic graph **A graph with no cycles**
- Connected undirected graph **An undirected graph is connected if every pair of vertices is connected by a path. The connected components of a graph are the equivalence classes of vertices under the "is reachable from" relation. An undirected graph is connected if it has exactly one connected component, that is, if every vertex is reachable from every other vertex.**
- Connected directed graph **A directed graph is strongly connected if every two vertices are reachable from each other. The strongly connected components of a directed graph are the equivalence classes of**

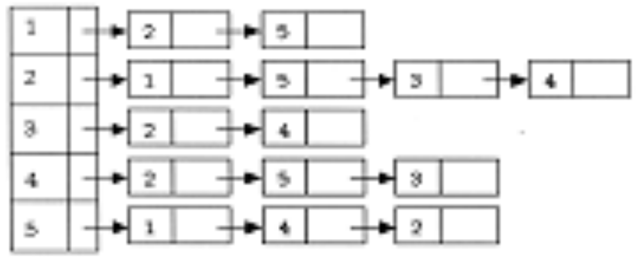
vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component.

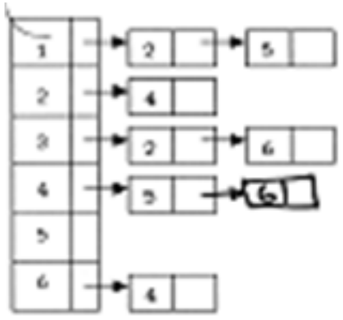
Graph Implementations

4. What is the adjacency list implementation of these two graphs?

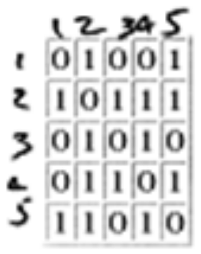




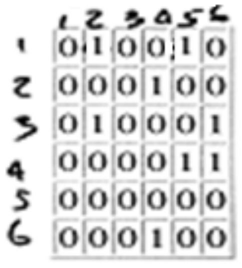




5. What is the adjacency matric implementation of the above two graphs?



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	0	1	0
2	0	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	0	0	1	1
5	0	0	0	0	0	0
6	0	0	0	1	0	0

6. How do the two implementations handle a weighted graph?

Adjacency list – add a weight on each node (along with vertex number)

Adjacency matrix – store the weight in the matrix instead of a 1

7. Two different representations of the graph data structure are discussed in the book, adjacency-list and adjacency-matrix. Please briefly discuss the runtime (in terms of |V| and |E| of these graph operations/algorithms using each implementation. Assume vertices are labeled as integers.

- What is the worst-case big-O runtime for checking to see if an edge from vertex u to vertex v exists?

Adjacency List – O(E)

Adjacency Matrix – O(1)

- How long does it take to compute the out-degree of every vertex of a directed graph?

With an adjacency-list representation of a directed graph, to compute the out-degree of a vertex you need to count the length of the adjacency-list for each vertex (there are |V| vertices). All the adjacency lists together sum up to length |E|, to compute the out-degree of every vertex is O(|V|+|E|). Unless the head of the adjacency list keeps track of how long the adjacency list is, then to compute the out-degree of every vertex is O (|V|).

Adjacency Matrix = O(V*V)

- How long does it take to compute the in-degree of every vertex of a directed graph?

With an adjacency-list representation of a directed graph, to compute the in-degree of a vertex you need to count the how many times that vertex appears in the adjacency-list for every other vertex. If you do this one vertex at a time (the in-efficient way), each vertex costs O (|E|) (check every edge) and since there are |V| vertices, to compute the in-degree of every vertex is O (|V|*|E|). However, with the addition of an array (O (|V| memory) to count up the in-degree of each vertex as you make a single traversal of the entire adjacency list (all edges), then to compute the in-degree of every vertex is O (|V|+|E|).

Adjacency Matrix = O(V*V)

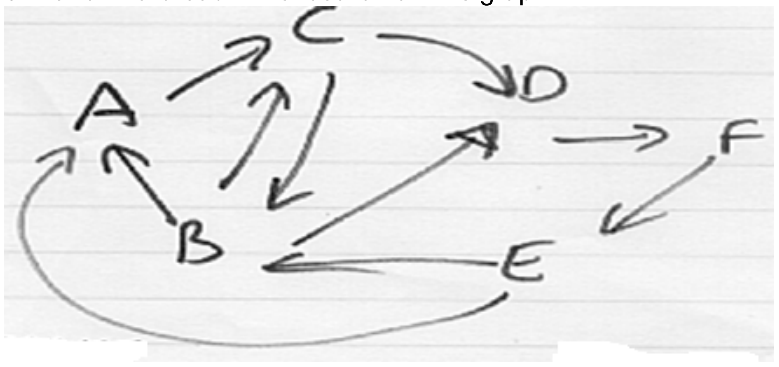
Graph Traversals

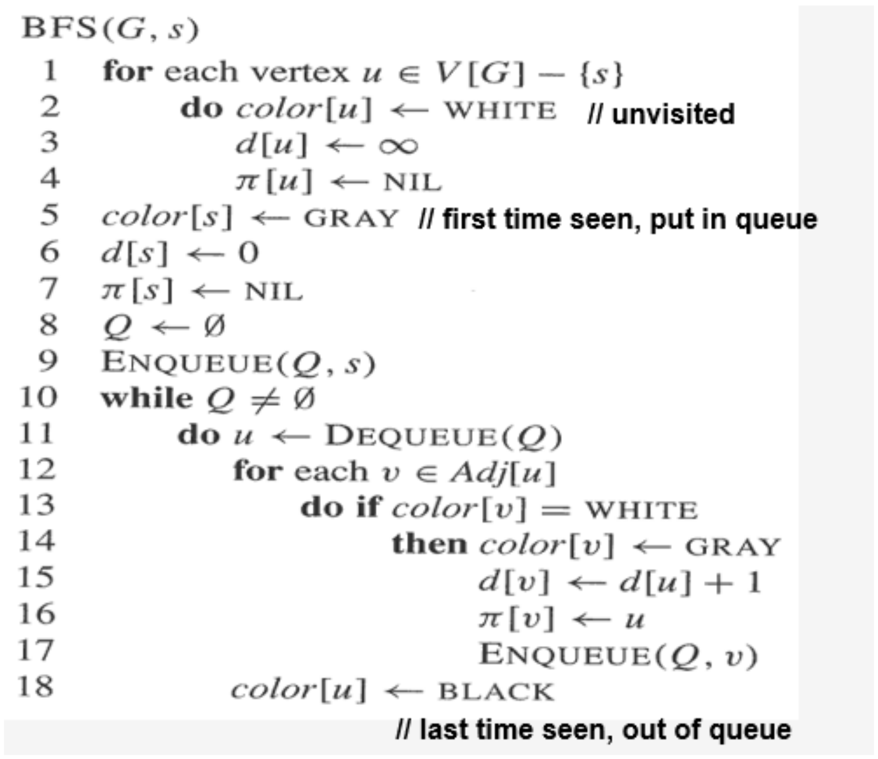
A way to search / visit all the vertices in a graph. There is not a unique answer usually.

- Undirected graph - if connected, all vertices will be visited
- Directed graph - Must be strongly connected to be able to visit all vertices

Breadth first - visit vertices one edge from a given (or random) source, two edges from source, etc. Uses a queue and some way to mark a vertex as visited (white initially, gray when first visited and put in queue, black when out of queue), label a vertex with how far from the source, and label a vertex with how its predecessor vertex was during the traversal.

