GRAPHS

An Introduction

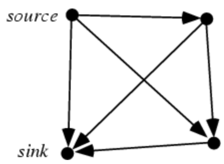
OULINE

- What are Graphs?
- Applications
- Terminology and Problems
- Representation (Adj. Mat and Linked Lists)
- Searching
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

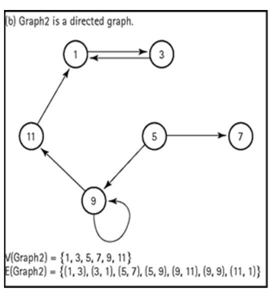
GRAPHS

- A **graph G** = **(**V,E**)** is composed of:
  - V: set of **vertices**
  - $E \subset V \times V$: set of **edges** connecting the **vertices**
- An **edge *e*** = (*u*,*v*) is a __ pair of vertices
  - Directed graphs (ordered pairs)
  - Undirected graphs (unordered pairs)

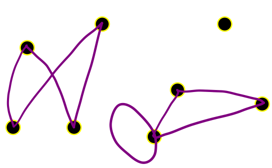
DIRECTED GRAPH



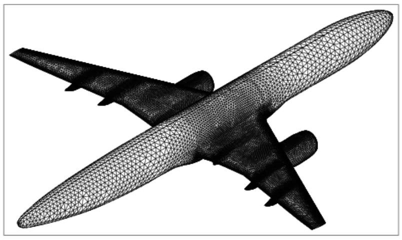
DIRECTED GRAPH



UNDIRECTED GRAPH



UNDIRECTED GRAPH



APPLICATIONS

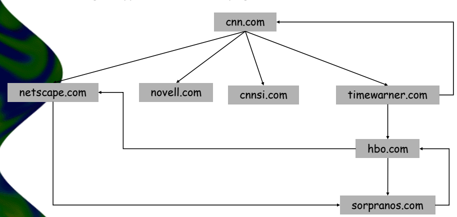
- Air Flights, Road Maps, Transportation.
- Graphics / Compilers
- Electrical Circuits
- Networks
- Modeling any kind of relationships (between people/web pages/cities/…)

SOME MORE GRAPH APPLICATIONS

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

- Web graph.
  - Node: web page.
  - Edge: hyperlink from one page to another.



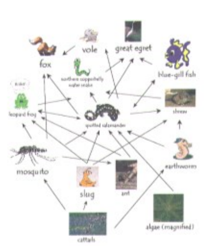
9-11 Terrorist Network

- Social network graph.
  - Node: people.
  - Edge: relationship between two people.

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

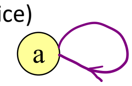
- Food web graph.
  - Node = species.
  - Edge = from prey to predator.



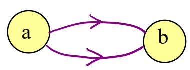
Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

TERMINOLOGY

- **a** is adjacent to **b** iff (**a**,**b**) $\in$ E.
- *degree*(a) = number of adjacent vertices (Self loop counted twice)
- Self Loop: (a,a)

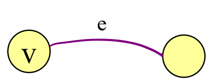


- Parallel edges: E = { ...(a,b), (a,b)...}

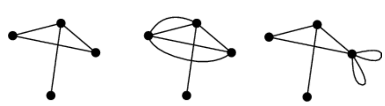


TERMINOLOGY

- A Simple Graph is a graph with no self loops or parallel edges.
- Incidence: v is incident to e if v is an end vertex of e.



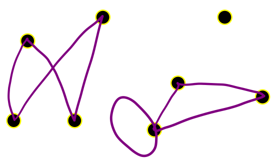
MORE...



simple graph *multigraph* *pseudograph*

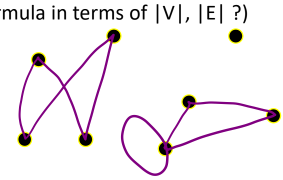
QUESTION

- Max Degree node? Min Degree Node? Isolated Nodes? Total sum of degrees over all vertices? Number of edges?



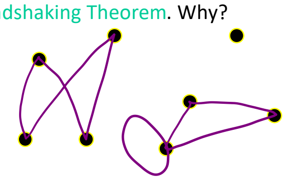
QUESTION

- Max Degree = 4. Isolated vertices = 1.
- |V| = 8 , |E| = 8
- Sum of degrees = 16 = ?
  - (Formula in terms of |V|, |E| ?)



QUESTION

- Max Degree = 4. Isolated vertices = 1.
- |V| = 8 , |E| = 8
- Sum of degrees = 2|E| = $\sum_{v \in V}$ degree(v)
  - Handshaking Theorem. Why?



QUESTION

- How many edges are there in a graph with 100 vertices each of degree 4?

QUESTION

- How many edges are there in a graph with 100 vertices each of degree 4?
  - Total degree sum = 400 = 2 |E|
  - 200 edges by the handshaking theorem.

HANDSHAKING:COROLLARY

The number of vertices with odd degree is always even.

Proof: Let V_1 and V_2 be the set of vertices of even and odd degrees, respectively

(Hence $V_1 \cap V_2 = \varnothing$, and $V_1 \cup V_2 = V$).

- Now we know that

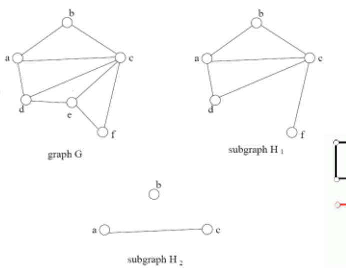
$$2|E| = \sum_{v \in V} \text{degree}(v)$$

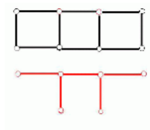
even. $$= \sum_{v \in V1} \text{degree}(v) + \sum_{v \in V2} \text{degree}(v)$$?

- Since degree(v) is odd for all $v \in V_2$, $|V_2|$ must be even.

TERMINOLOGY

A graph $H(V_H, E_H)$ is a *subgraph* of $G(V_G, E_G)$ if and only if $V_H \subset V_G$ and $E_H \subset E_G$.

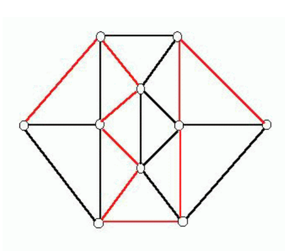




PATH AND CYCLE

- An alternating sequence of vertices and edges beginning and ending with vertices
  - each edge is incident with the vertices preceding and following it.
  - No edge appears more than once.
  - A path is ***simple*** if all nodes are distinct.
- Cycle
  - A path is a cycle if and only if $v_0 = v_k$
    - The beginning and end are the same vertex.