8. Perform a breadth first search on this graph.





```
BFS($G, s$)
for each vertex $u \in V[G] - \{s\}$
        do $color[u] \leftarrow$ WHITE   // unvisited
            $d[u] \leftarrow \infty$
            $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ GRAY   // first time seen, put in queue
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
        do $u \leftarrow$ DEQUEUE($Q$)
            for each $v \in Adj[u]$
                do if $color[v] =$ WHITE
                    then $color[v] \leftarrow$ GRAY
                        $d[v] \leftarrow d[u] + 1$
                        $\pi[v] \leftarrow u$
                        ENQUEUE($Q, v$)
            $color[u] \leftarrow$ BLACK
                    // last time seen, out of queue
```

CS430 Lecture 22 Activities - KEY

Opening Questions

1. What is the runtime for breadth first search (if you restart the search from a new source if everything was not visited from the first source)?

Time = *O(V + E)*.
***O(V)* because every vertex enqueued at most once.**
***O(E)* because every vertex dequeued at most once and we examine *(u, v)* only when *u* is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.**

2. Does a breadth first search always reach all vertices?

You only reach vertices that are connected to the source. No, not if the undirected graph is not connected and the directed graph is not strongly connected.

3. How can you use a breadth first search to find the shortest path (minimum number of edges) from a given source vertex to all other vertices?

BFS finds the distance (# of edges) to each reachable vertex in a graph *G* = (*V*, *E*) from a given source vertex *s* ∈ *V*. Define the *shortest-path distance* δ(*s*, *v*) from *s* to *v* as the minimum number of edges in any path from vertex *s* to vertex *v*; if there is no path from *s* to *v*, then δ(*s*, *v*) = ∞.

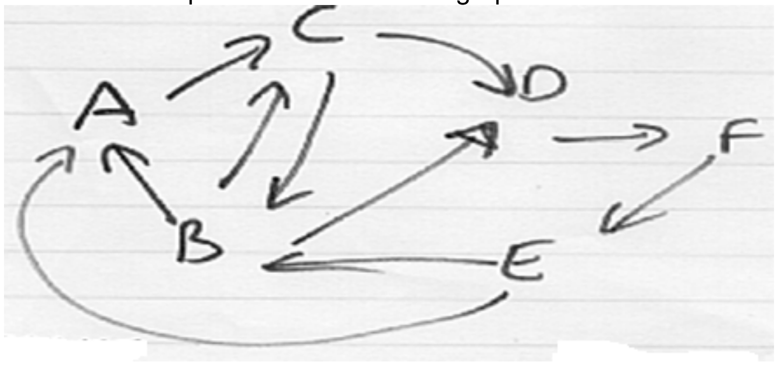
4. If you look at the predecessor edges which were used to connect to an unvisited vertex, what do these predecessor edges form? Is it unique for a graph?

The procedure BFS builds a breadth-first tree from the source as it searches the graph. It is not unique, it depends on the order that adjacent vertices are put in the queue.

Depth First Search

As we visit a vertex we try to move to a new adjacent vertex that hasn't yet been visited, until nowhere else to go, then backtrack. Uses a stack and some way to mark a vertex as visited (white initially, gray when first visited and put in stack, black when out of stack), label a vertex with a counter for first time seen, and another counter for last time seen (we will see why later), and label a vertex with how its predecessor vertex was during the traversal.

1. Perform a depth first search on this graph.



```
DFS(G)
1  for each vertex u ∈ V[G]
2      do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6      do if color[u] = WHITE
7          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY          ▷ White vertex u has just been discovered.
2  time ← time +1
3  d[u] ← time
4  for each v ∈ Adj[u]      ▷ Explore edge (u, v).
5      do if color[v] = WHITE
6          then π[v] ← u
7              DFS-VISIT(v)
8  color[u] ← BLACK         ▷ Blacken u; it is finished.
9  f[u] ← time ← time +1
```

2. What is the runtime for depth first search (if you restart the search from a new source if everything was not visited from the first source)?

Time = *O(V + E)*.
***O(V)* because every vertex pushed at most once.**

***O(E)* because every vertex popped at most once and we examine *(u, v)* only when *u* is popped. Therefore, every edge examined at most once.**

Another interesting property of depth-first search is that the search can be used to classify the edges of graph G based on how they are traversed

- ***Tree edges*** are edges in the depth-first forest G_π. Edge (*u*, *v*) is a tree edge if *v* was first discovered by exploring edge (*u*, *v*).
- ***Back edges*** are those edges (*u*, *v*) connecting a vertex *u* to an ancestor *v* in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
- ***Forward edges*** are those non-tree edges (*u*, *v*) connecting a vertex *u* to a descendant *v* in a depth-first tree.
- ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

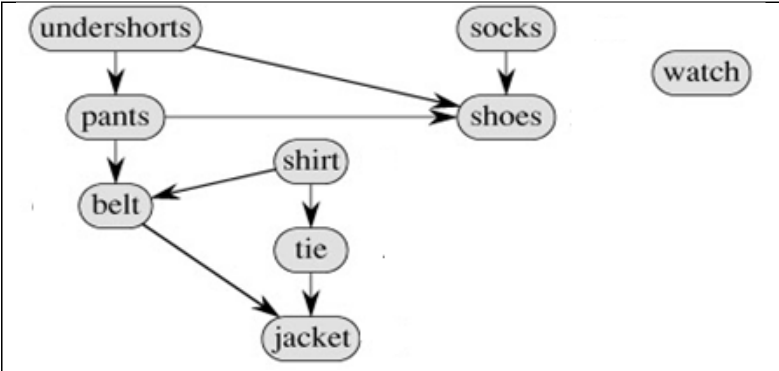
3. If a graph has no back edges when completing a depth first search what does that tell us about the graph?
A directed graph is acyclic if and only if a depth-first search yields no "back" edges

demo BFS/DFS http://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Topological sort (a DFS application)

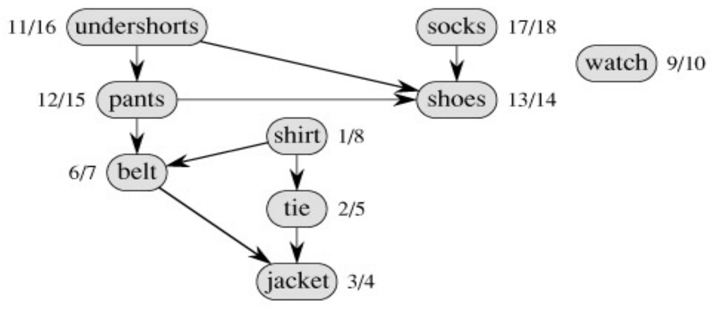
- A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.)
- A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied earlier.
- Directed acyclic graphs are used in many applications to indicate precedence among events
- A depth-first search can be used to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called.

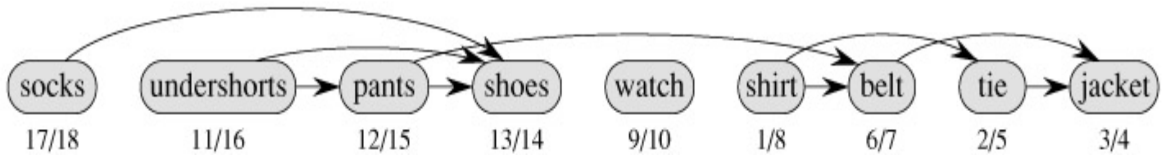
4. Perform a topological sort on this graph.



TOPOLOGICAL-SORT(*G*)

```
1. call DFS(G) to compute finishing times f[v]
   for each vertex v
2. as each vertex is finished, insert it onto the
   front of a linked list
3. return the linked list of vertices
```





5. Why does the topological sort work? What is its runtime?
Each node n gets prepended to the output list L only after considering all other nodes which depend on n (all descendants of n in the graph). Specifically, when the algorithm adds node n, we are guaranteed that all nodes

which depend on n are already in the output list L: they were added to L either by the recursive call to visit() which ended before the call to visit n, or by a call to visit() which started even before the call to visit n. Since each edge and node is visited once, the algorithm runs in linear time.