PATH EXAMPLE

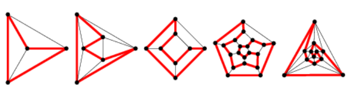


CONNECTED GRAPH

- Undirected Graphs: If there is at least one path between every pair of vertices. (otherwise disconnected)
- Directed Graphs:
  - Strongly connected
  - Weakly connected

HAMILTONIAN CYCLE

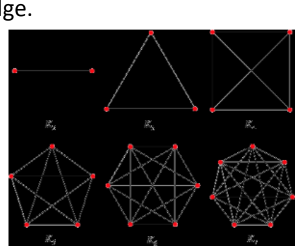
- A cycle that transverses every vertex exactly once.



In general, the problem of finding a Hamiltonian circuit is NP-Complete.

COMPLETE GRAPH

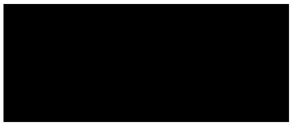
- Every pair of graph vertices is connected by an edge.



n(n-1)/2 edges

Directed Acyclic Graphs

- A DAG is a directed graph with no cycles

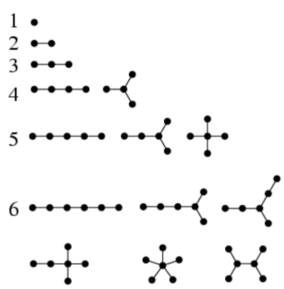


- Often used to indicate precedences among events, i.e., event *a* must happen before *b*

Tree

A connected graph with n nodes and n-1 edges

A Forest is a collection of trees.

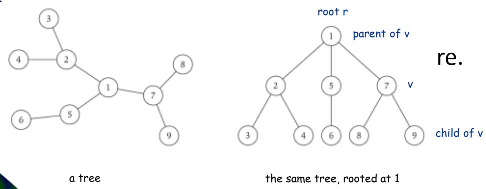


Trees

- An undirected graph is a tree if it is connected and does not contain a cycle.
- Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.

Rooted Trees

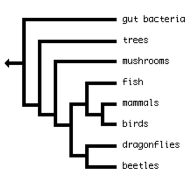
- Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.



re.

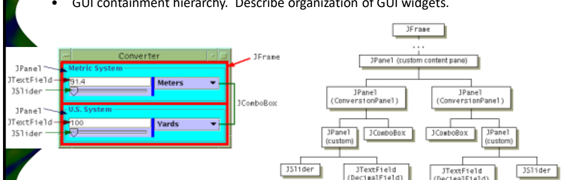
Phylogeny Trees

- Phylogeny trees. Describe evolutionary history of species.



GUI Containment Hierarchy

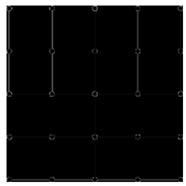
- GUI containment hierarchy. Describe organization of GUI widgets.



Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

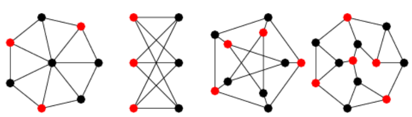
SPANNING TREE

Connected subset of a graph G with n-1 edges which contains all of V



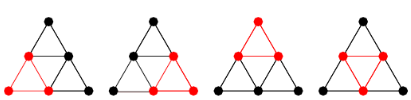
INDEPENDENT SET

- An independent set of *G* is a subset of the vertices such that no two vertices in the subset are adjacent.



CLIQUES

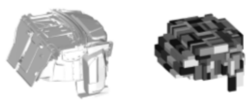
- a.k.a. complete subgraphs.



cliques

TOUGH PROBLEM
IS

- Find the maximum cardinality independent set of a graph G.
  - NP-Complete
  - Unknown if a poly time algorithm exists unless P = NP.



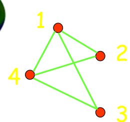
TOUGH PROBLEM
TSP

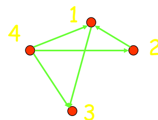
- Given a weighted graph G, the nodes of which represent cities and weights on the edges, distances; find the shortest tour that takes you from your home city to all cities in the graph and back.
  - Can be solved in O(n!) by enumerating all cycles of length n.
  - Dynamic programming can be used to reduce it in $O(n^2 2^n)$.

REPRESENTATION

- Two ways
  - Adjacency List
    - (as a linked list for each node in the graph to represent the edges)
  - Adjacency Matrix
    - (as a boolean matrix)

Representing Graphs

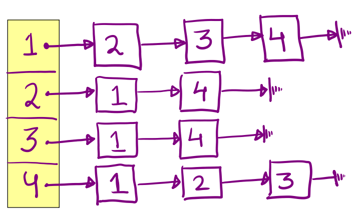




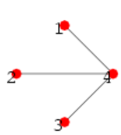
Vertex	Adjacent Vertices
1	2, 3, 4
2	1, 4
3	1, 4
4	1, 2, 3

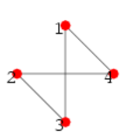
Initial Vertex	Terminal Vertices
1	3
2	1
3	
4	1, 2, 3

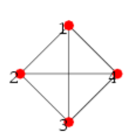
Adjacency List



Adjacency Matrix



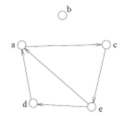




$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

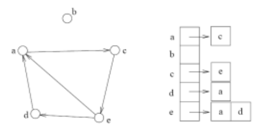
ANOTHER EXAMPLE

1. Adjacency Matrix



	a	b	c	d	e
a	0	0	1	0	0
b	0	0	0	0	0
c	0	0	0	0	1
d	1	0	0	0	0
e	1	0	0	1	0

2. Adjacency List



AL VS AM

- AL: Takes O(|V| + |E|) space
- AM: Takes O(|V|*|V|) space
- Question: How much time does it take to find out if (v_i, v_j) belongs to E?
  - AM ?
  - AL ?

AL VS AM

- AL: Takes O(|V| + |E|) space
- AM: Takes O(|V|*|V|) space
- Question: How much time does it take to find out if (v_i, v_j) belongs to E?
  - AM : O(1)
  - AL : O(|V|) in the worst case.

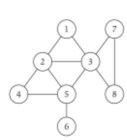
AL VS AM

- AL : Total space = 8|V| + 16|E| bytes (For undirected graphs its 8|V| + 32|E| bytes)
- AM : |V| * |V| / 8
- Question: What is better for very sparse graphs? (Few number of edges)

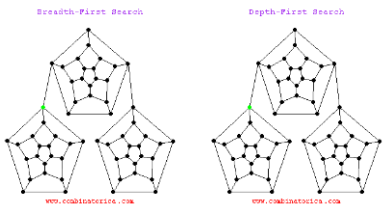
GRAPH TRAVERSAL

CONNECTIVITY

- s-t connectivity problem. Given two node s and t, is there a path between s and t?
- s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?
- Applications.
  – Maze traversal.
  – Kevin Bacon number / Erdos number
  – Fewest number of hops in a communication network.
  – Friendster.



BFS/DFS



BFS : Breadth First Search
DFS : Depth First Search

BFS/DFS

- Breadth-first search (BFS) and depth-first search (DFS) are two distinct orders in which to visit the vertices and edges of a graph.
- BFS: radiates out from a root to visit vertices in order of their distance from the root. Thus closer nodes get visited first.

Breadth First Search

- Question: Given G in AM form, how do we say if there is a path between nodes a and b?
- Note: Using AM or AL its easy to answer if there is an edge (a,b) in the graph, but not path questions. This is one of the reasons to learn BFS/DFS.

BFS

- A **Breadth-First Search (BFS)** traverses a **connected component** of a graph, and in doing so defines a **spanning tree**.

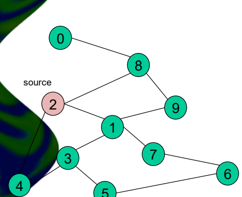
Source: Lecture notes by **Sheung-Hung POON**

BFS

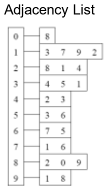
Algorithm *BFS*(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
   do v := dequeue(Q);
      for each w adjacent to v
         do if flag[w] = false
               then flag[w] := true;
                    enqueue(Q, w)
```

EXAMPLE



Adjacency List



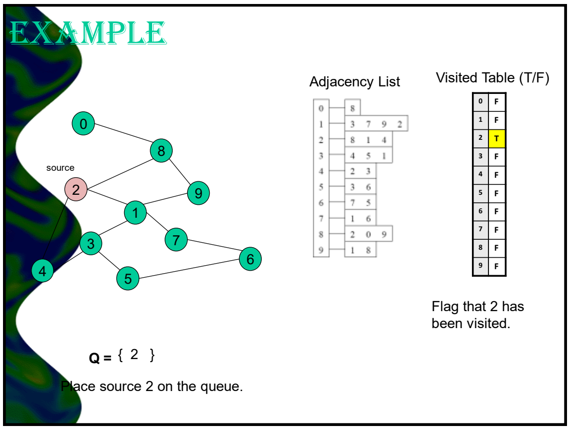
Visited Table (T/F)

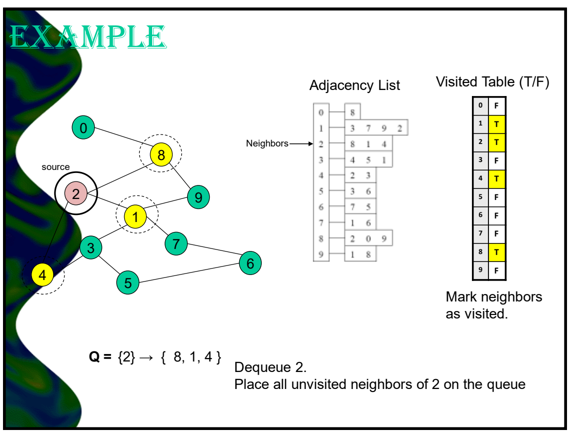
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

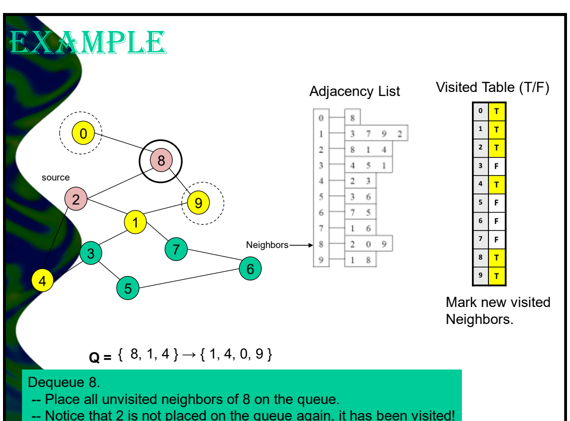
Initialize visited table (all empty F)

Q = { }

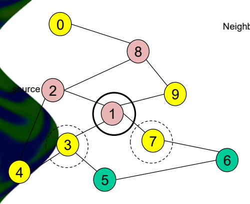
Initialize **Q** to be empty


EXAMPLE
Adjacency List
Visited Table (T/F)
source
Flag that 2 has been visited.
Q = { 2 }
Place source 2 on the queue.


EXAMPLE
Adjacency List
Visited Table (T/F)
Neighbors
source
Mark neighbors as visited.
Q = {2} → { 8, 1, 4 }
Dequeue 2.
Place all unvisited neighbors of 2 on the queue


EXAMPLE
Adjacency List
Visited Table (T/F)
source
Neighbors
Mark new visited Neighbors.
Q = { 8, 1, 4 } → { 1, 4, 0, 9 }
Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!

EXAMPLE



Adjacency List

Neighbors →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

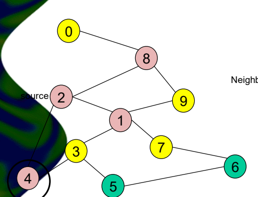
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Mark new visited Neighbors.

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been visited yet.

EXAMPLE



Adjacency List

Neighbors →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

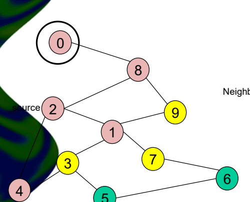
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
-- 4 has no unvisited neighbors!

EXAMPLE



Adjacency List

Neighbors →

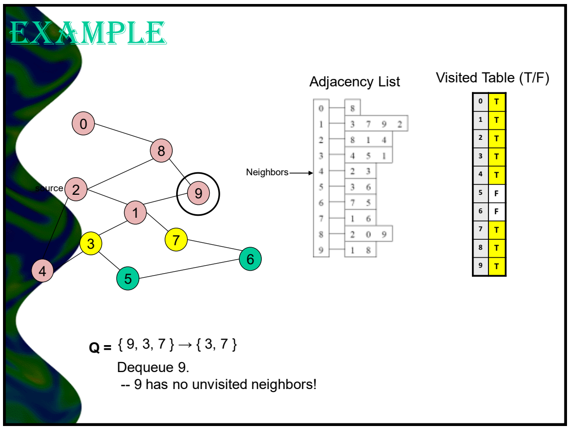
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

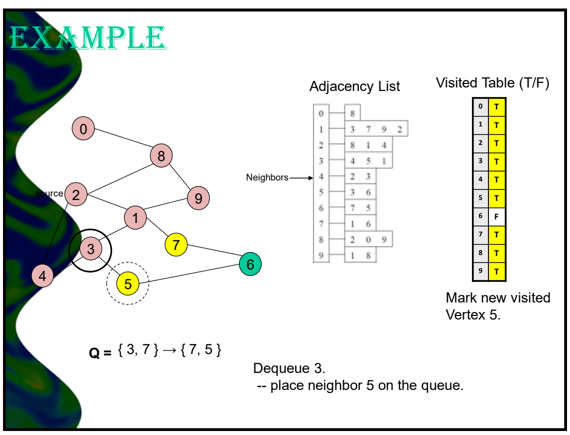
Visited Table (T/F)

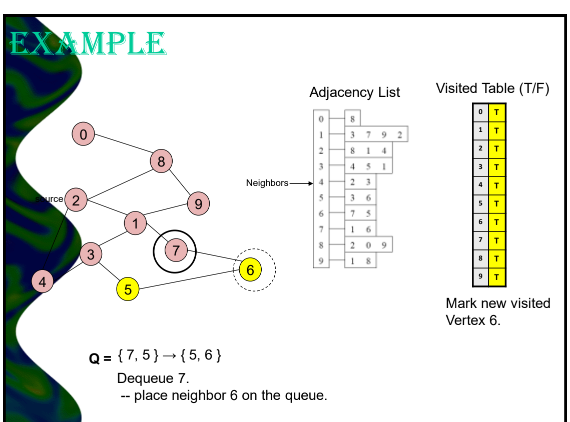
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

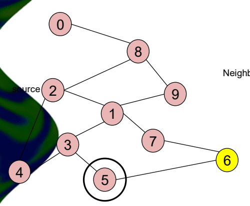
Dequeue 0.
-- 0 has no unvisited neighbors!


EXAMPLE
Adjacency List
Visited Table (T/F)
Neighbors
Q = { 9, 3, 7 } → { 3, 7 }
Dequeue 9.
-- 9 has no unvisited neighbors!


EXAMPLE
Adjacency List
Visited Table (T/F)
Neighbors
Mark new visited
Vertex 5.
Q = { 3, 7 } → { 7, 5 }
Dequeue 3.
-- place neighbor 5 on the queue.


EXAMPLE
Adjacency List
Visited Table (T/F)
Neighbors
Mark new visited
Vertex 6.
Q = { 7, 5 } → { 5, 6 }
Dequeue 7.
-- place neighbor 6 on the queue.

EXAMPLE



Adjacency List

Neighbors → 4

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

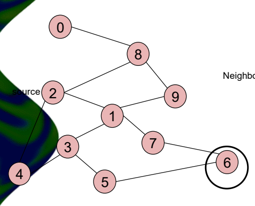
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

Dequeue 5.

-- no unvisited neighbors of 5.

EXAMPLE



Adjacency List

Neighbors → 4

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

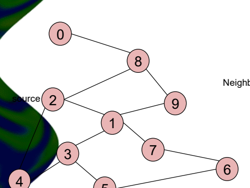
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 6 } → { }

Dequeue 6.

-- no unvisited neighbors of 6.

EXAMPLE



Adjacency List

Neighbors → 4

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

What did we discover?

Look at "visited" tables.

There exist a path from source vertex 2 to all vertices in the graph!

Q = { } **STOP!!! Q is empty!!!**

Time Complexity of BFS
(Using adjacency list)

- Assume adjacency list
  - n = number of vertices m = number of edges