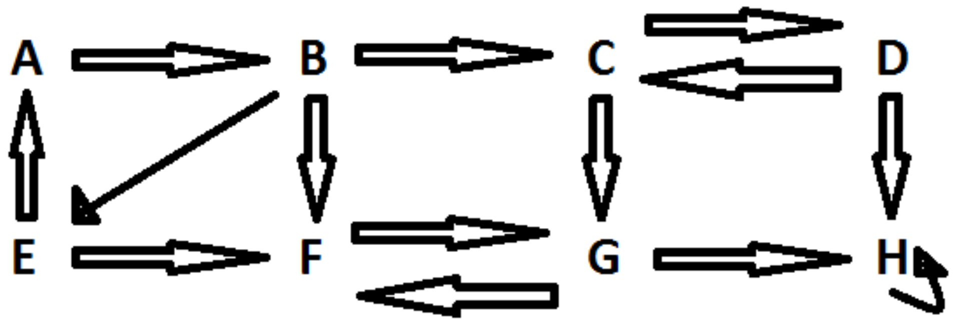
<u>The Parenthesis Theorem</u>
The parenthesis theorem tells us that, for two vertices u, v ∈ V , it cannot be the case that d[u] < d[v] < f[u] < f[v]; that is, the intervals [d[u], f[u]] and [d[v], f[v]] are either disjoint or nested. This is a simple consequence of the depth-first nature of DFS. If the algorithm discovers u and then discovers v, it cannot later back out of u without first backing out of v.

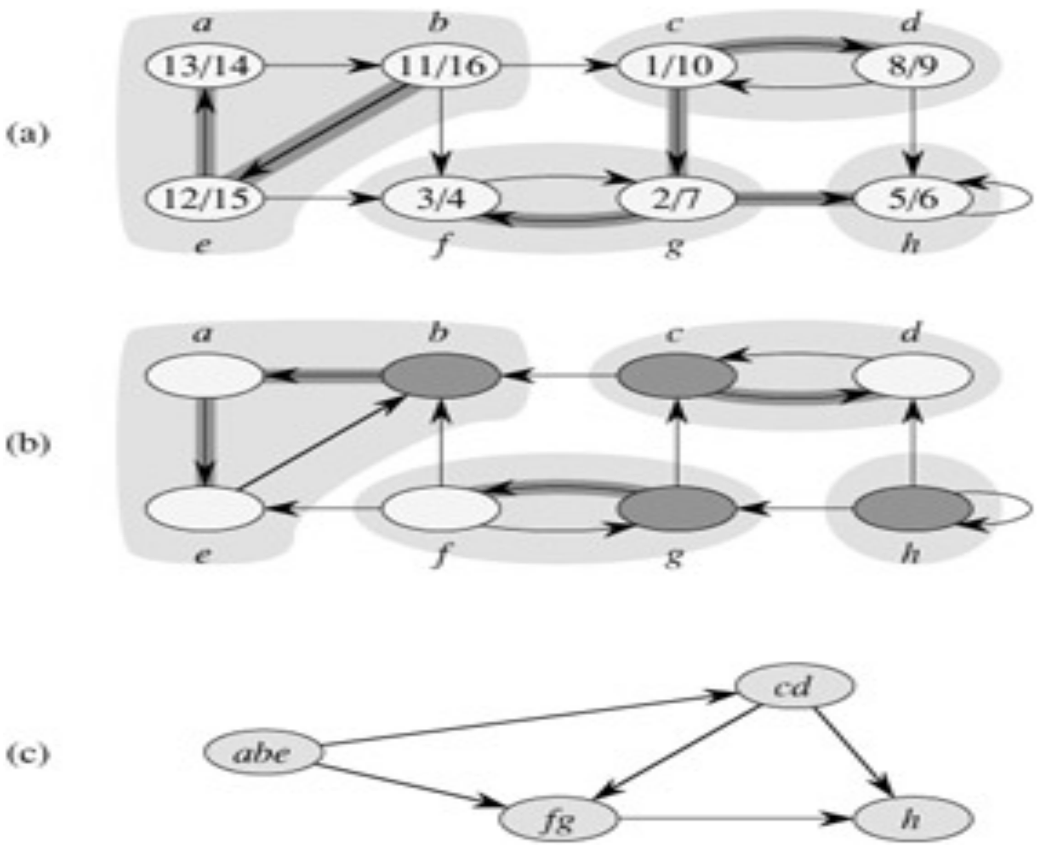
<u>Strongly Connected Components (a DFS application)</u>
A graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

STRONGLY-CONNECTED-COMPONENTS (*G*)

1. call DFS (*G*) to compute finishing times *f*[*u*] for each vertex *u*
2. compute G^T (the transpose of the graph)
3. call DFS (G^T), but in the main loop of DFS, consider the vertices in order of decreasing *f*[*u*] (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

6. Find the strongly connected components.





6. Discuss: G and G^T will have the same strongly connected components.

7. Discuss: The component with the latest finish time vertex will have no edges in the transpose to any other component.

CS430 Lecture 23 Activities - KEY

Opening Questions

1. What is the difference between a tree and a graph?

Trees have no cycles. Tree is a graph with only one path between any two nodes (vertices). Tree is a connected graph with v vertices and v-1 edges

2. Give a recursive definition for a tree.

Parent node with 0 or more children, each of which itself is a tree

3. In a weighted undirected graph, what is the difference between a minimum spanning tree and a shortest path in a graph?

A minimum spanning tree is made up of the edges of minimum total weight that still span (connect) all vertices in the graph. This does not assure the minimum path distance between all vertices.
A shortest path is the minimum path distance between two vertices.

4. Since shortest paths contain shortest sub-paths (optimal substructure), name an algorithmic approach that we might try to find a shortest path in a graph.

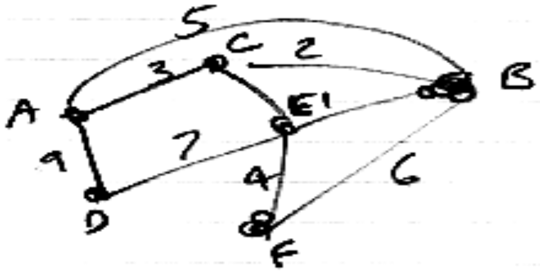
Greedy algorithm **Dynamic programming**

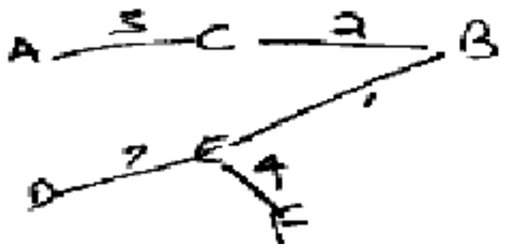
Minimum Spanning Trees

1. Give a definition of a Minimum Spanning Tree, and find a MST of the below graph.

Min Total Tree Weight That Spans Graph, Weighted/Undirected graph - Touch every vertex, No cycles

$$Minimize\left(\sum_{\{u,v\}\in T} w(u,v)\right)$$

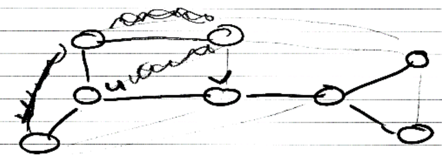




2. Prove a Minimum Spanning Tree has optimal substructure.

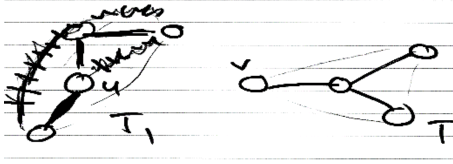
Start with problem and an optimal MST solution. If I remove an edge from the optimal MST solution, am I left with two optimal MST sub-solutions to sub-problems of the original problem?