```
Algorithm BFS(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
   do v := dequeue(Q);
      for each w adjacent to v
         do if flag[w] = false
               then flag[w] := true;
                    enqueue(Q, w)
```

O(n + m)

No more than n vertices are ever put on the queue.

How many adjacent nodes will we ever visit. This is related to the number of edges. How many edges are there?

$\Sigma_{vertex\ v} deg(v) = 2m$*

*Note: this is not per iteration of the while loop. This is the sum over all the while loops!

Time Complexity of BFS
(Using adjacency matrix)

- Assume adjacency matrix
  - n = number of vertices m = number of edges

```
Algorithm BFS(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
   do v := dequeue(Q);
      for each w adjacent to v
         do if flag[w] = false
               then flag[w] := true;
                    enqueue(Q, w)
```

O(n²)

So, adjacency matrix is not good for BFS!!!

No more than n vertices are ever put on the queue. O(n)

Using an adjacency matrix. To find the neighbors we have to visit all elements In the row of v. That takes time O(n).

Path Recording

- BFS only tells us if a path exists from source s, to other vertices v.
  - It doesn't tell us the path!
  - We need to modify the algorithm to record the path.
- Not difficult
  - Use an additional predecessor array *pred[0..n-1]*
  - *Pred[w] = v*
    - *Means that vertex w was visited by v*

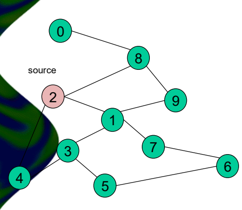
BFS + Path Finding

```
Algorithm BFS(s)
    for each vertex v
        do flag(v) := false;
           pred[v] := −1;
    Q = empty queue;
    flag[s] := true;
    enqueue(Q, s);
    while Q is not empty
      do v := dequeue(Q);
         for each w adjacent to v
             do if flag[w] = false
                  then flag[w] := true;
                       pred[w] := v;
                       enqueue(Q, w)
```

Set pred[v] to -1 (let -1 means no path to any vertex)

Record who visited w

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

		Pred
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

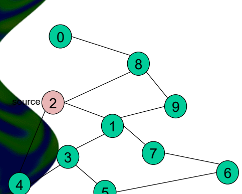
Initialize visited table (all empty F)

Initialize Pred to -1

Q = { }

Initialize **Q** to be empty

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

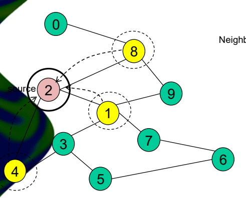
		Pred
0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.

EXAMPLE



Adjacency List

Neighbors → 2

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

	Visited	*Pred*
0	F	-
1	T	2
2	T	-
3	F	-
4	T	2
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

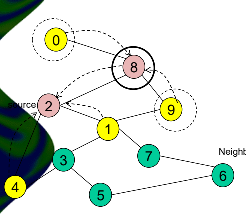
Mark neighbors as visited.

Record in Pred who was visited by 2.

Q = {2} → { 8, 1, 4 }

Dequeue 2.
Place all unvisited neighbors of 2 on the queue

EXAMPLE



Adjacency List

Neighbors → 8

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

	Visited	*Pred*
0	T	8
1	T	2
2	T	-
3	F	-
4	T	2
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

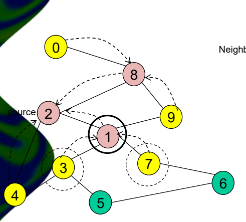
Mark new visited Neighbors.

Record in Pred who was visited by 8.

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!

EXAMPLE



Adjacency List

Neighbors → 1

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

	Visited	*Pred*
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

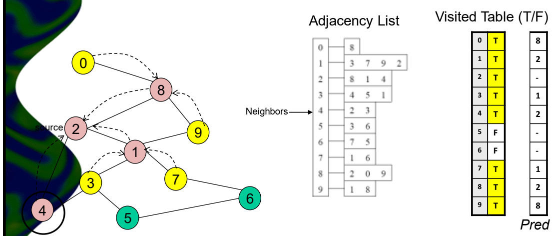
Mark new visited Neighbors.

Record in Pred who was visited by 1.

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been visited yet.

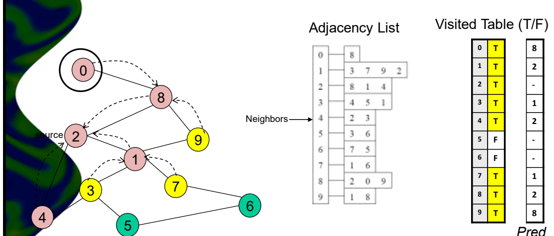
EXAMPLE



Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
– 4 has no unvisited neighbors!

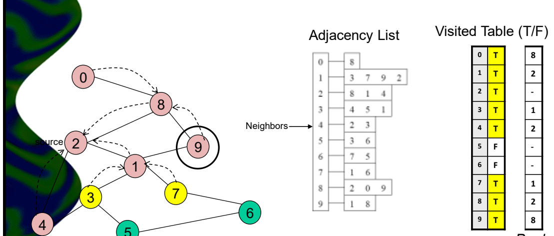
EXAMPLE



Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.
– 0 has no unvisited neighbors!

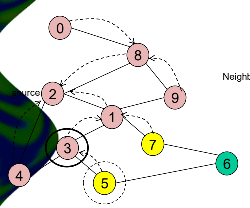
EXAMPLE



Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
– 9 has no unvisited neighbors!

EXAMPLE



Adjacency List

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	F	-
7	T	1
8	T	2
9	T	8

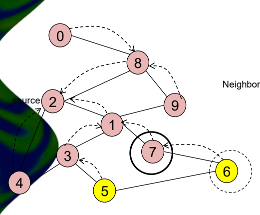
Mark new visited Vertex 5.

Record in Pred who was visited by 3.

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.

EXAMPLE



Adjacency List

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

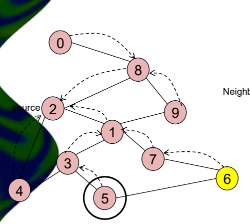
Mark new visited Vertex 6.

Record in Pred who was visited by 7.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue.

EXAMPLE



Adjacency List

Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

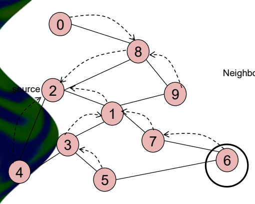
Visited Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Q = { 5, 6} → { 6 }

Dequeue 5.
-- no unvisited neighbors of 5.

EXAMPLE



Adjacency List

Neighbors →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

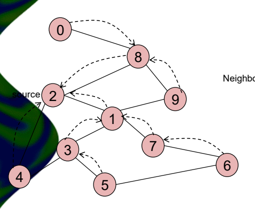
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

Q = { 6 } → { }

Dequeue 6.
- no unvisited neighbors of 6.

EXAMPLE



Adjacency List

Neighbors →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

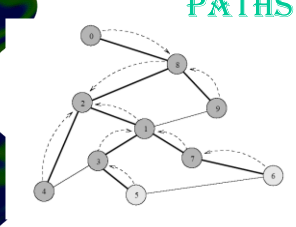
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

Pred now stores the path!

Q = { } **STOP!!! Q is empty!!!**

PRED ARRAY REPRESENTS PATHS



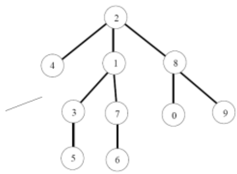
nodes	visited by
0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

Algorithm *Path*(w)
1. **if** $pred[w] \neq -1$
2. **then**
3. *Path*($pred[w]$);
4. output w

Try some examples.
Path(0) ->
Path(6) ->
Path(1) ->

BFS TREE

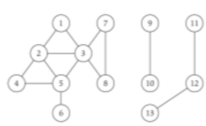
- We often draw the BFS paths as a m-ary tree, where ***s*** is the root.



Question: What would a “level” order traversal tell you?

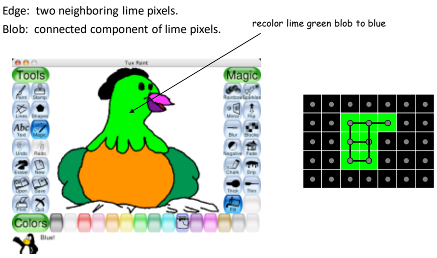
CONNECTED COMPONENT

- Connected component. Find all nodes reachable from s.



FLOOD FILL

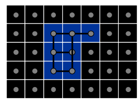
- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.



Flood Fill

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.

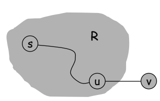
recolor lime green blob to blue



Connected Component

- Connected component. Find all nodes reachable from s.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u,v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



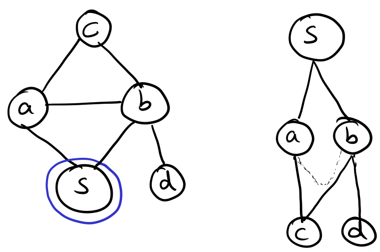
it's safe to add v

More on Paths and Trees in Graphs

BFS

- Another way to think of the BFS tree is the physical analogy of the BFS Tree.
- Sphere-String Analogy : Think of the nodes as spheres and edges as unit length strings. Lift the sphere for vertex **s.**

SPHERE-STRING ANALOGY

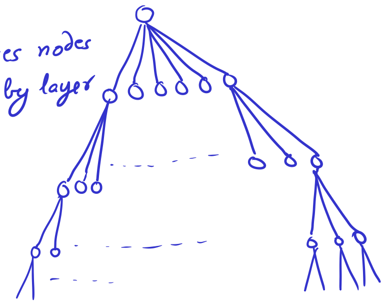


BFS : PROPERTIES

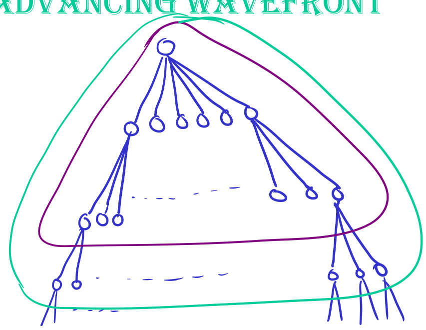
- At some point in the running of BFS, **Q** only contains vertices/nodes at layer **d.**
- If **u** is removed before **v** in BFS then
  - dist(u) $\leq$ dist(v)
- At the end of BFS, for each vertex **v** reachable from **s**, the dist(v) equals the shortest path length from s to v.

BFS

Processes nodes
layer by layer



BFS:Advancing Wavefront



Old Wine in New Bottle