ASSUME WE HAVE OPTIMAL MST FOR $G=\{V,E\}$



REMOVE ANY EDGE FROM MST (u,v)

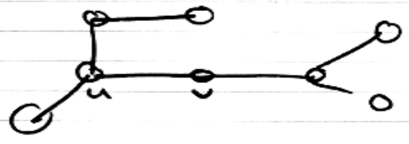
THIS WILL ~~ALWAYS~~ YIELD 2 SUB TREES



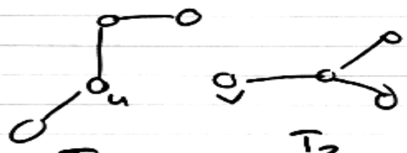
$V_1+V_2=V$

$E_1+E_2 \le E$

CLAIM
T_1 IS MST FOR $G_1=\{V_1,E_1\}$
T_2 IS MST FOR $G_2=\{V_2,E_2\}$



MST T FOR $G=(V,E)$



T_1 IS MST FOR $\overset{\text{SET}}{G_1} = \{\overset{\text{SET}}{V_1}, E_1\}$

$V_1 \uplus V_2 = V$ SET OF VERTICES

T_2 IS MST FOR $G_2=\{V_2,E_2\}$

$E_1 + E_2 + (u,v) + \boxed{\text{OTHER EDGES}}$ FROM V_1 TO V_2 $= E$

MIN? YES

$w(T) = w(T_1) + w(T_2) + w(u,v)$

$\uparrow$

IF $w(T_x) < w(T_1)$

THEN $w(T)$ WAS NOT OPTIMAL

CONTRADICTION

$w(T)$ IS OPTIMAL

IF $\exists\, T_x$ WAS A MORE OPTIMAL MST FOR G_1

CANNOT HAPPEN

3. What are some possible greedy approaches to find a Minimum Spanning Tree? Prove correct or show counterexample
Divide and conquer – split graph in half repeatedly til base case, choose shortest edge to connect the subgraphs.
WILL NOT WORK
Prim's – pick minimum edge from visited vertex set to unvisited vertex set

$G = \{V, E\}$

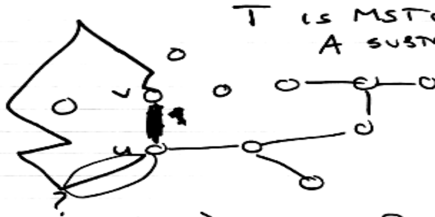
FIND MST T

PROVE GREEDY CHOICE (PRIM) LEADS TO OPTIMAL SOLUTION

ASSUME WE HAVE MST

IF MST COULD BE BUILT USING GREEDY CHOICE $\Rightarrow$ DONE (WITH INDUCTION)

ELSE MST COULD NOT BE BUILT USING GREEDY CHOICE, SHOW WE COULD CHANGE PART OF MST TO INCLUDE GREEDY AND STILL BE OPTIMAL



T IS MST OF G

A SUBTREE OF T

$A \subseteq T$

MST SO FAR

(u,v) IS THE PRIM GREEDY CHOICE

$\rightarrow$ IF $(u,v) \in T \Rightarrow$ DONE, INDUCTION

$\rightarrow$ IF $(u,v) \notin T$, NEED TO SHOW THAT T IS NOT OPTIMAL (CONTRADICTS THAT T WAS OPTIMAL)

IF EDGE (u,v) IS NOT IN T, THERE MUST BE ANOTHER PATH FROM u TO v IN T

IN THAT PATH, REPLACE THE FIRST EDGE FROM u TO ? WITH (u,v) T IS STILL SPAN TREE

$w(u,v) < w(u,?)$

Kruskal's Algorithm – Choose minimum weight edge that does not connect two vertices that are already in the same connected component (don't form cycle)