```
forall v ε V:
   dist(v) = ∞; prev(v) = null;
dist(s) = 0
Queue q; q.push(s);
while (!Q.empty())
    v = Q.dequeue();
    for all e=(v,w) in E
        if dist(w) = ∞:
            – dist(w) = dist(v)+1
            – Q.enque(w)
            – prev(w)= v
```

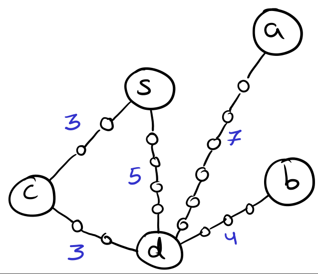
Dijkstra's SSSP Alg BFS with positive int weights

- for every edge e=(a,b) ε E, let w_e be the weight associated with it. Insert w_e-1 dummy nodes between a and b. Call this new graph G'.
- Run BFS on G'. dist(u) is the shortest path length from s to node u.
- Why is this algorithm bad?

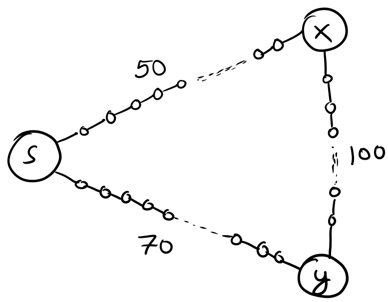
How do we speed it up?

- If we could run BFS without actually creating G', by somehow simulating BFS of G' on G directly.
- Solution: Put a system of alarms on all the nodes. When the BFS on G' reaches a node of G, an alarm is sounded. Nothing interesting can happen before an alarm goes off.

An Example



Another Example



Alarm Clock Alg

alarm(s) = 0

until no more alarms

- wait for an alarm to sound. Let next alarm that goes off is at node v at time t.
  - dist(s,v) = t
  - for each neighbor w of v in G:
    - If there is no alarm for w, alarm(w) = t+weight(v,w)
    - If w's alarm is set further in time than t+weight(v,w), reset it to t+weight(v,w).

Recall BFS

forall v ε V:

  dist(v) = ∞; prev(v) = null;

dist(s) = 0

Queue q; q.push(s);

while (!Q.empty())

  v = Q.dequeue();

  for all e=(v,w) in E

    if dist(w) = ∞:

- dist(w) = dist(w)+1
- Q.enque(w)
- prev(w)= v

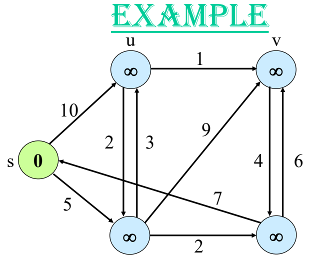
DIJKSTRA'S SSSP

forall v ε V:
  dist(v) = ∞; prev(v) = null;
dist(s) = 0
Magic_DS Q; Q.insert(s,0);
while (!Q.empty())
  v = Q.delete_min();
  for all e=(v,w) in E
    if dist(w) > dist(v)+weight(v,w) :
      – dist(w) = dist(v)+weight(v,w)
      – Q.insert(w, dist(w))
      – prev(w)= v

THE MAGIC DS: PQ

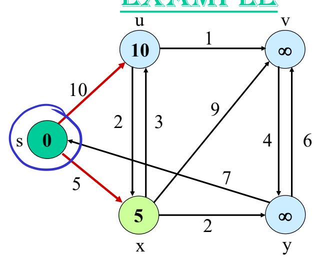
- What functions do we need?
  - insert() : Insert an element and its key. If the element is already there, change its key (only if the key decreases).
  - delete_min() : Return the element with the smallest key and remove it from the set.

EXAMPLE



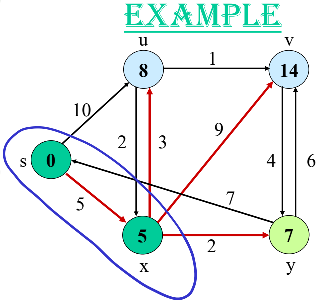
s 0
u ∞
v ∞
x ∞
y ∞

EXAMPLE



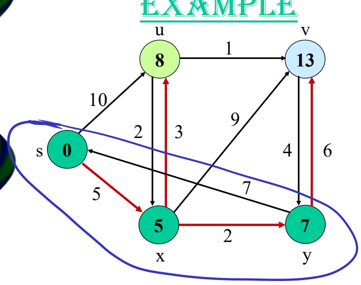
s 0
u 10
v ∞
x 5
y ∞

EXAMPLE

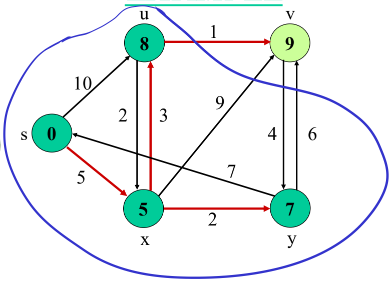


s 0
u 8
v 14
x 5
y 7

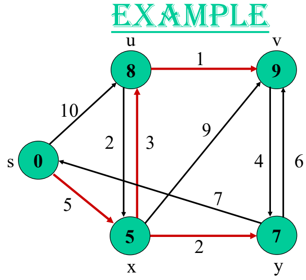
EXAMPLE



EXAMPLE



EXAMPLE



ANOTHER VIEW REGION GROWTH

1. Start from s
2. Grow a region R around s such that the SPT from s is known inside the region.
3. Add v to R such that v is the closest node to s outside R.
4. Keep building this region till R = V.

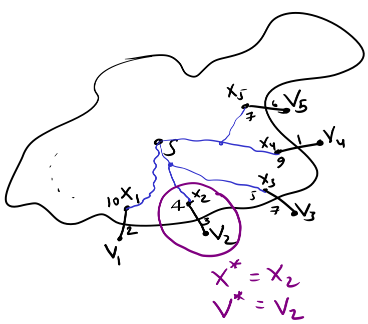
HOW DO WE FIND V?

Pick $v \notin R$ s.t.

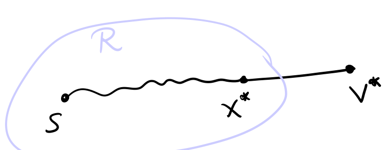
$$\min_{x \in R} \; dist(s,x) + weight(x,v)$$

Let (x^*, v^*) be the opt.

EXAMPLE



S,V*



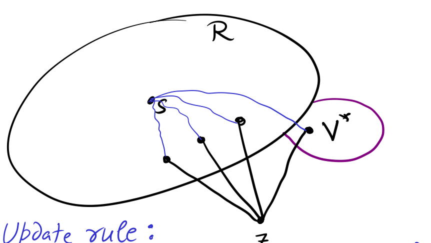
Is this the shortest path to V^*?

Why?

Old Wine in New Bottle