ASSUME YOU HAVE AN OPTIMAL $\overset{\text{MST}}{\text{ANSWER}}$ TO $G=(V,E)$

$A \in E$

MST

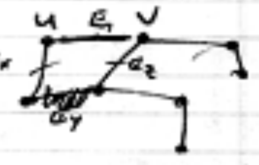
SORT EDGES E BY INCREASING WEIGHT

~~[illegible]~~ $A = \{ E_? \; E_{?} \; \dots \}$

$E_1 \leq E_2 \leq E_3 \dots \leq E_n$

IF $E_1 \in A$, THEN A CONTAINS GREEDY CHOICE, CONTINUE INDUCTING DONE

IF $E_1 \notin A$ GREEDY CHOICE IS NOT IN MST ANSWER A



$E_1 = (u,v)$

TAKE ANY EDGE ALONG PATH FROM u TO V

$E_1 \leq E_y$

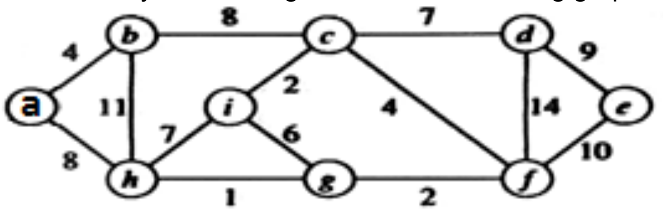
$E_x, E_y, E_2 \geq E_1$

$A + E_1 - E_y$

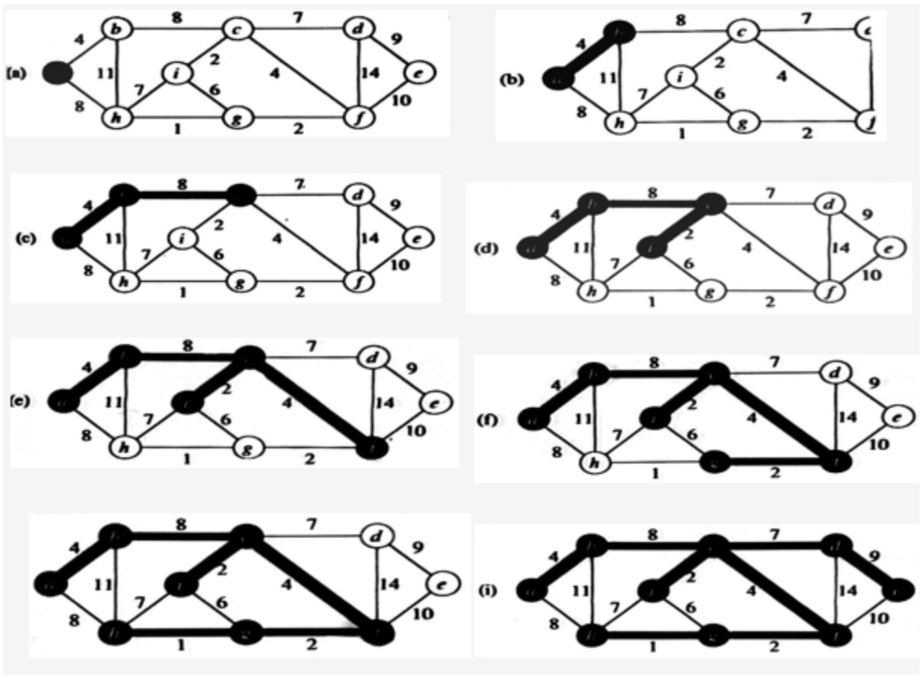
REPLACE E_y WITH E_1

$\leq A$

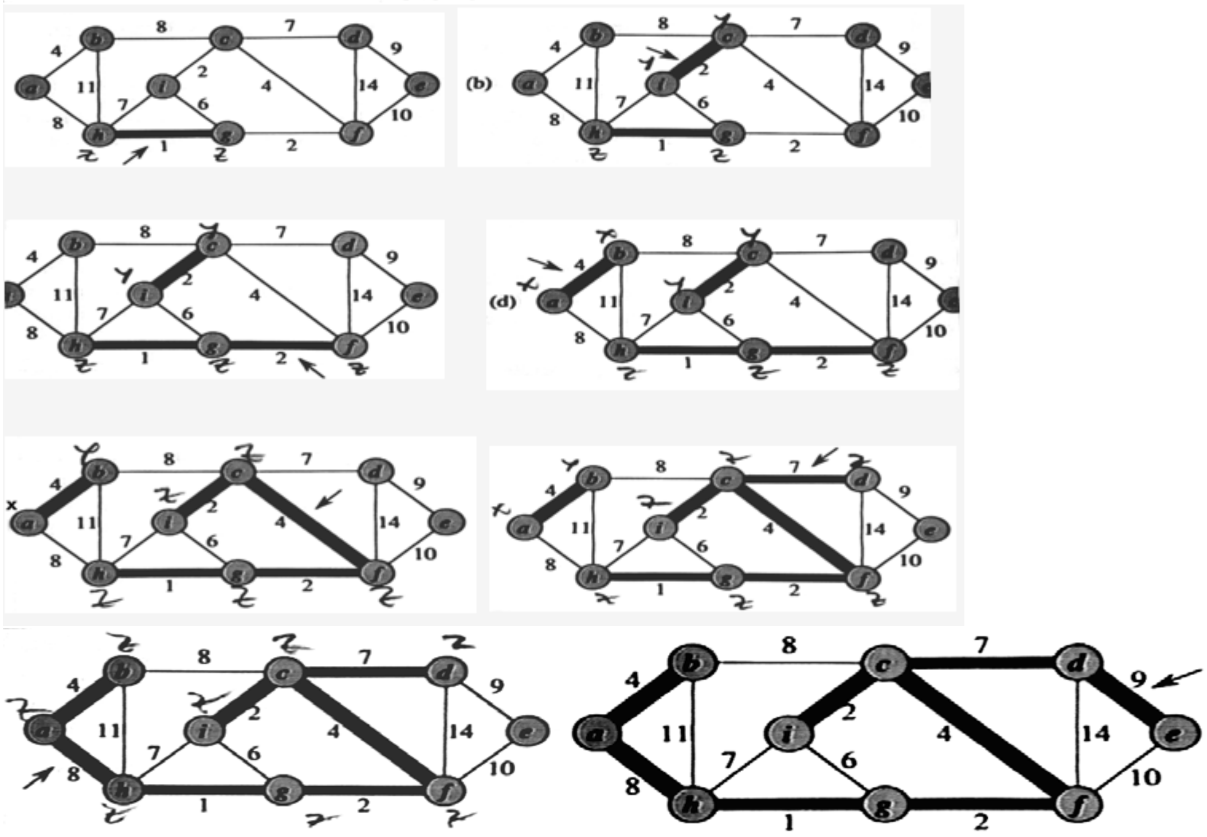
4. Demonstrate your MST algorithm on the following graph and write pseudocode.



```
MST-PRIM(G, w, r)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = $ NIL
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN($Q$)
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u,v) < v.key$
            $v.\pi = u$
            $v.key = w(u,v)$
```



```
MST-KRUSKAL(G, w)
1  A ← Ø
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing
5  for each edge (u, v) ∈ E, taken in non
6       do if FIND-SET(u) ≠ FIND-SET
7             then A ← A ∪ {(u, v)}
8                  UNION(u, v)
```



demo prim http://en.wikipedia.org/wiki/File:Prim-algorithm-animation-2.gif

demo kruskal https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

CS430 Lecture 24 Activities - KEY
Shortest Path Problem

How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbf{R}$

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$= \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$=$ sum of edge weights on path p.

Shortest-path weight u to v:

$$\delta(u, v) = \begin{cases} \min \{ w(p) : u \overset{p}{\leadsto} v \} & \text{if there exists a path } u \leadsto v, \\ \infty & \text{otherwise.} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Variants

- Single-source: Find shortest paths from a given source vertex s ∈ V to every vertex v ∈ V.
- Single-destination: Find shortest paths to a given destination vertex.
- Single-pair: Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- All-pairs: Find shortest path from u to v for all u, v ∈ V. We'll see algorithms for all-pairs in the next chapter.

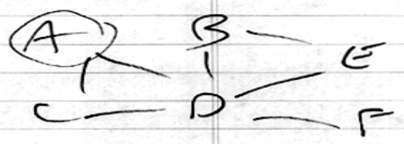
Negative-weight edges - OK, as long as no negative-weight cycles are reachable from the source.

- If we have a negative-weight cycle, just keep going around it, and get *w(s, v)*=-∞for all *v* on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph.

1. What would the brute force approach be to solve the shortest path problem, and what is its run time?

BRUTE FORCE

TRY ALL POSSIBILITIES
FOR PATHS, PICK SHORTEST.



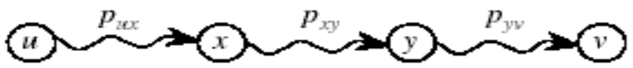
ø (A)
1 (AC) (AD)
2 (ACD) ADB ADC? ADE? ADF
3 ACDA ACDB ACDE ~~ACDF~~
ADCA A

2. Prove optimal substructure for the shortest path problem

Optimal substructure

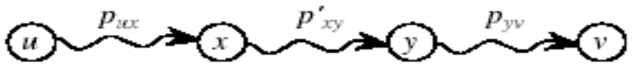
Lemma
Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



Suppose this path p is a shortest path from u to v.
Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\leadsto} y$.
Then $w(p'_{xy}) < w(p_{xy})$.
Construct p':



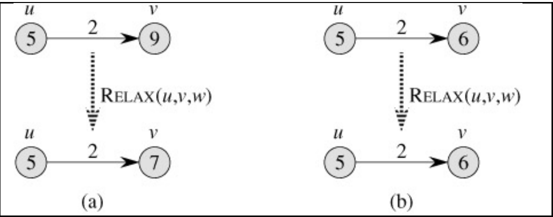
Output of single-source shortest-path algorithm
For each vertex $v \in V$:

- *d*[*v*] = *δ(s, v),* Initially, *d*[*v*]=∞, Reduces as algorithms progress. But always maintain *d*[*v*] ≥ *δ(s, v)*. Call *d*[*v*] a *shortest-path estimate*.
- *π*[*v*] = predecessor of *v* on a shortest path from *s,* If no predecessor, *π*[*v*] = NIL, *π* induces a tree—*shortest-path tree*