```
forall v ε V:
    dist(v) = ∞; prev(v) = null;
dist(s) = 0
R = {};
while R != V
    Pick v not in R with smallest distance to s
    for all edges (v,z) ε E
          if(dist(z) > dist(v) + weight(v,z)
                    dist(z) = dist(v)+weight(v,z)
                    prev(z) = v;
    Add v to R
```

Updates



Update rule:

(Best way to reach z?)

Running Time?

delete_min = ?

insert = ?

Running Time?

$$\text{delete\_min} = |V|$$
$$\text{insert} = |E|$$

Running Time?

- If we used a linked list as our magic data structure?

$$\text{delete\_min}() \rightarrow O(|V|)$$
$$\text{insert}() \rightarrow \cancel{O(1)}\ O(|V|)$$
$$\text{Total} = |V|\ \text{delete\_min}() + |E|\ \text{insert}() = \cancel{O(|V|^2)}\ |V|^3$$

Binary Heap?

$$\text{delete\_min}() \rightarrow O(\log |V|)$$
$$\text{insert}() \rightarrow O(\log |V|)$$
$$\text{Total} \rightarrow O(|E| \log |V|)$$

D-ARY HEAP

Why?

$delete\_min() \rightarrow O(d \log_d |V|)$

$insert() \rightarrow O(\log_d |V|)$

$Total \rightarrow O((|V| d + |E|) \log_d |V|)$

FIBONACCI HEAP

$delete\_min() \rightarrow O(1)$ Amortized

$insert() \rightarrow O(\log |V|)$

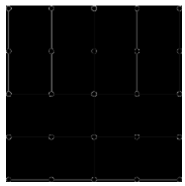
$Total \rightarrow O(|V| \log |V| + |E|)$

A SPANNING TREE

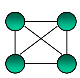
- Recall?
- Is it unique?
- Is shortest path tree a spanning tree?
- Is there an easy way to build a spanning tree for a given graph G?
- Is it defined for disconnected graphs?

SPANNING TREE

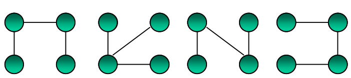
Connected subset of a graph G with n-1 edges which contains all of V.



SPANNING TREE



A connected, undirected graph



Some spanning trees of the graph

EASY ALGORITHM

To build a spanning tree:

Step 1: T = one node in V, as root.

Step 2: At each step, add to tree one edge from a node in tree to a node that is not yet in the tree.

SPANNING TREE PROPERTY

Adding an edge **e**=(**a**,**b**) not in the tree creates a cycle containing only edge **e** and edges in spanning tree.

Why?

SPANNING TREE PROPERTY

- Let c be the first node common to the path from a and b to the root of the spanning tree.
- The concatenation of (a,b) (b,c) (c,a) gives us the desired cycle.

LEMMA 1

- In any tree, T = (V,E),
  |E| = |V| - 1
- Why?

Lemma 1

- In any tree, T = (V,E),
  |E| = |V| - 1
- Why?
- Tree T with 1 node has zero edges.
- For all n>0, P(n) holds, where
- P(n) : A Tree with n nodes has n-1 edges.
- Apply MI. How do we prove that given P(m) true for all 1..m, P(m+1) is true?

Undirected Graphs n Trees

- An undirected graph G = (V,E) is a tree iff
  (1) it is connected
  (2) |E| = |V| – 1

Lemma 2

Let C be the cycle created in a spanning tree T by adding the edge e = (a,b) not in the tree. Then removing any edge from C yields another spanning tree.

Why? How many edges and vertices does the new graph have? Can (x,y) in G get disconnected in this new tree?

Lemma 2

- Let T' be the new graph
- T' has n nodes and n-1 edges, so it must be a tree if it is connected.
- Let (x,y) be not connected in T'. The only problem in the connection can be the removed edge (a,b). But if (a,b) was contained in the path from x to y, we can use the cycle C to reach y (even if (a,b) was deleted from the graph).

Weighted Spanning Trees

Let w_e be the weight of an edge e in G=(V,E).

Weight of spanning tree = Sum of edge weights.

Question: How do we find the spanning tree with minimum weight.
This spanning tree is also called the Minimum Spanning Tree.

Is the MST unique?

Minimum Spanning Trees

- Applications
  - networks
  - cluster analysis
    - used in graphics/pattern recognition
  - approximation algorithms (TSP)
  - bioinformatics/CFD

CUT PROPERTY

- Let X be a subset of V. Among edges crossing between X and V \ X, let e be the edge of minimum weight. Then e belongs to the MST.
- Proof?

CYCLE PROPERTY

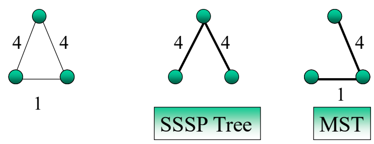
- For any cycle C in a graph, the heaviest edge in C does not appear in the MST.
- Proof?

QUESTION

- Is the SSSP Tree and the Minimum spanning tree the same?
- Is one the subset of the other always?

QUESTION

- Is the SSSP Tree and the Minimum spanning tree the same?
- Is one the subset of the other always?



OLD WINE IN NEW BOTTLE

```
forall v ε V:
   dist(v) = ∞; prev(v) = null;
dist(s) = 0
Heap Q; Q.insert(s,0);
while (!Q.empty())
    v = Q.delete_min();
    for all e=(v,w) in E
        if dist(w) > dist(v)+weight(v,w)  :
            – dist(w) = dist(v)+weight(v,w)
            – Q.insert(w, dist(w))
            – prev(w)= v
```

A SLIGHT MODIFICATION JARNIK'S OR PRIM'S ALG.

forall v ε V:

dist(v) = ∞; prev(v) = null;

dist(s) = 0

Heap Q; Q.insert(s,0);

while (!Q.empty())

v = Q.delete_min();

for all e=(v,w) in E

if dist(w) > ~~dist(v)+~~ weight(v,w) :

- dist(w) = ~~dist(v) +~~ weight(v,w)
- Q.insert(w, dist(w))
- prev(w)= v

Our First MST Alg.

forall v ε V:
  dist(v) = ∞; prev(v) = null;
dist(s) = 0
Magic_DS Q; Q.insert(s,0);
while (!Q.empty())
  v = Q.delete_min();
  for all e=(v,w) in E
    if dist(w) > weight(v,w) :
    – dist(w) = weight(v,w)
    – Q.insert(w, dist(w))
    – prev(w)= v

How does the running time depend on the Magic_DS?

- heap?
- insert()?
- delete_min()?
- Total time?
- What if we change the Magic_DS to fibonacci heap?

Prim's/Jarnik's Algorithm

- best running time using fibonacci heaps
  – O(E + VlogV)
- Why does it compute the MST?

Another Alg: Krushkal's

- sort the edges of G in increasing order of weights
- Let S = {}
- for each edge e in G in sorted order
  - if the endpoints of e are disconnected in S
  - Add e to S

Have U seen this before?

- Sort edges of G in increasing order of weight
- T = {}   // Collection of trees
- For all e in E
  - If T union {e} has no cycles in T
    - then T = T union {e}

return T

Naïve running time O((|V|+|E|)|V|) = O(|E||V|)

How to speed it up?

- To O(E + VlogV)
  - Using union find data structures.
- Surprisingly the idea is very simple.

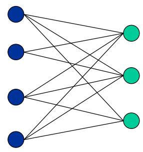
Other Applications

3.4 Testing Bipartiteness

Bipartite Graphs

- Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

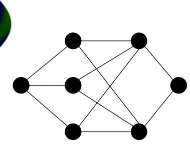
- Applications.
  – Stable marriage: men = red, women = blue.
  – Scheduling: machines = red, jobs = blue.



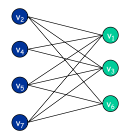
a bipartite graph

Testing Bipartiteness

- Testing bipartiteness. Given a graph G, is it bipartite?
  - Many graph problems become:
    - easier if the underlying graph is bipartite (matching)
    - tractable if the underlying graph is bipartite (independent set)
  - Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



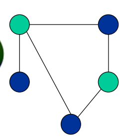
a bipartite graph G



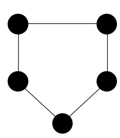
another drawing of G

An Obstruction to Bipartiteness

- Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.
- Pf. Not possible to 2-color the odd cycle, let alone G.



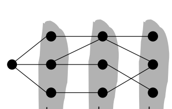
bipartite
(2-colorable)



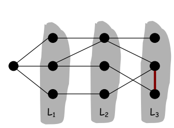
not bipartite
(not 2-colorable)

Bipartite Graphs

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  (ii) An edge of G joins two nodes of the same layer, and G contains an
  odd-length cycle (and hence is not bipartite).



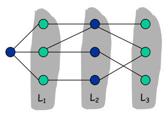
Case (i)



Case (ii)

Bipartite Graphs

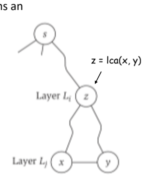
- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  - (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- Pf. (i)
  - Suppose no edge joins two nodes in the same layer.
  - By previous lemma, this implies all edges join nodes on same level.
  - Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

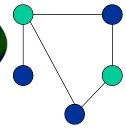
Bipartite Graphs

- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  - (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- Pf. (ii)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let z = lca(x, y) = lowest common ancestor.
  - Let L_i be level containing z.
  - Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
  - Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from y to z}} + \underbrace{(j-i)}_{\text{path from z to x}}$, which is odd. ▪

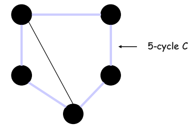


Obstruction to Bipartiteness

- Corollary. A graph G is bipartite iff it contain no odd length cycle.



bipartite
(2-colorable)

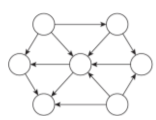


not bipartite
(not 2-colorable)

3.5 Connectivity in Directed Graphs

Directed Graphs

- Directed graph. G = (V, E)
  - Edge (u, v) goes from node u to node v.



- Ex. Web graph - hyperlink points from one web page to another.
  - Directedness of graph is crucial.
  - Modern web search engines exploit hyperlink structure to rank web pages by importance.

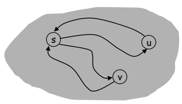
Graph Search

- Directed reachability. Given a node s, find all nodes reachable from s.
- Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?
- Graph search. BFS extends naturally to directed graphs.
- Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

- Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.
- Def. A graph is strongly connected if every pair of nodes is mutually reachable.
- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.
- Pf. ⇒ Follows from definition.
- Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.
  Path from v to u: concatenate v-s path with s-u path. ▪

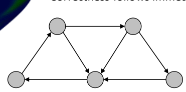
ok if paths overlap



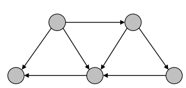
Strong Connectivity: Algorithm

- Theorem. Can determine if G is strongly connected in O(m + n) time.
- Pf.
  - Pick any node s.
  - Run BFS from s in G.
  - Run BFS from s in G^{rev}.
  - Return true iff all nodes reached in both BFS executions.
  - Correctness follows immediately from previous lemma. ▪

reverse orientation of every edge in G



strongly connected

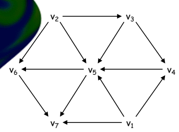


not strongly connected

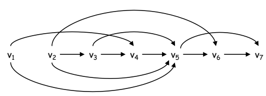
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

- Def. An DAG is a directed graph that contains no directed cycles.
- Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.
- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



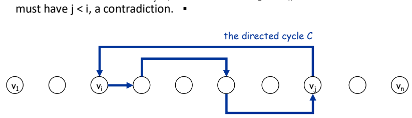
a topological ordering

Precedence Constraints

- Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.
- Applications.
  - Course prerequisite graph: course v_i must be taken before v_j.
  - Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Directed Acyclic Graphs

- Lemma. If G has a topological order, then G is a DAG.
- Pf. (by contradiction)
  - Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
  - Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
  - By our choice of i, we have $i < j$.
  - On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



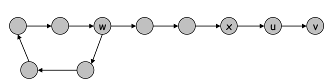
the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

- Lemma. If G has a topological order, then G is a DAG.
- Q. Does every DAG have a topological ordering?
- Q. If so, how do we compute one?

Directed Acyclic Graphs

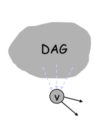
- Lemma. If G is a DAG, then G has a node with no incoming edges.
- Pf. (by contradiction)
  - Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
  - Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
  - Then, since u has at least one incoming edge (x, u), we can walk backward to x.
  - Repeat until we visit a node, say w, twice.
  - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



Directed Acyclic Graphs

- Lemma. If G is a DAG, then G has a topological ordering.
- Pf. (by induction on n)
  - Base case: true if n = 1.
  - Given DAG on n > 1 nodes, find a node v with no incoming edges.
  - G - { v } is a DAG, since deleting v cannot create cycles.
  - By inductive hypothesis, G - { v } has a topological ordering.
  - Place v first in topological ordering; then append nodes of G - { v }
  - in topological order. This is valid since v has no incoming edges. ▪

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```



Topological Sorting Algorithm: Running Time

- Theorem. Algorithm finds a topological order in O(m + n) time.

- Pf.
  - Maintain the following information:
    - `count[w]` = remaining number of incoming edges
    - S = set of remaining nodes with no incoming edges
  - Initialization: O(m + n) via single scan through graph.
  - Update: to delete v
    - remove v from S
    - decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
    - this is O(1) per edge ▪