Initialization - All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    d[v]←∞
    π[v] ← NIL
d[s] ← 0
```

Relaxing an edge *(u, v)* - Can we improve the shortest-path estimate (best seen so far) from the source s to *v* by going through *u* and taking edge *(u, v)*?

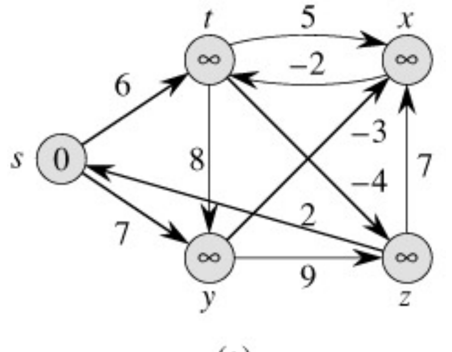
RELAX*(u, v, w)*
if *d*[*v*] > *d*[*u*] + *w(u, v)*
    d[*v*] ← *d*[*u*] + *w(u, v)*
    π[*v*]← *u*
	 u
	v
	5
	2
	9
	RELAX(u,v,w)
	5
	2
	7
	(a)
	u
	v
	5
	2
	6
	RELAX(u,v,w)
	5
	2
	6
	(b)

The algorithms differ in the order and how many times they relax each edge.

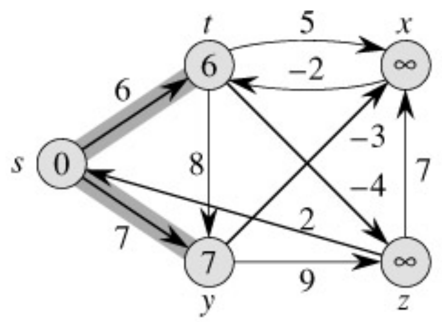
Shortest Path Algorithm - Bellman-Ford
The most straightforward of the "relax an edge" algorithms. Relaxes the edges in a fixed order (any fixed order) |v|-1 times. Not a greedy algorithm.

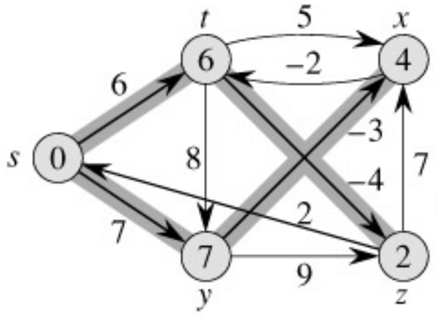
- Allows negative-weight edges.
- Computes *d*[*v*] and *π*[*v*] for all *v* ∈ *V*.
- Returns TRUE if no negative-weight cycles reachable from *s*, FALSE otherwise.

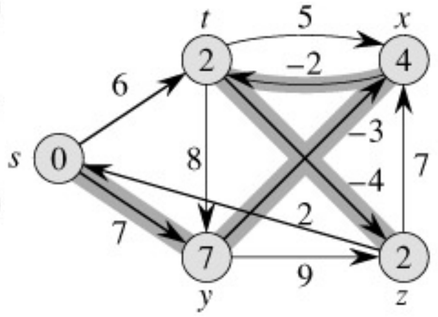
```
BELLMAN-FORD(V, E, w, s)
   INIT-SINGLE-SOURCE(V, s)
   for i ← 1 to |V|-1
      for each edge (u, v) ∈ E
         RELAX(u,v,w)
   for each edge (u, v) ∈ E   // all edges, in any order,
same order each time
      if d[v] > d[u] + w(u, v)
         then return FALSE
   return TRUE
```

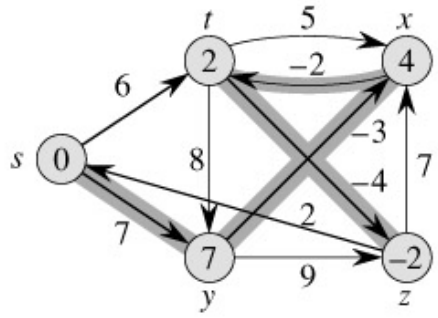

(a)

3. Execute Bellman-Ford on the above graph from source s for this edge order (*t, x*), (*t, y*), (*t, z*), (*x, t*), (*y, x*), (*y, z*), (*z, x*), (*z, s*), (*s, t*), (*s, y*). Update the d[v] and π[v] values for each iteration.


(b)


(c)


(d)


(e)

4. What is the runtime of Bellman-Ford?
The first for loop relaxes all edges |*V*|-1 times. *O(VE) = O(V^3)*

5. Prove Bellman-Ford is correct.
Values you get on each pass and how quickly it converges depends on order of relaxation. But guaranteed to converge after |V|-1 passes, assuming no negative-weight cycles.
Proof **Use path-relaxation property.**
Let *v* be reachable from *s*, and let *p* = *v*0*, v*1*, . . . , vk* be a shortest path from *s* to *v*, where *v*0 = *s* and *vk* = *v*. Since *p* is acyclic, it has ≤|*V*|-1 edges, so *k* ≤ |*V*|-1
Each iteration of the "for" loop relaxes all edges. In the worst case we happen to be relaxing the edges in reverse order on this longest path:

- **First iteration relaxes *(v*0*, v*1*)***
- **Second iteration relaxes *(v*1*, v*2*)***
- ***k*th iteration relaxes *(vk*-1*, vk)***

By the path-relaxation property, *d*[*v*] = *d*[*vk*] = *δ(s, vk) = δ(s, v)*.

CS430 Lecture 25 Activities - KEY

Opening Questions

1. We saw the Bellman-Ford algorithm found the shortest path from a source to all other vertices by “brute force” relaxing every edge in the graph in a fixed order |v|-1 times. Why did it need to do this |v|-1 times? And with this in mind could we improve on the Bellman-Ford for certain graphs?

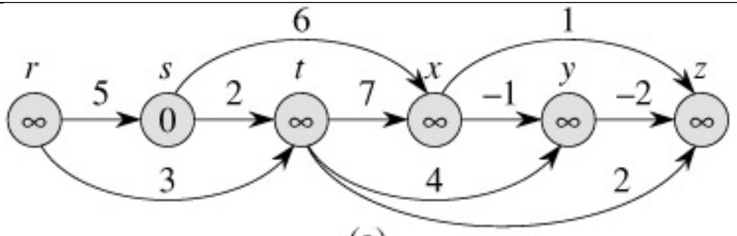
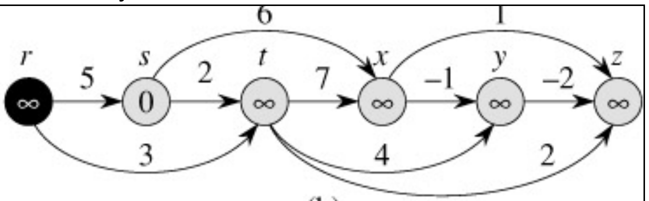
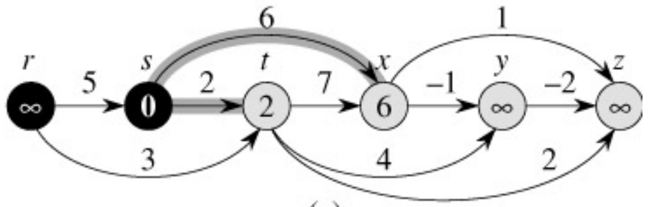
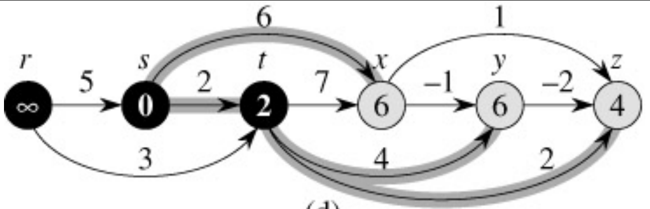
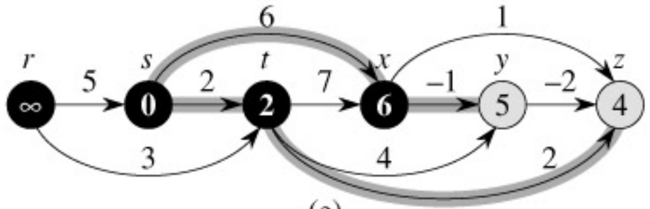
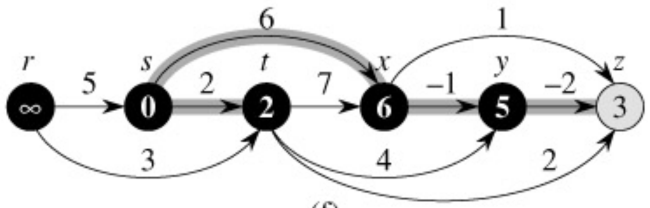
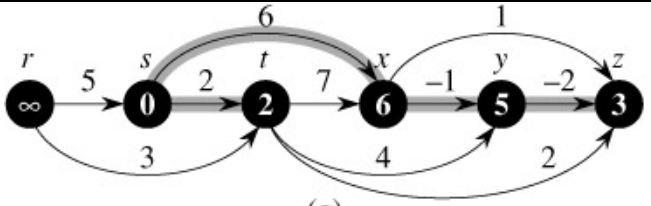
In the worst case the order we relax the edges is reverse order for the longest path. We could improve if we knew the order to relax the edges in. In an acyclic graph we could do a topological sort first and relax the edges in that order, once.

DAG Shortest Path Algorithm

By relaxing the edges of a weighted DAG (directed acyclic graph) *G* = (*V*, *E*) in topological sort order of its vertices, we can compute shortest paths from a single source. Shortest paths are always well defined in a DAG, since even if there are negative-weight edges, no negative-weight cycles can exist.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      do for each vertex v ∈ Adj[u]
5             do RELAX(u, v, w)
```

1. Here is the topological sort on a DAG. Find the shortest path from s to every other vertex.

 (a)	 (b)
 (c)	 (d)
 (e)	 (f)
 (g)	

2. What is the runtime for DAG Shortest Path?

Θ(*V* + *E*) time

3. Discuss why DAG Shortest Path is correct.
Because we process vertices in topologically sorted order, edges of any path must be relaxed in order of appearance in the path.

- **Edges on any shortest path are relaxed in order.**
- **By path-relaxation property, correct.**

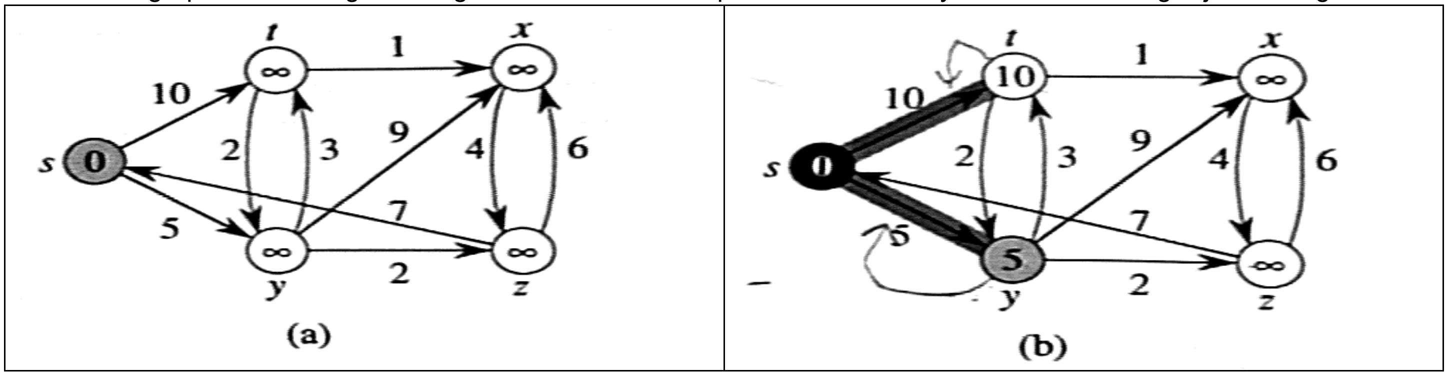
4. If we restrict the graph to having no negative edges, given a source s, what is the shortest path from s to one of its adjacent vertexes?
Since there are no negative edges once we relax the edges leaving the source s and pick the adjacent vertex with the smallest short path estimate we know that short path estimate is an actual shortest path to that one vertex. We could not get there any faster by going through any other vertex. The other adjacent vertexes may have their short path estimate improved upon.

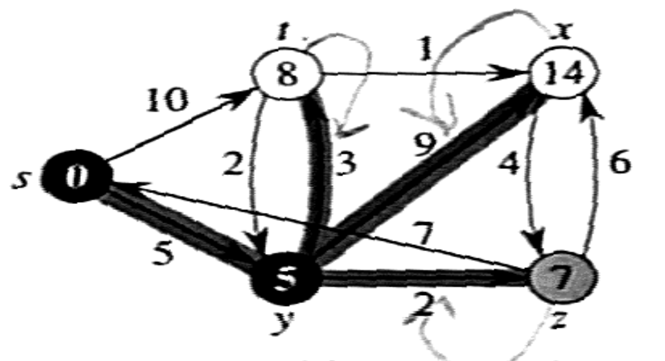
Dijkstra's Shortest Path Algorithm

- No negative-weight *edges*.
- Essentially a weighted version of breadth-first search.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weight estimates (*d*[*v*]).
- Have two sets of vertices:
  - *S* = vertices whose final shortest-path weights are determined,
  - *Q* = priority queue = *V*-*S*.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest") vertex in V-S to add to S.

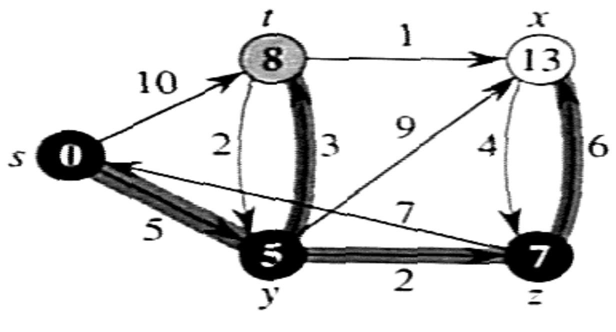
```
DIJKSTRA(V, E, w, s)
  INIT-SINGLE-SOURCE(V, s)
  S ← empty set
  Q ← V   // i.e., insert all vertices into Q by "d" values
  while Q not empty
    u ← EXTRACT-MIN(Q)
    S ← S ∪ {u}
    for each vertex v ∈ Adj[u]
       RELAX(u,v,w)   // possibly updates a short path estimate "d" value and moves the vertex forward in the queue
```

5. Here is a graph with no negative edges. Find the shortest path from s to every other vertex using Dijkstra's algorithm.

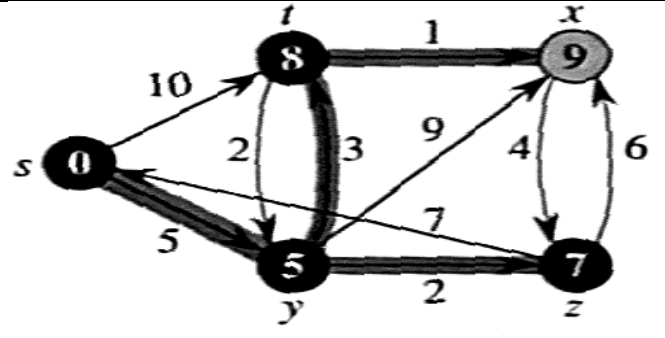




(c)



(d)



(e)

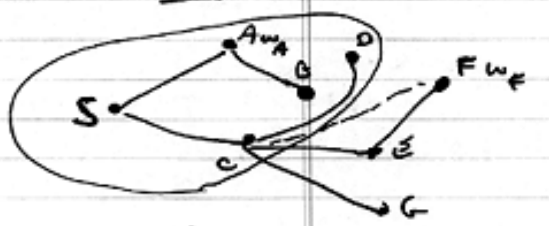
6. What is the runtime for Dijkstra's algorithm?
Like Prim's algorithm, depends on implementation of priority queue. If binary heap, each operation takes *O(*lg *V)* time, overall *O(E* lg *V)*.

7. Prove the greedy choice in Dijkstra's algorithm (pick the vertex with the smallest shortest path estimate, not including the vertices we are done with) leads to an optimal solution.
Assume we have a solution: we know the shortest path from V1 to every other vertex. "S" is the set of edges in the solution. If S does not contain the greedy choice at the last step, we can remove the non-greedy last edge added to S and add the greedy choice to S and get just as good a solution.

$G = (V, E)$

SOURCE $S \in V$

ASSUME WE HAVE ANSWER **S** WHICH IS SHORTEST PATHS FROM S TO ALL OTHER VERTICES



$W_?$ IS WEIGHT OF PATH FROM S TO ?

GREEDY CHOICE AT SOME PREVIOUS STEP WAS NOT CHOSEN

$$w_c + \overline{CF} < w_c + \overline{CE} + \overline{EF}$$

$\uparrow$ GREEDY CHOICE

IF $\overline{CF}$ GREEDY, $\overline{CP} < \overline{CE} + \overline{EP}$

CONTRADICTION, WE FOUND SHORTER PATH $S \rightarrow F$

Dijkstra's Algorithms http://oopweb.com/Algorithms/Documents/AnimatedAlgorithms/VolumeFrames.html

CS430 Lecture 26 Activities - KEY

<u>All-Pairs Shortest Paths Problem</u>

- Given a directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbb{R}$, $|V| = n$.
- Goal: create an $n \times n$ matrix of shortest-path distances from every vertex to every other vertex $\delta(u, v)$.
- Could run BELLMAN-FORD once from each vertex:
    - $O(V^2E)$ which is $O(V^4)$ if the graph is *dense* ($E =\sim V^2$).
- If no negative-weight edges, could run Dijkstra's algorithm once from each vertex:
    - $O(V E \lg V)$ with binary heap—$O(V^3 \lg V)$ if dense
- We'll see how to do in $O(V^3)$ in all cases with dynamic programming (we have already shown the shortest path problem has optimal substructure).

The formal problem statement:

- Assume that G is given as adjacency matrix of weights: $W = (w_{ij})$, with vertices numbered 1 to n.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j\ , \\ \text{weight of } (i, j) & \text{if } i \neq j, (i, j) \in E\ , \\ \infty & \text{if } i \neq j, (i, j) \notin E\ . \end{cases}$$

- Output is the shortest path matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$.

Dynamic Programming Steps

1. Define structure of optimal solution, including what are the largest sub-problems.
2. Recursively define optimal solution
3. Compute solution using table bottom up
4. Construct Optimal solution

To help us develop the first dynamic programming approach we can restate the All-Pairs Shortest Paths problem as follows.
Find the shortest path from every vertex to every other vertex considering at most paths of |V|-1 edges (longest simple path for |V| vertices).

1. Define structure of optimal solution.
The shortest path between any 2 vertexes i and j of at most |V|-1 edges is either:

- **The shortest path |V|-2 edges between i and j OR**
- **The shortest path of |V|-2 edges between i and a vertex adjacent to j plus that edge adjacent to j**

2. Recursively define optimal solution

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j\ , \\ \infty & \text{if } i \neq j\ . \end{cases}$$

Base Case: Smallest sub-problem is shortest paths of 0 edge or less.

Second smallest sub-problem is shortest paths of 1 edge or less is the adjacency matrix, W_{ij}

Let $l^{(m)}_{ij}$ = weight of shortest path from *i to j* that contains ≤ *m* edges.

$$\begin{aligned} & m \geq 1 \\ \Rightarrow l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ & (k \text{ is all possible predecessors of } j) \\ &= \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \\ & \text{since } w_{jj} = 0 \text{ for all } j\ . \end{aligned}$$

Shortest path matrix $D = (d_{ij})$, where $d_{ij} = l^{(n-1)}_{ij}$

<u>Slow All-Pairs Shortest Paths Algorithm</u>

Compute a solution bottom-up: Compute $L^{(1)} = W$, *then* $L^{(2)}$ *from* $L^{(1)}$ *, etc. . . ,* $L^{(n-1)}$

```
EXTEND$(L, W, n)$
  create $L'$ an $n \times n$ matrix
  for $i \leftarrow 1$ to $n$
    for $j \leftarrow 1$ to $n$
      $l'_{ij} \leftarrow \infty$
     for $k \leftarrow 1$ to $n$
      $l_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
  return $L'$

SLOW-APSP$(W, n)$
  $L^{(1)} \leftarrow W$
  for $m \leftarrow 2$ to $n - 1$
    $L^{(m)} \leftarrow$EXTEND$(L^{(m-1)}, W, n)$
  return $L^{(n-1)}$
```

3. What is the runtime of EXTEND and SLOW-APSP?
EXTEND: (n^3) SLOW-APSP: (n^4)

Improving on SLOW-APSP
Note the code to multiply two nxn matrixes (A * B) together to get C, an nxn matrix

```
for $i \leftarrow 1$ to $n$
  for $j \leftarrow 1$ to $n$
    $c_{ij} \leftarrow 0$
    for $k \leftarrow 1$ to $n$
      $c_{ij} \leftarrow c_{ij} + a_{ik} * b_{kj}$
```

4. How does this matrix multiply code compare to the EXTEND code? Why do we care?
$L \rightarrow A$ $W \rightarrow B$ $L' \rightarrow C$ min $\rightarrow$ + + $\rightarrow$ * $\infty \rightarrow 0$
So, we can view EXTEND as just like matrix multiplication
Why do we care?
Because our goal is to compute $L^{(n-1)}$ as fast as we can. Don't need to compute *all* the intermediate $L^{(1)}, L^{(2)}, L^{(3)}, \ldots, L^{(n-2)}$.
Suppose we had a matrix A and we wanted to compute A^{n-1} (like calling EXTEND n-1 times).
Could compute $A, A^2, A^4, A^8, \ldots$
If we knew $A^m = A^{n-1}$ for all $m \geq n$. 1, could just finish with A^r, where r is the smallest power of 2 that's $\geq n$-1.
(r = $^{2\lg(n-1)}$)

Faster All-Pairs Shortest Paths Algorithm
Compute a solution bottom-up: Compute $L^{(1)} = W$, *then* $L^{(2)}$ *from* $L^{(1)}$, *then* $L^{(4)}$ *from* $L^{(4)}$ *etc. . . ,* $L^{(n-1)}$

```
FASTER-APSP$(W, n)$
  $L^{(1)} \leftarrow W$
  $m \leftarrow 1$
  while $m < n-1$
    $L^{(2m)} \leftarrow$EXTEND$(L^{(m)}, L^{(m)}, n)$
    $m \leftarrow 2m$
  return $L^{(m)}$
```

5. What is the runtime of FASTER-APSP?
$O(n^3 \lg n)$

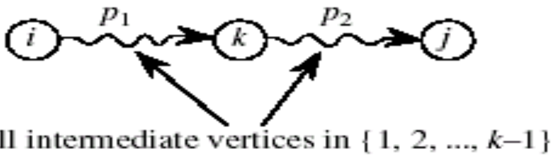
Floyd-Warshall Algorithm

To help us develop another dynamic programming approach we can restate the All-Pairs Shortest Paths problem as follows.
Find the shortest path from every vertex to every other vertex considering at most all other vertices intermediate on the paths.

6. Define structure of optimal solution.

The shortest path between any 2 vertexes i and j with intermediate vertices in {1, 2*, . . . , k*} (any k vertices, order does not matter) is either:

- **k is not an intermediate vertex and all intermediate vertices of the path are in {1, 2, . . . , k-1}   OR**
- **k is an intermediate vertex**



7. Recursively define optimal solution and write pseudocode.

Base Case: Smallest sub-problem is path from I to j with no intermediate vertices, the adjacency matrix, W*ij*

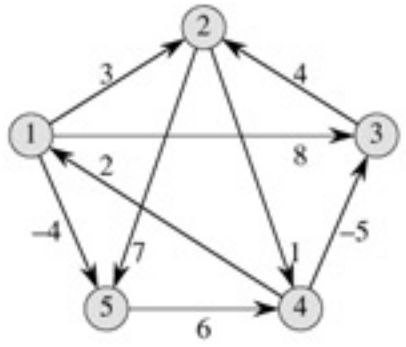
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$

Shortest path matrix $D = (d_{ij})$, where $d_{ij} = d^{(n)}_{ij}$

```
FLOYD-WARSHALL(W, n)
D(0) ← W
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            $d^{(k)}_{ij} \leftarrow \min\{ d^{(k-1)}_{ij} , d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \}$
return $D^{(n)}$
```

8. What is the run time of Floyd-Warshall? ***Time: (n^3)***

9. Demonstrate Floyd-Warshall.



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$